C-PathGen: An Efficient CPU-Parallel Critical Path Generation Algorithm

CHE CHANG, University of Wisconsin-Madison, Madison, United States
BOYANG ZHANG, University of Wisconsin-Madison, Madison, United States
CHENG-HSIANG CHIU, University of Wisconsin-Madison, Madison, United States
DIAN-LUN LIN, IBM, Armonk, United States
YI-HUA CHUNG, University of Wisconsin-Madison, Madison, United States
WAN-LUAN LEE, Cadence Design Systems Inc, San Jose, United States
ZIZHENG GUO, Peking University, Beijing, China
YIBO LIN, Peking University, Beijing, China
TSUNG-WEI HUANG, University of Wisconsin-Madison, Madison, United States

Critical Path Generation (CPG) is fundamental for many static timing analysis (STA) applications. As the circuit complexity continues to increase, CPG runtime has quickly become the bottleneck due to its time-consuming and iterative nature. Despite many CPG algorithms introduced by existing timers, nearly all of them are limited to *a single CPU thread*, leading to long runtime for large CPG queries. To mitigate this runtime challenge, we need a parallel CPG algorithm. However, designing a parallel CPG algorithm is very challenging because we need to strategically partition the path search space into multiple groups that can run in parallel while accommodating different slack priorities. To overcome this challenge, we propose *C-PathGen*, an exact and efficient CPU-parallel CPG algorithm. Building on our prior CPU-parallel CPG algorithm, PathGen, C-PathGen overcomes its inaccuracy by filtering out non-critical paths. Additionally, to eliminate contention caused by atomic operations on generated paths, we introduce a new exploration strategy that processes the paths directly in place. Compared to PathGen, C-PathGen is 6.8×–10.3× faster while producing exact results when dealing with large path queries on industrial circuit graphs.

1 Introduction

Critical Path Generation (CPG) is an important step for static timing analysis (STA) applications [2] to analyze the timing criticality of a circuit design [4, 5, 25, 28, 40, 52, 96, 98]. As the design complexity increases, the runtime of CPG can become a bottleneck in many STA engines [34]. To solve this problem, the STA community has developed several CPG algorithms that efficiently generate top-k critical paths. For example, iTimerC [67] introduces a branch-and-bound algorithm to remove redundant path traversals; iitRace [89] introduces a pin coloring strategy to conduct path reduction; OpenTimer [40] introduces a fast implicit path representation algorithm comprising a *suffix tree* and a *prefix tree* to speed up critical path search. Although existing CPG algorithms have demonstrated

Authors' Contact Information: Che Chang, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: che.chang@wisc.edu; Boyang Zhang, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: bzhang523@wisc.edu; Cheng-Hsiang Chiu, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: chenghsiang.chiu@wisc.edu; Dian-Lun Lin, IBM, Armonk, New York, United States; e-mail: aaron.dl.lin@ibm.com; Yi-Hua Chung, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: yihua.chung@wisc.edu; Wan-Luan Lee, Cadence Design Systems Inc, San Jose, California, United States; e-mail: wanluan@cadence.com; Zizheng Guo, Peking University, Beijing, China; e-mail: gzz@pku.edu.cn; Yibo Lin, Peking University, Beijing, China; e-mail: yibolin@pku.edu.cn; Tsung-Wei Huang, University of Wisconsin-Madison, Madison, Wisconsin, United States; e-mail: tsung-wei.huang@wisc.edu.

ACM 1557-7309/2026/8-ART
https://doi.org/10.1145/3844513

promising performance [34], *nearly all of them are limited to a single CPU thread.* For large CPG queries, their runtime can be very slow. For example, a CPG query of one million paths can take 2.5 seconds [40], where industrial STA applications typically issue thousands of CPG queries during timing-driven optimization.

To mitigate this runtime challenge, prior GPU-accelerated CPG algorithms [4, 24] expand the critical path search space in parallel. While GPU-parallel CPG algorithms achieve substantial speedup over existing algorithms, a CPU-parallel CPG algorithm needs to co-exist due to the following reasons: (1) According to our industrial partners, supporting GPU requires significant investment and involves non-trivial modifications to existing codebases compared to CPU-parallel enhancement. (2) CPG is used in the loop of many STA applications, whereas not all of them can benefit from GPU. For example, incremental CPG and timing-update workloads may not exhibit enough data parallelism to benefit from GPU acceleration [28]. Consequently, despite being an orthogonal direction to GPU, we argue that there is a need for a CPU-parallel CPG algorithm to co-enhance the performance of STA applications.

However, designing a CPU-parallel CPG algorithm is very challenging due to the following reasons: (1) We cannot simply use GPU-parallel CPG approaches [4, 24] out of the box due to the difference in parallelism models between GPUs and CPUs. For example, GPU-parallel CPG relies on massive parallelism (e.g., thousands of GPU threads) and GPU-specific path data structures to explore many critical paths at the same time. However, CPU-side data structures are fundamentally different from GPU-side data structures and often do not support as many threads as GPUs. (2) To generate paths concurrently and accurately, we need to strategically partition generated critical paths into multiple groups that can run in parallel while appropriately accommodating their slack priorities. (3) As we generate more paths, we may experience an unbalanced mix of critical paths with different slack priorities in certain partitions, which hampers both the performance and the accuracy.

To overcome these challenges, we propose *C-PathGen*, an efficient CPU-parallel CPG algorithm. Building on its predecessor *PathGen* [6], C-PathGen overcomes its inaccuracy by filtering out non-k-critical paths. We summarize three technical contributions of C-PathGen as follows:

- We design a CPU-parallel CPG algorithm that efficiently categorizes generated critical paths into multiple groups of different slack priorities. Since the paths in the same group have similar slacks, these paths share the same priority and can expand their search spaces concurrently.
- PathGen counts on a redistribution strategy that dynamically adjusts path priorities to enhance accuracy. However, this strategy cannot guarantee exact path results because it may leave some paths unexplored. To overcome this limitation, we design a new threshold-based strategy that computes a slack threshold to filter out non-k-critical paths, thus guaranteeing exactness.
- PathGen counts on a multi-level queue scheduler and atomic dequeue operations to simultaneously process generated paths from multiple threads. However, when dealing with large path counts, the active queue will suffer from very heavy contention as many threads compete for queue access, which hampers performance. To address this problem, we design a parallel-scan-based strategy that directly processes the paths in place to completely eliminate dequeue contention from the path storage.

While C-PathGen is built upon our prior work, PathGen, it introduces a redesigned core CPG framework with several new algorithms to address the limitations of PathGen. Specifically, unlike PathGen, which does not guarantee exact path results under concurrent execution, C-PathGen guarantees exactness in parallel path exploration while optimizing the underlying path data structures for high performance. We evaluated C-PathGen on large industrial circuit graphs as well as non-circuit graphs to demonstrate its effectiveness in STA and the broader CPG applications. We also compared C-PathGen with two state-of-the-art STA engines, Tatum [87] and OpenSTA [12], to highlight its efficiency. On industrial circuits, C-PathGen achieves 6.8×–10.3× speedup over PathGen [6] while producing exact results when dealing with large path queries. In some circuits, our experiments show that PathGen deviates from the correct result by up to 5% on average. In comparison with Tatum [87]

and OpenSTA [12], C-PathGen is 72.3×–88.2× faster than Tatum and 105.2×–370× faster than OpenSTA when generating reports for large path queries.

The remainder of this paper is organized as follows: Section 2 discusses the background of CPG. We also discuss related work in this section. Since C-PathGen builds on top of PathGen [6], we will discuss the technical details of PathGen in 3 first and then present C-PathGen in Section 4. Finally, we will present the experimental results in Section 5.

2 Background

2.1 Critical Path Generation

The circuit network is modeled as a directed acyclic graph $G = (V, E)$. V is a set of n vertices representing pins of circuit components (e.g., logic gates and flip-flops), and E is a set of m directed edges representing pin-to-pin connections. Each edge $e = (u, v) \in E$ is directed from a source vertex u to a destination vertex v and is associated with a delay. Note that a timing arc between two pins may have different delays depending on its transition type (rise or fall). C-PathGen follows OpenTimer [40]'s graph formulation, where each edge stores the delay value corresponding to the queried signal type in a flat array. A path is an ordered sequence of edges $\langle e_1, e_2, \ldots, e_i \rangle$. The path cost is defined as the sum of delays along all edges in the path. 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 cost, depending on how the graph is formulated [40]).

2.2 Implicit Path Representation

While several CPG algorithms [40, 67, 89] exist, we adopt the *implicit path representation algorithm* proposed by OpenTimer [40], which outperforms other algorithms in both time and space complexity. As shown in Fig. 1, 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, which is constructed with topological relaxation. The suffix tree acts as a basis for us to discover possible branches to generate critical paths. Fig. 1(a) illustrates an example graph and its suffix tree. Black edges denote the suffix tree, and gray edges denote the non-suffix-tree edges (edges that do not belong to the suffix tree). Numbers beside the edges denote the edge weights. Numbers on the vertices denote the shortest distance to their destination vertex (T).

A prefix tree orders the non-suffix-tree edges into a tree, where each node implicitly represents a path deviating from its parent at that edge. The tree is rooted at the shortest path in the suffix tree (Fig. 1(b)). For example, node e_{SC} represents the path $\langle e_{SC}, e_{CF}, e_{FT}, e_{DT} \rangle$, shown in black in Fig. 1(c). To retrieve path delay, each non-suffix-tree edge e is annotated with a deviation cost, $dvi[e] = dis[src[e]] + weight[e] - dis[dst[e]]$. The deviation cost is the extra distance incurred by deviating on e instead of following the shortest path (e.g., $dvi[e_{SC}] = 5$ in Fig. 1(b)). Table 1 lists the data fields of a prefix tree node [40].

Constructor	Members
Pfx(p, e, w)	p: parent node e: deviation edge w: cumulative deviation cost

Table 1. Data fields of a prefix tree node (Pfx).

Although we use "deviation cost", "path cost", and "slack" interchangeably depending on context (e.g., algorithm explanation), these terms are algorithmically equivalent when ranking critical paths.

2.3 Path Correctness, Path Exactness, and Path Error

We define correctness as the property that the generated paths exactly match a golden reference implementation in terms of path trace, path cost, and criticality ordering. Specifically, a correct result reports the same set of paths with identical path traces and costs, ordered consistently with the reference by nondecreasing path cost. If multiple paths have the same cost, they form an equal-cost tie class; paths inside the same tie class have the same criticality. When two paths have the same cost, we compare their ordered pin/edge IDs from source to sink to break ties.

We define exactness as the guarantee that correctness is preserved under concurrent execution. An algorithm is exact if it produces the same top-k paths, with the same costs and cost-based ordering, as the golden reference regardless of parallel execution order or scheduling nondeterminism.

A path error occurs when a generated path deviates from the golden reference. Such deviations may include missing paths, incorrect ordering across different path costs, duplicate paths, invalid path traces, or discrepancies in path cost. In this work, we focus on discrepancies in path cost relative to the golden reference as the primary error metric, since path cost determines path criticality and ordering between different tie classes.

2.4 Related Work

CPU-based CPG algorithms have been widely studied in the STA community. iTimerC [67] and iitRace [89] reduce redundant path traversal through branch-and-bound search and pin coloring, respectively. OpenTimer [40] further improves CPG by efficiently representing paths with suffix and prefix trees, which accelerates path search. However, OpenTimer's CPG algorithm is single-threaded, so large path queries can still take seconds on industrial circuits and become a bottleneck when invoked repeatedly in timing-driven optimization loops. On the other hand, incremental CPG algorithms like Ink [3, 5] focus on caching and reusing path information between adjacent path queries, which is effective when design changes are localized and many previously generated paths remain valid. However, this type of path reuse is very workload-dependent. For instance, it can become inefficient when an optimization step affects a wide region of the circuit, where many cached paths must be discarded.

Parallel CPG algorithms focus on accelerating the CPG process for large path queries by leveraging multi-threading to explore critical paths simultaneously. PathGen [6], which builds on OpenTimer's implicit path representation, is a CPU-parallel CPG algorithm that schedules prefix tree nodes through multi-level priority queues. While it successfully accelerates the path search process, its queue-based scheduling algorithm can prematurely terminate before all k-critical candidates have been explored. As a result, PathGen is not exact, limiting its adoption by STA applications. Tatum [87] is another CPU-parallel STA engine. It supports setup and hold analysis, clock skew, multiple clocks, and timing exceptions. It operates on an abstract timing graph and accelerates block-based STA through shared graph traversals, cache-optimized data structures, and multicore execution. Its main focus is parallel timing propagation and STA infrastructure, whereas C-PathGen focuses on parallel top-k CPG, which uses the propagated timing information to enumerate and construct the most critical path traces. Its CPG algorithm is based on slack ranking at endpoints and traceback through timing tags. It collects slack tags at output nodes, ranks the corresponding endpoint/tag pairs by slack, and reconstructs selected paths by tracing backward through the timing graph. For large path queries, the runtime can increase steeply because many endpoint/tag pairs must be ranked and many paths must be traced back and reconstructed. OpenSTA [12] is a widely used STA engine under the OpenROAD project [63]. The key strength of OpenSTA lies in its rich support for industry formats, such as Verilog, Liberty, SDC, SDF, SPEF, and a Tcl-based analysis environment. However, its CPG algorithm is built on the classical A* search, with parallelism limited to level-by-level propagation. As reported in OpenTimer [40], this design incurs substantial synchronization overhead.

On an orthogonal front, GPU-parallel approaches exploit massive data parallelism from the CPG process and delegate the computation to GPU. For example, G-PBA [24] is the first work to accelerate path-based timing

analysis (PBA) on GPU and shows over 40× speedup over CPU-parallel baselines. However, its CPG algorithm is not exact because it has the same problem of premature pruning as PathGen. Lately, G-PathGen [4] overcomes this limitation by synchronizing GPU path expansion around a cost threshold, ensuring that all paths that may belong to the top-k set are generated before the search advances into less critical regions. The performance gains of these GPU-parallel approaches, however, rely on amortizing CPU–GPU preprocessing and data transfer overheads. For one-shot analyses or medium-sized path queries, these overheads can overshadow the benefits of GPU acceleration. Moreover, according to our industry partners, production STA engines are predominantly implemented for CPUs. Although migrating parts of the codebase to GPUs is an important long-term direction, an efficient CPU-parallel solution remains essential to coexist with existing CPU-based infrastructures and support workloads where GPU acceleration is impractical or unavailable.

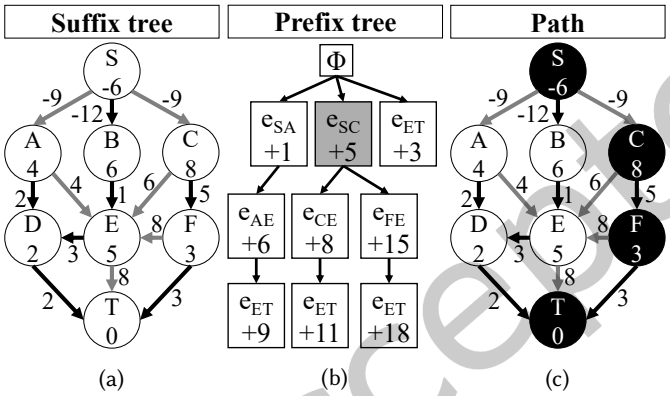


Fig. 1. Implicit path representation using suffix tree and prefix tree. Prefix $\langle e_{SC} \rangle$ + Suffix $\langle e_{CF}, e_{FT} \rangle$= Path $\langle e_{SC}, e_{CF}, e_{FT} \rangle$.

3 PathGen Preliminaries and Limitations

C-PathGen builds on the path representation used by PathGen [6], but it changes how the generated prefix-tree nodes are scheduled and when the search is allowed to terminate. To help the readers understand the motivation of C-PathGen, this section summarizes the necessary PathGen mechanisms: multi-level priority grouping, premature top-k termination, and atomic queue contention.

3.1 Multi-Level Priority Grouping

PathGen parallelizes prefix-tree expansion by replacing the single global priority queue used by sequential critical path generation with a multi-level queue (MLQ) scheduler. Each queue owns a range of deviation costs, and lower-cost ranges have higher priority. A worker repeatedly selects the lowest-level non-empty queue, dequeues one prefix-tree node, expands it to generate child nodes, and enqueues each child into the queue corresponding to its cumulative deviation cost. Fig. 2(a) illustrates this cost-range grouping. The MLQ scheduler is useful because nodes with nearby costs can be expanded concurrently, but it is an approximation of the total path ordering. Within a queue, PathGen does not process nodes in strict increasing-cost order.

3.2 Premature Top-k Termination

PathGen terminates once it has recovered k paths. This termination rule is efficient, but it causes insufficient exploration. It does not guarantee that all paths with costs lower than the latest discovered path have already been

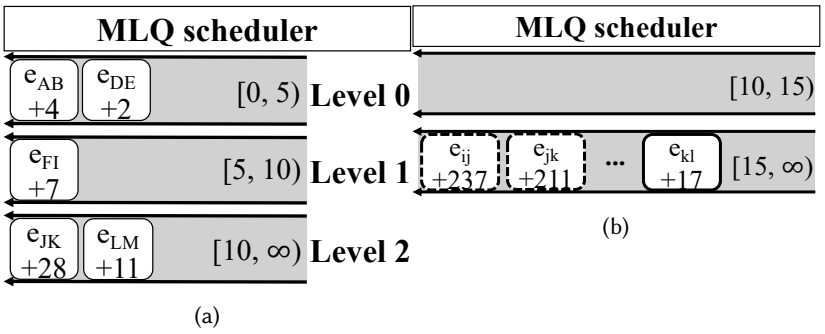


Fig. 2. PathGen concepts. (a) PathGen groups prefix-tree nodes into concurrent queues by deviation-cost range. (b) This grouping exposes parallel work, but it also treats all nodes in a queue as having the same priority. When many nodes fall into one active queue, the mix of low and high priority often makes PathGen miss k-critical paths.

explored. Because prefix-tree nodes are grouped by ranges and processed concurrently, a lower-cost node can remain waiting while other threads process higher-cost paths. Fig. 2(b) shows an example: a node with a smaller deviation cost (+17) may still be present in a lower-priority queue when PathGen has already accumulated k reported paths. This is the source of PathGen's inexactness under parallel execution. C-PathGen addresses this issue by delaying termination until the relevant cost window has been fully processed. C-PathGen's termination criteria are also supported with a formalized proof (Section 4.2).

3.3 Atomic Queue Contention

PathGen's MLQ scheduler also relies on concurrent queue operations. Every expansion requires an atomic dequeue from the active queue and some enqueues into destination queues. When many critical paths fall into the same cost range, many workers repeatedly access the same queue, and these thread-safe operations require atomic synchronization. This contention becomes increasingly expensive as the path count and thread count grow. C-PathGen addresses atomic dequeue contention by replacing repeated atomic dequeues with in-place parallel scans over generated paths stored in concurrent vectors.

Remaining details, including the full algorithm, queue-partitioning heuristics, and node-redistribution protocol, are described in the original PathGen paper [6].

4 C-PathGen

As summarized in Section 3, although PathGen exposes parallelism through MLQ scheduling, it has two major limitations: (1) On the accuracy side, it is not exact because parallel queue processing can terminate before all k-critical candidates are explored. (2) On the performance side, it scales poorly due to heavy contention from concurrent queue accesses. To overcome these limitations, C-PathGen introduces (1) a theorem that ensures the generation of exact k-critical paths and (2) a parallel-scan-based prefix tree expansion to remove dequeue contention, discussed in Sections 4.1 and 4.2. Note that C-PathGen is designed as a CPG algorithm that can be used or integrated into an existing timer such as OpenTimer, rather than a standalone STA engine. As a result, we will focus on explaining the algorithm design of C-PathGen.

4.1 Parallel-scan-based Prefix Tree Expansion

Unlike PathGen, which relies on multiple concurrent queues for its multi-level scheduler, C-PathGen builds on the idea of the multi-level scheduling strategy, but adopts concurrent vectors [55] instead. When dequeueing a

generated path for further exploration, rather than letting the threads concurrently compete for queue access, it is more efficient to keep the paths in place and scan them in parallel. A simple concurrent vector suffices for both concurrent insertion and parallel scanning of generated paths, avoiding costly dequeue contention.

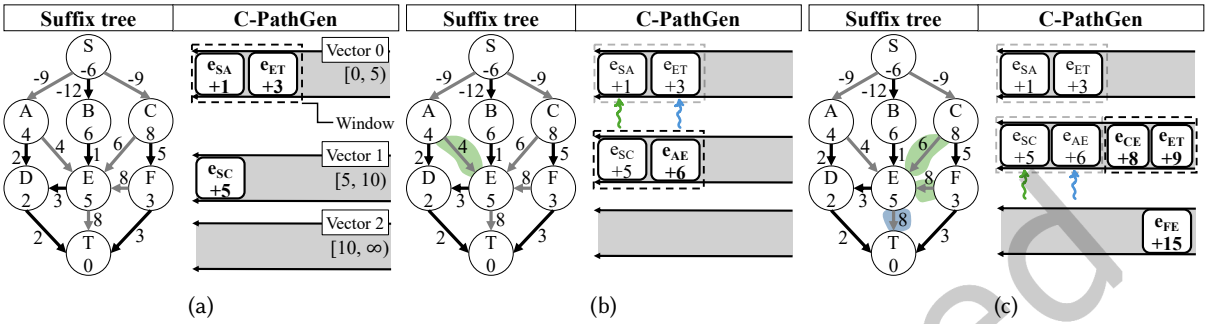


Fig. 3. Illustration of C-PathGen's parallel-scan-based prefix tree expansion. (a) Initial expansion of the prefix tree root yields Pfx(e_{SA}, 1), Pfx(e_{ET}, 3), and Pfx(e_{SC}, 5), placed into the vectors by deviation cost. (b) In vector 0, Pfx(e_{SA}, 1) and Pfx(e_{ET}, 3) form a window of paths expanded concurrently by two threads; the expansion yields Pfx(e_{AE}, 6). (c) In vector 1, Pfx(e_{SC}, 5) and Pfx(e_{AE}, 6) form the next window, and the expansion yields Pfx(e_{CE}, 8), Pfx(e_{FE}, 15), and Pfx(e_{ET}, 9). Pfx(e_{CE}, 8) and Pfx(e_{ET}, 9) form the next window.

Fig. 3 illustrates C-PathGen's parallel-scan-based strategy using a small prefix-tree expansion example. Fig 3(a) shows a suffix tree on the left and three concurrent vectors on the right. Nodes outlined in bold are new nodes in a vector. After expanding the prefix tree root, we generate Pfx(e_{SA}, 1), Pfx(e_{ET}, 3), and Pfx(e_{SC}, 5), which are placed into vectors based on their deviation costs. In vector 0, Pfx(e_{SA}, 1) and Pfx(e_{ET}, 3) form a *window* of paths that are expanded concurrently by two threads: the green one to expand Pfx(e_{SA}, 1) and the blue one to expand Pfx(e_{ET}, 3), as shown in (b). Expanding Pfx(e_{SA}, 1) yields Pfx(e_{AE}, 6); expanding Pfx(e_{ET}, 3) yields nothing because e_{ET} is already at the destination. The processed window is then grayed out. In (b), vector 1 contains Pfx(e_{SC}, 5) and the newly generated Pfx(e_{AE}, 6), which together form another window. In (c), expanding the window of paths in vector 1 yields Pfx(e_{CE}, 8), Pfx(e_{FE}, 15), and Pfx(e_{ET}, 9). Pfx(e_{CE}, 8) and Pfx(e_{ET}, 9) then form the next window.

In terms of path priority update, similar to PathGen's node redistribution strategy, once the higher-priority vectors (vectors 0 and 1) have no new paths, C-PathGen updates the ranges of each vector and moves all paths from the lowest-priority vector (vector 2) to higher-priority vectors to update their priorities. Unlike PathGen, which updates ranges while generating new paths, C-PathGen separates these steps; it first updates ranges, then generates paths to ensure all paths are placed correctly before further processing. In C-PathGen, we refer to one priority update as one *step*, and define the step size as the width of each range (e.g., vector 0 covers [0, 5) and vector 1 covers [5, 10), so the step size is 5). The step size determines the parallelism at each step and work efficiency, which impacts performance. We will discuss this in our experiments.

Algorithm 1 describes C-PathGen's parallel-scan-based prefix tree expansion. We initialize the path counter *num_paths*, termination flag *finished*, and solution set Ψ (line 1:3). An array of `size_t` pairs stores the start and end points of the windows in each (line 4). While path generation is not finished (line 5:26), we iterate through all the vectors (line 6). The window start point is zero, and the end point is the size of the current vector (line 7:10). While the window is not empty (line 11:17), all paths inside are expanded concurrently. After expansion, we synchronize the threads, update *num_paths*, adjust the window if new paths appear (line 14:17). We clear the vector if paths inside it are fully expanded (line 18:20). Now, if more than k paths are generated (line 21:23), we stop and sort the results (line 27:28). This termination condition (*finished*) is evaluated only after a window

Algorithm 1: C-PathGen(k, CV, d, Δ)

Input: path count k, array of N concurrent vectors CV, destination vertex d, step size Δ
Output: critical path solution set Ψ

```
1  num_paths ← 0;
2  finished ← false;
3  Ψ ← ϕ ;                                                    // initialize solution set
4  windows ← array of std::pair<size_t, size_t> ;   // record the window start and end points for each
     vector (atomics)
5  while !finished
6      for i ← 0 to N − 1
           /* initialize window                                                          */
7          windows[i].first ← 0;
8          windows[i].second ← CV[i].size();
           /* get current window                                                         */
9          wbeg ← windows[i].first;
10         wend ← windows[i].second;
11         while wbeg < wend
12             Parallel Foreach node ∈ CV[i][wbeg:wend]
13                 SpurCV(node, d, CV) ;          // same as SpurMLQ, except using concurrent vectors
14             synchronize all threads ;                  // implicit barrier after Parallel Foreach
15             num_paths += (wend−wbeg);
               /* update window                                                          */
16             wbeg += (wend−wbeg);
17             wend ← CV[i].size();
           /* no more new windows in this vector, clear and reset                        */
18         CV[i].clear();
19         windows[i].first ← 0;
20         windows[i].second ← 0;
21         if num_paths ≥ k then
22             finished ← true;
23             break;
24     if !finished then
25         Update cost ranges (with step size Δ) and promote nodes from the lowest-priority vector to others;
26         Downsize the lowest-priority vector and update windows;
27 Move all the paths from CV to Ψ;
28 Sort Ψ in ascending order of deviation cost;
29 return the first k paths of Ψ;
```


has been fully processed. The **Parallel Foreach** construct follows fork–join semantics and serves as an implicit synchronization barrier; thus, when $num\_paths \geq k$ is evaluated, all concurrent expansions have completed, and we are not forming new windows in the current vector.

If the generation procedure is not terminated, we update the cost ranges to promote paths from the lowest-priority vector (line 25). For example, if $\Delta = 5$ and the ranges are $[0, 5)$, $[5, 10)$, and $[10, \infty)$, since all paths

with costs within [0, 10) are found, prefix tree expansion will only generate paths with costs no smaller than 10. Therefore, we update the ranges to [10, 15), [15, 20), and [20, ∞). This update is done in parallel. After the range update, we resize the lowest-priority vector and update *windows* (line 25). After generating enough paths, we merge all paths from CV into Ψ, sort them (line 27:28), and return only the first k paths (line 29). Note that if the user requires the path traces, it is done after Algorithm 1 completes, since we do not want the path trace recovery process to interrupt path generation.

4.2 Exact k-critical Path Generation

PathGen is not exact because it stops after finding k paths, but these may not be the true k-critical ones since exploration is out of order. For example, in Fig. 3(c), if k is set to 5 and Pfx(e_{ET}, 9) is found before Pfx(e_{CE}, 8), PathGen would stop after finding Pfx(e_{ET}, 9) and misses Pfx(e_{CE}, 8), leading to inexact results.

To address this issue, after generating k paths, C-PathGen stops the path generation process *only when the current vector has no new windows*. This means all paths within that vector's cost range have been found, ensuring no paths are missed. For example, in Fig. 3(c), if k is set to 5, C-PathGen stops only after the window that contains Pfx(e_{CE}, 8) and Pfx(e_{ET}, 9) is processed and no more new windows are formed, which guarantees the 5-critical set is exact.

To formalize the exactness guarantee, we introduce the following notations. Let Δ denote the fixed step size used to partition the path cost space into contiguous cost ranges. Let α denote the upper bound of the highest-priority cost range at termination. At termination, the active cost range is $[\alpha - \Delta, \alpha)$, and no cost range with upper bound smaller than α remains unprocessed.

We highlight two invariants (Lemma 1 and 2) that guarantee the exactness of C-PathGen:

Lemma 1 (Promotion Completeness). *Any prefix tree node corresponding to a path with cost $c \leq \alpha$ that resides in a lower-priority vector will be promoted to a higher-priority vector whenever cost ranges are updated.*

Lemma 2 (Window Completeness). *For any active vector, C-PathGen expands all paths in its current window and does not terminate until the window is fully processed and no new paths can be inserted into that vector.*

C-PathGen maintains these invariants to properly determine the priority of the generated paths and not miss any k-critical paths. Using Lemma 1 and 2, Theorem 1 proves the exactness of C-PathGen.

Theorem 1 (Exactness). *C-PathGen generates exact top-k critical paths.*

Proof. Assume for contradiction that C-PathGen terminates with an active cost range $[\alpha - \Delta, \alpha)$ and fails to generate a k-critical path p with cost $c \leq \alpha$.

If p resides in a higher-priority vector, then by Lemma 2 and the fork–join synchronization semantics of Algorithm 1, all paths in that vector must have been fully expanded before termination. Hence, p would have been generated and included in the final ranking, contradicting the assumption.

If p resides in the lowest-priority vector, then since $c \leq \alpha$, Lemma 1 guarantees that p must have been promoted to a higher-priority vector during cost range updates, after which it would have been expanded prior to termination. This again contradicts the assumption.

Therefore, no k-critical path with cost $c \leq \alpha$ can be missed, and C-PathGen generates exact top-k critical paths. □

5 Experimental Results

We implemented C-PathGen using C++ and compiled it with GCC-11.4 on a 4.8-GHz 64-bit Linux machine of an Intel Core i5-13500 Processor. This machine has a maximum of 20 threads. We enable the optimization flag `-O3` and C++17 standard `-std=c++17`. We evaluate C-PathGen on five large industrial circuit graphs (des_perf,

vga_lcd, netcard, leon3mp, and leon2) generated by an open-source STA tool, OpenTimer [40]. To further evaluate the scalability of C-PathGen on even-larger circuit graphs, we follow the methodology used in the TAU Timing Analysis Contest [34] to synthetically construct larger circuit instances by replicating an input circuit graph 4× and 8× and randomly inserting edges across the replication boundaries. Additionally, we evaluated the performance of C-PathGen beyond circuit graphs by selecting four large non-circuit graphs (ldoor, M6, nlpkkt120, and cage15) from DIMACS Graph Challenge [1], allowing us to study the generalizability of C-PathGen beyond STA workloads. For undirected graphs, we induced a direction from smaller to larger vertex IDs to avoid cycles. By default, we set a step size of 2.5 and use 20 oneTBB concurrent vectors [55]. We validate the path costs of C-PathGen by comparing them with the exact costs generated by OpenTimer.

Regarding the accuracy evaluation of PathGen and C-PathGen, we compare the generated top-k paths against the golden reference produced by OpenTimer's sequential CPG algorithm. OpenTimer's CPG implementation has been validated through TAU Timing Analysis Contests [34, 35], where its results were verified against the golden reference generated by IBM's Einstimer, making it a reliable baseline for accuracy evaluation. Also, OpenTimer's path report has shown a strong correlation with both OpenSTA and industrial timers [40]. Since path criticality is fully determined by path cost, we evaluate correctness by measuring discrepancies in path cost relative to the golden reference. Therefore, we define the average path cost error (Err_{avg}) as the average relative difference across all reported paths, and the maximum path cost error (Err_{max}) as the maximum relative difference among all reported paths.

Beyond validating path-cost errors, we also validate path identity. Specifically, for paths with different costs, C-PathGen must match OpenTimer's ordering. For paths with the same cost, any order among them is valid because they are equally critical, so we treat them as an equal-cost group. For each equal-cost group, we compare the recovered path traces from C-PathGen and OpenTimer, where a path trace is represented by its ordered pin/edge IDs from source to sink. By comparing path traces within each equal-cost group, we can detect missing paths, duplicate paths, path-trace mismatches, and cost mismatches: a missing, duplicated, or incorrect path changes the trace set in the corresponding cost group, while a cost mismatch places the path into the wrong group. Therefore, the 0% error reported for C-PathGen means that its reported paths match OpenTimer in both cost and path trace.

5.1 Baselines

We consider OpenTimer [40], PathGen [6], Tatum [87], and OpenSTA [12] as our baselines, as they are state-of-the-art open-source STA engines with CPG support. OpenTimer is a single-threaded critical path generator that generates exact k-critical paths while outperforming existing single-threaded methods in both time and space complexities. PathGen is a CPU-parallel critical path generator that parallelizes OpenTimer's sequential algorithm using a multi-level scheduling strategy. As highlighted earlier, the main limitation of PathGen is that it is not exact since its path priority estimation often misses some k-critical paths. For PathGen, by default, we use 80 queues and enable geometric slack partitioning for the best performance. We also enable node redistribution for the highest accuracy. For both Tatum and OpenSTA, we directly invoke their timing analysis and path reporting APIs. For Tatum, we invoke `update_timing()` and `collect_worst_setup_timing_paths(...)`. For OpenSTA, we invoke the Tcl commands `update_timing` and `report_checks`. We use these APIs to measure the runtime of graph setup, timing propagation, and path reporting.

5.2 Overall Performance Comparison

Table 2 compares the average/maximum path cost error ($\text{Err}_{avg/max}$), overall performance, and peak memory usage (Mem) between OpenTimer [40], PathGen [6], and C-PathGen. The peak memory usage was measured using isolated per-algorithm processes and the Maximum Resident Set Size field reported by `/usr/bin/time -v`.

For OpenTimer, we measure the runtime of suffix tree construction (Sfxt) and prefix tree expansion (Pfxt). For PathGen and C-PathGen, we only measure their prefix tree expansion runtime, as they only parallelize prefix tree expansion; their suffix tree construction algorithm is the same as OpenTimer. Table 2 also lists three key graph statistics: number of vertices ($|V|$), number of edges ($|E|$), and diameter ($|D|$). The diameter is the longest distance between any two vertices. Since STA applications (e.g., sign-off) often need to analyze millions of paths, we set the path count k to 1M. We use the maximum thread count of 20 for both PathGen and C-PathGen.

In terms of performance, C-PathGen is faster than PathGen on all graphs. For example, C-PathGen's speedup over OpenTimer is 13× on vga_lcd and 16.5× on M6, while PathGen is 1.9× and 1.6×. This is because PathGen's concurrent enqueue and dequeue require costly atomic operations, while C-PathGen completely avoids atomic dequeue operations by processing the paths in place using parallel scans. In terms of accuracy, PathGen incurs large errors on certain graphs. For example, PathGen's Err_{max} reaches 199% on des_perf and 48.6% on cage15. This is because PathGen's path priority estimation can miss many k-critical paths, leading to high errors. In contrast, C-PathGen is always exact with 0% Err_{avg} and Err_{max}. This is because C-PathGen computes a cost threshold that filters out non-k-critical paths.

In terms of memory usage, in all three algorithms, the dominant memory component is the graph and suffix tree. The remaining difference comes from the allocation behavior of each scheduler. OpenTimer maintains a priority queue during prefix-tree expansion, whose peak size can be large on dense graphs. PathGen stores paths in concurrent queues, while C-PathGen stores active windows and generated paths in segmented concurrent vectors. Although C-PathGen uses fewer scheduling containers than PathGen, its vector-based storage may retain allocated segments and temporary buffers during window expansion. Overall, C-PathGen remains within the same memory range as OpenTimer and PathGen while significantly reducing prefix-tree expansion time.

Table 2. Overall performance and accuracy comparison between OpenTimer [40], PathGen [6], and C-PathGen. All data is an average of ten runs.

				OpenTimer			PathGen (20 threads)				C-PathGen (20 threads)			
Graph	$\|V\|$	$\|E\|$	$\|D\|$	Sfxt (ms)	Pfxt (ms)	Mem (GB)	Err_{avg} (%)	Err_{max} (%)	Pfxt (ms)	Mem (GB)	Err_{avg} (%)	Err_{max} (%)	Pfxt (ms)	Mem (GB)
des_perf	303K	387K	132	80.5	665.0	1.034	5.0	199.0	549.8 (1.2×)	1.557	0.0	0.0	96.8 (6.9×)	0.767
vga_lcd	397K	498K	130	106.2	1095.8	0.645	0.0	0.6	584.8 (1.9×)	0.892	0.0	0.0	84.4 (13.0×)	0.985
netcard	3.9M	4.9M	284	1432.4	2160.7	3.366	0.1	1.5	542.8 (4.0×)	3.513	0.0	0.0	181.7 (11.9×)	3.291
leon3mp	3.3M	4.1M	304	1292.2	2227.0	2.940	0.1	0.6	727.8 (3.1×)	3.114	0.0	0.0	186.1 (12.0×)	2.871
leon2	4.3M	5.2M	308	1704.9	2600.5	3.680	1.5	25.3	1336.1 (1.9×)	3.829	0.0	0.0	238.7 (10.9×)	3.595
ldoor	952K	22.7M	6734	1227.2	8024.0	16.005	0.0	0.0	49074.1 (0.2×)	13.428	0.0	0.0	1545.5 (5.2×)	13.778
M6	3.5M	10.5M	2221	1630.7	2639.3	5.830	0.0	0.0	1690.9 (1.6×)	5.454	0.0	0.0	159.5 (16.5×)	5.392
nlpkkt120	3.5M	46.6M	2	3249.9	7931.9	24.339	0.0	0.0	11215.0 (0.7×)	21.513	0.0	0.0	887.8 (8.9×)	23.238
cage15	5.1M	47.0M	147	9874.8	13895.1	30.693	3.4	48.6	14126.2 (1.0×)	26.879	0.0	0.0	2250.2 (6.2×)	27.756

$|D|$: graph diameter. $\text{Err}_{avg/max}$: average/max path cost error. Sfxt/Pfxt: suffix tree construction/prefix tree expansion runtime.

To evaluate OpenTimer, PathGen, and C-PathGen on even larger circuits, Table 3 presents the same comparison on replicated leon2, leon3mp, and netcard graphs. The suffixes _x4 and _x8 indicate that the corresponding circuit graphs are replicated by 4× and 8×, respectively. Compared to Table 2, the replicated circuits show larger PathGen errors because they contain many more paths with similar costs. These paths are placed into the same queues, so PathGen has many more nearly equal candidates to choose from. When PathGen stops after finding k paths, some other paths with the same or lower cost may still be waiting in the queues. Therefore, PathGen is more likely to miss k-critical paths on the replicated circuits. In terms of performance, we did not see longer runtimes on replicated circuits. This is because the path generation stage is governed by the number of candidate

paths that must be explored to identify the top-k paths, rather than by graph size alone. Therefore, a larger circuit does not necessarily make the top-k path search deeper or harder in proportion to graph size.

Table 3. Performance and accuracy comparison on replicated circuits.

Graph	$\|V\|$	$\|E\|$	$\|D\|$	OpenTimer			PathGen (20 threads)				C-PathGen (20 threads)			
				Sfxt (ms)	Pfxt (ms)	Mem (GB)	Err_{avg} (%)	Err_{max} (%)	Pfxt (ms)	Mem (GB)	Err_{avg} (%)	Err_{max} (%)	Pfxt (ms)	Mem (GB)
leon2_x4	17.3M	20.5M	1235	6769.7	129.4	14.455	1362.32	2609.48	577.0 (0.2×)	14.545	0.0	0.0	28.2 (4.6×)	14.430
leon2_x8	34.6M	41.0M	2471	14383.2	130.2	28.696	1416.30	2794.83	571.5 (0.2×)	28.738	0.0	0.0	29.5 (4.4×)	28.687
leon3mp_x4	13.5M	16.2M	1219	5352.7	120.8	11.315	1994.42	4369.45	660.1 (0.2×)	11.384	0.0	0.0	29.3 (4.1×)	11.324
leon3mp_x8	27.0M	32.3M	2439	9594.9	123.9	22.419	1270.20	2210.17	621.4 (0.2×)	22.415	0.0	0.0	28.8 (4.3×)	22.415
netcard_x4	16.0M	19.2M	1139	5830.5	124.3	13.188	2862.23	4534.44	611.8 (0.2×)	13.270	0.0	0.0	47.0 (2.6×)	13.293
netcard_x8	32.0M	38.4M	2279	11800.6	122.1	26.144	2610.24	5008.15	635.2 (0.2×)	26.220	0.0	0.0	45.3 (2.7×)	26.260

$|D|$: graph diameter. $\text{Err}_{avg/max}$: average/max path cost error. Sfxt/Pfxt: suffix tree construction/prefix tree expansion runtime.

5.3 Performance and Accuracy at Different Thread Counts

We first explain the accuracy degradation of PathGen observed under high thread counts. PathGen adopts a multi-level queue (MLQ)–based parallelization strategy for prefix tree expansion, where multiple paths are processed concurrently. Under high concurrency, nondeterministic scheduling can cause higher-priority paths to be processed later than lower-priority ones, resulting in misordered path results; this ultimately leads to path cost discrepancies relative to the golden reference shown in Fig. 4.

Fig. 4 plots (1) the speedups of C-PathGen and PathGen over OpenTimer and (2) the path errors of PathGen at different thread counts on two large graphs (leon2 and ldoor). We omit C-PathGen's error plot since it is exact. In terms of performance, increasing the thread count increases the speedup of PathGen. For example, on leon2, the speedup increases from 2.7× at eight threads to 5.4× at 20 threads. This is because more threads can clear the queues faster. However, higher thread counts increase contention, so using the maximum thread count does not always yield the optimal speedup. For example, on ldoor, the speedup drops from 3.1× at 16 threads to 1.6× at 20 threads. For C-PathGen, the speedup always increases when we increase the thread count. For example, the speedup increases from 4.8× at four threads to 11× at 20 threads. Regardless of the thread count, C-PathGen is always faster than PathGen. This is because PathGen's concurrent dequeues require costly atomic operations, while C-PathGen entirely avoids atomic dequeues by using parallel scans to process paths in place.

In terms of accuracy, increasing the thread count enlarges PathGen's difference between Err_{avg} and Err_{max}. For example, on leon2, the error difference increases from 9% at eight threads to 36% at 16 threads. This is because with more threads, path exploration becomes more random, which increases the chance of missing k-critical paths, resulting in larger error gaps. On ldoor, the error difference reaches 100% at 16 threads and drops again at 20 threads. This occurs because at 16 threads, PathGen misses many k-critical paths, leading to large error gaps. In contrast, C-PathGen is always exact with no errors. This is because C-PathGen computes a cost threshold that filters out non-k-critical paths.

To further evaluate C-PathGen's performance beyond 20 threads, we run it on another server machine with two Intel Xeon Gold 6138 CPUs, totaling 40 physical cores and 80 hardware threads. We measure the speedup of C-PathGen over OpenTimer at 1–80 threads. We select the three largest circuit benchmarks, leon2, leon3mp, and netcard. Note that this experiment is different from Fig. 4 because the CPU model, clock frequency, NUMA topology, and thread placement differ from the CPU used in the original experiments. Therefore, we only use it as an additional scalability test. As shown in Fig. 5, C-PathGen's performance continues to improve slightly beyond 20 threads on some circuits, but the overall gains start saturating at about 20-32 threads. For example, netcard

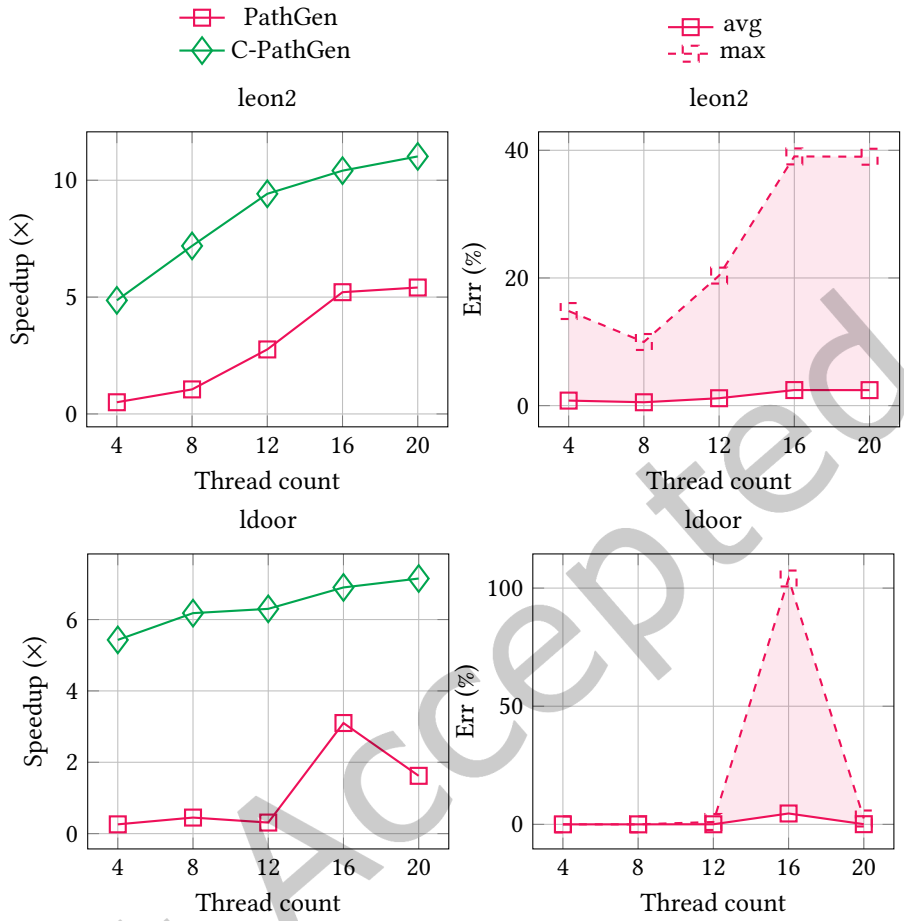


Fig. 4. (1) Speedups of C-PathGen and PathGen over OpenTimer and (2) the path errors of PathGen at different thread counts on leon2 and ldoor.

improves from 2.78× at 20 threads to 2.91× at 64 threads, while leon3mp improves from 2.53× at 20 threads to 2.67× at 40 threads. In contrast, leon2 reaches 2.29× at 40 threads and then drops to 2.01× at 80 threads. This is because adding more threads also adds multithreading overhead, such as synchronization, contention for shared data, and memory traffic. After enough threads are used, these overheads can outweigh the benefit of additional parallelism.

5.4 Performance at Different Path Counts

Fig. 6 plots the runtime of PathGen and C-PathGen at different path counts (k) on two large graphs (leon3mp and M6). We denote PathGen with N threads as PathGen$_N$, and we use the same notation for C-PathGen. On leon3mp, their runtimes are similar for small path counts. For example, when $k = 10^3$, PathGen and C-PathGen both take about 1 ms. This is because small path counts involve very few accesses to their path data structures, resulting in a similarly short runtime. For large path counts, C-PathGen consistently outperforms PathGen. For example, at

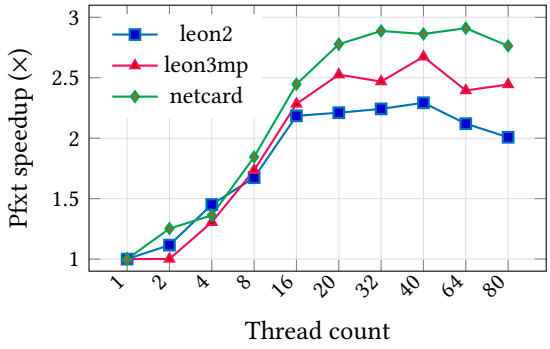


Fig. 5. Thread scalability experiment of C-PathGen on a two-socket Intel Xeon Gold 6138 server with 40 physical cores and 80 hardware threads.

k =10K, 100K, and 1M, C-PathGen$_{12}$ is 1.5 ms, 24.4 ms, and 370 ms faster, respectively. This is because PathGen's atomic dequeue operations create heavy contention, whereas C-PathGen avoids this by letting threads scan paths in place without competing for data access. On M6, at large path counts, PathGen outperforms C-PathGen at certain thread counts. For example, at k =10K, PathGen$_{12}$ is 10.6 ms faster. This is because PathGen stops exactly at k paths, while C-PathGen generates extra paths to ensure exactness. Despite the extra work, using a small step size minimizes the number of extra paths (discussed in Section 5.9).

To further compare OpenTimer, PathGen, and C-PathGen at different path counts, Fig. 8 plots the path generation runtime from $k = 1$ to $k = 1$M on leon2, leon3mp, and netcard. For very small k, OpenTimer is faster because the small queries do not provide enough work to amortize C-PathGen's multithreading overhead. For example, at $k = 10$ on leon2, OpenTimer takes 0.0035 ms, while C-PathGen takes 0.022 ms. As k increases, C-PathGen becomes faster than OpenTimer at $k = 100$K on all three circuits. For example, at $k = 100$K, C-PathGen takes 7.43 ms on leon2, 7.28 ms on leon3mp, and 6.42 ms on netcard, while OpenTimer takes 10.43 ms, 9.80 ms, and 9.99 ms, respectively. This is because these queries expose enough path expansion work for C-PathGen to run concurrently, while OpenTimer remains single-threaded. C-PathGen also outperforms PathGen for most medium and large path counts. For example, at $k = 1$M, C-PathGen takes 60.05 ms on leon2, 58.23 ms on leon3mp, and 72.76 ms on netcard, while PathGen takes 582.37 ms, 602.79 ms, and 582.87 ms, respectively. This corresponds to 8.01×–10.4× speedup over PathGen at $k = 1$M. This is because larger path counts expose more path generation work, where C-PathGen's parallel in-place scanning largely avoids atomic queue contention.

5.5 Accuracy at Different Path Counts

Fig. 7 plots the path errors of PathGen at different path counts (k) on netcard. For all thread counts, Err$_{max}$ peaks at $k = 100$. For example, Err$_{max}$ is 12% at 12 threads and 113% at 16 threads. This is likely because the region near the 100^{th} path contains many similar-cost paths, making correct ordering more difficult for PathGen. This example also shows that increasing the thread count tends to worsen both Err$_{avg}$ and Err$_{max}$ at $k = 100$. This happens because the region near the 100^{th} path contains many similar-cost paths, and more threads lead to greater interleaving in exploration, increasing the chance of misordering paths.

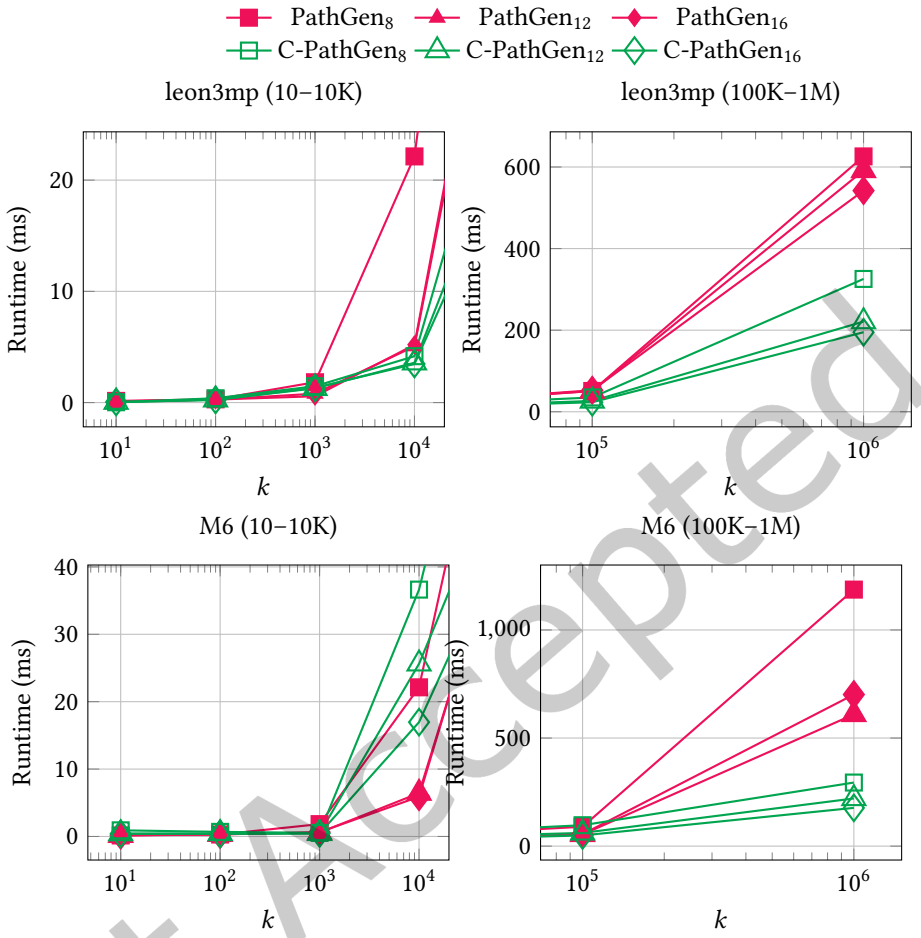


Fig. 6. Runtime of PathGen and C-PathGen at different path counts with 8, 12, and 16 threads on leon3mp and M6.

5.6 Effectiveness of Node Redistribution

When too many paths enter the lowest-priority queue, PathGen struggles to distinguish path priorities, leading to high error. In this case, PathGen's node redistribution (NR) strategy moves some paths from the lowest-priority queue to higher-priority queues to restore the proper ordering of these paths and reduce errors.

Fig. 9 plots the path errors of PathGen at different thread counts with and without NR on two large graphs (vga_lcd and ldoor). "PathGen^{-NR}" refers to PathGen without NR; "PathGen^{NR}" refers to PathGen with NR. We omit C-PathGen's error plot as it is exact. Similar to Table 2, we set k =1M.

On vga_lcd, PathGen^{NR} is more accurate than PathGen^{-NR} in terms of Err$_{max}$. For example, Err$_{max}$ is 1924.6% without NR and 371.5% with NR at eight threads. This is because NR moves paths from the lowest-priority queue to higher-priority queues, which restores proper ordering of the paths and reduces the error. On ldoor, however, NR has no effect. For example, Err$_{max}$ is around 3% both without NR and with NR at 20 threads. This is likely due to ldoor's dense structure, which produces many similar-cost paths. Too many of these paths end up in the

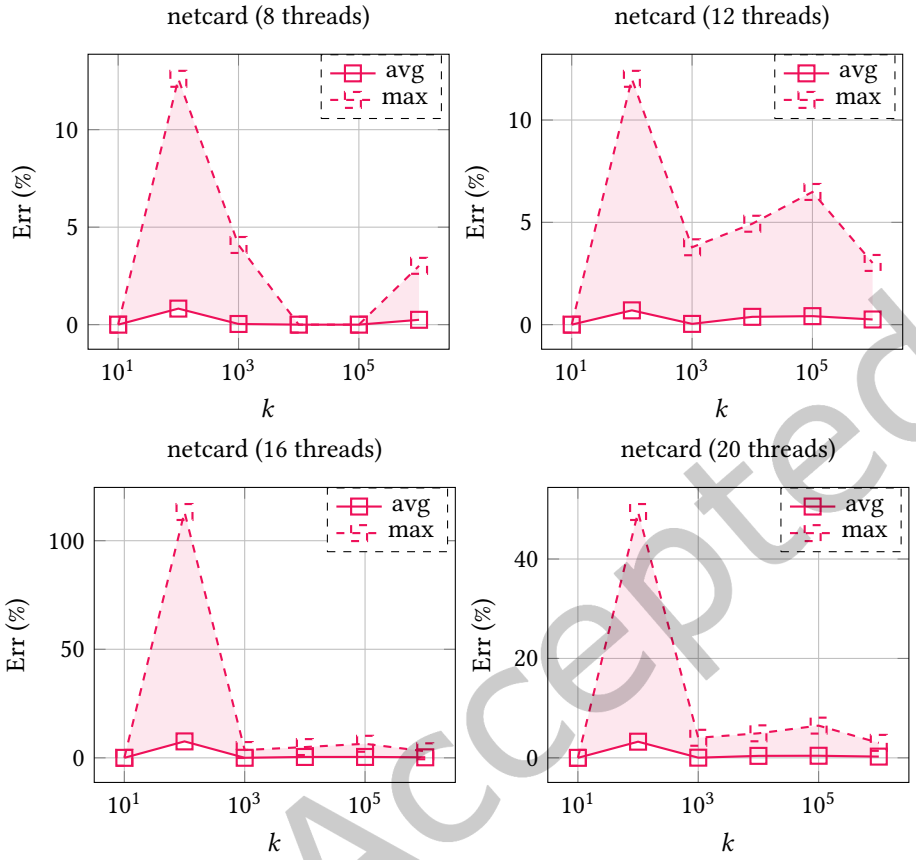


Fig. 7. Path errors of PathGen at different path counts with 8, 12, 16, and 20 threads on netcard.

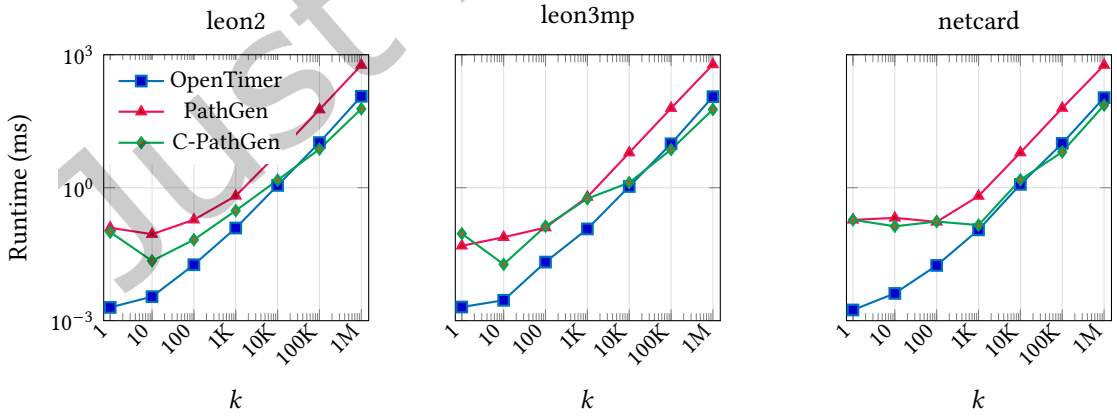


Fig. 8. Path generation runtime of OpenTimer, PathGen, and C-PathGen at different path counts on leon2, leon3mp, and netcard.

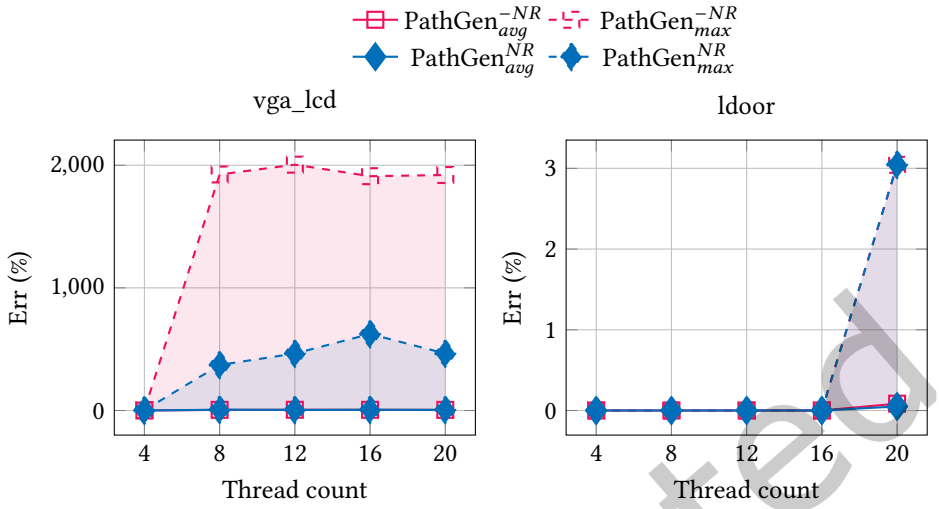


Fig. 9. Path errors of PathGen with and without node redistribution on vga_lcd and ldoor.

lowest-priority queue, and NR fails to move most of them to higher-priority queues, thus failing to reduce the error.

5.7 Performance under Different Partitioning Strategies of Slack Distribution

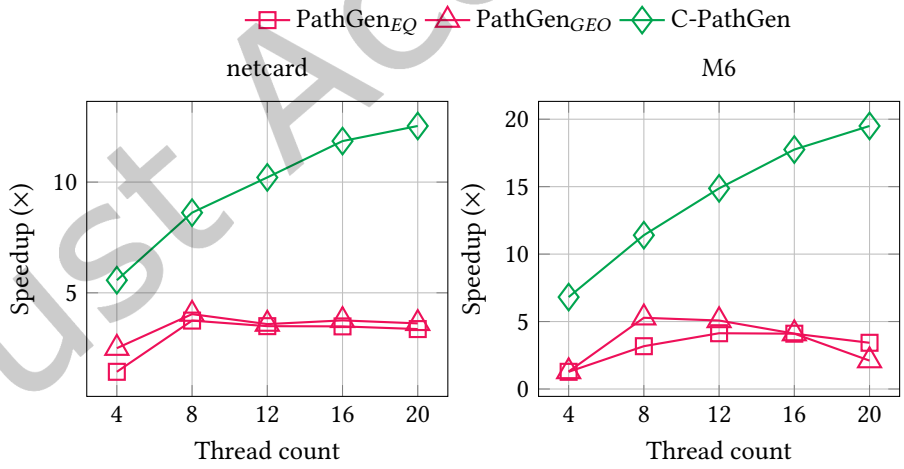


Fig. 10. Speedups of PathGen (under different partitioning strategies of slack distribution) and C-PathGen over OpenTimer on netcard and M6.

PathGen uses two strategies to partition generated paths into different priority queues. The equal distribution (EQ) strategy assigns each queue the same cost range, but since most paths in circuit graphs have small slacks, this causes higher-priority queues to contain many paths and incurs heavy contention. To address this, the geometric

distribution (GEO) strategy assigns smaller ranges to higher-priority queues and larger ranges to lower-priority queues, balancing the path count in each queue.

Fig. 10 plots PathGen's speedup over OpenTimer with different partitioning strategies of slack distributions on two large graphs (netcard and M6). We also plot C-PathGen's speedup. "PathGen$_{EQ}$" refers to PathGen with the equal partitioning strategy; "PathGen$_{GEO}$" refers to PathGen with the geometric partitioning strategy. Similar to Table 2, we set k =1M.

On netcard, PathGen$_{GEO}$ is consistently faster than PathGen$_{EQ}$. For example, at eight threads, the speedup increases from 1.4× with EQ to 2.5× with GEO, and at 20 threads, from 3.3× to 3.6×. This is because GEO reduces contention in higher-priority queues by narrowing their cost ranges, moving some paths to lower-priority queues for better balance. On M6, however, PathGen$_{GEO}$ is not faster with high thread counts. For example, the speedup is 4.1× both with EQ and GEO at 16 threads; the speedup is 3.4× with EQ and 2.1× with GEO at 20 threads. This is likely due to M6's dense structure, which produces too many path candidates so that the cost of atomic operations outweighs the benefits of reduced contention. In contrast, C-PathGen does not have severe contention issues, so its speedup keeps increasing with more threads. For example, on M6, the speedup increases from 6.8× at four threads to 19.4× at 20 threads. This is because C-PathGen completely avoids atomic dequeue operations by using parallel scans to process paths in place, which largely reduces contention.

5.8 Performance at Different Dequeue Bulk Sizes

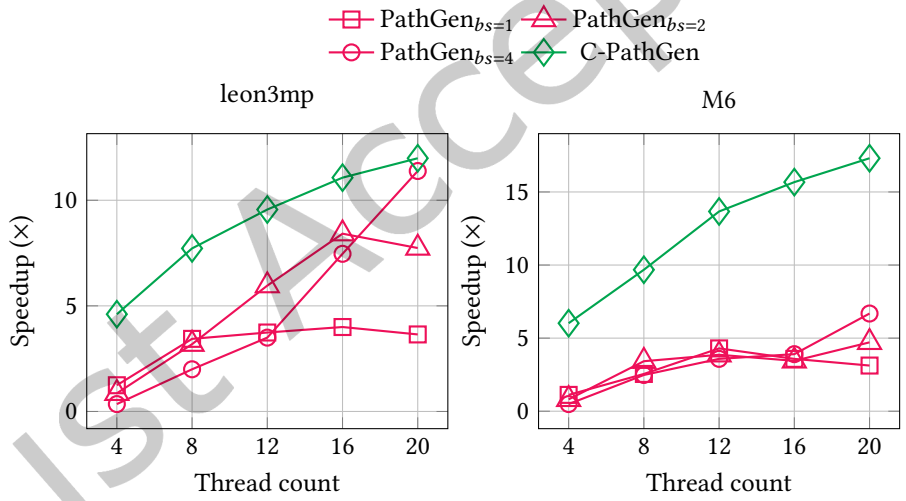


Fig. 11. Speedups of PathGen (with different bulk sizes) and C-PathGen over OpenTimer at different thread counts on leon3mp and M6.

With PathGen's concurrent-queue-based strategy, bulk dequeues improve performance by letting a thread fetch multiple paths at once, causing it to revisit the queue and contend with other threads less frequently.

Fig. 11 plots PathGen's speedup over OpenTimer with different bulk sizes on two large graphs (leon3mp and M6). We also plot C-PathGen's speedup. "PathGen$_{bs=N}$" refers to PathGen with a bulk size of N. Similar to Table 2, we set k =1M.

On leon3mp, with larger bulk sizes, PathGen's speedup continues to grow with more threads. For example, with $bs = 2$, the speedup increases from 3.1× at eight threads to 8.4× at 16 threads; with $bs = 4$, it increases

from 1.9× to 11.3× at 20 threads. This is because larger bulks assign each thread more paths to work on before returning to the queue, which reduces contention. However, at lower thread counts, larger bulk sizes do not always improve performance. For example, on leon3mp at 12 threads, the speedup is 3.4× with $bs = 4$ but 8.4× with $bs = 2$. With fewer threads, contention is low already, so a larger bulk size keeps each thread busy longer and clears the queues more slowly. On M6, larger bulk sizes improve performance only at the highest thread count. For example, at 20 threads, the speedup is 6.6× with $bs = 4$, 4.7× with $bs = 2$, and 3.1× with $bs = 1$; at 12 threads, however, the speedup is 3.5× with $bs = 4$, 3.8× with $bs = 2$, and 4.2× with $bs = 1$. This is because M6's dense structure produces many path candidates, so more threads are needed to process them quickly before larger bulk sizes can take effect. In contrast, C-PathGen's speedup consistently grows without needing bulk operations. For example, on M6, the speedup increases from 6× at four threads to 17.3× at 20 threads. While PathGen still faces dequeue contention even with bulk operations, C-PathGen entirely avoids it by using parallel scans to process paths.

5.9 Work Efficiency at Different Step Sizes

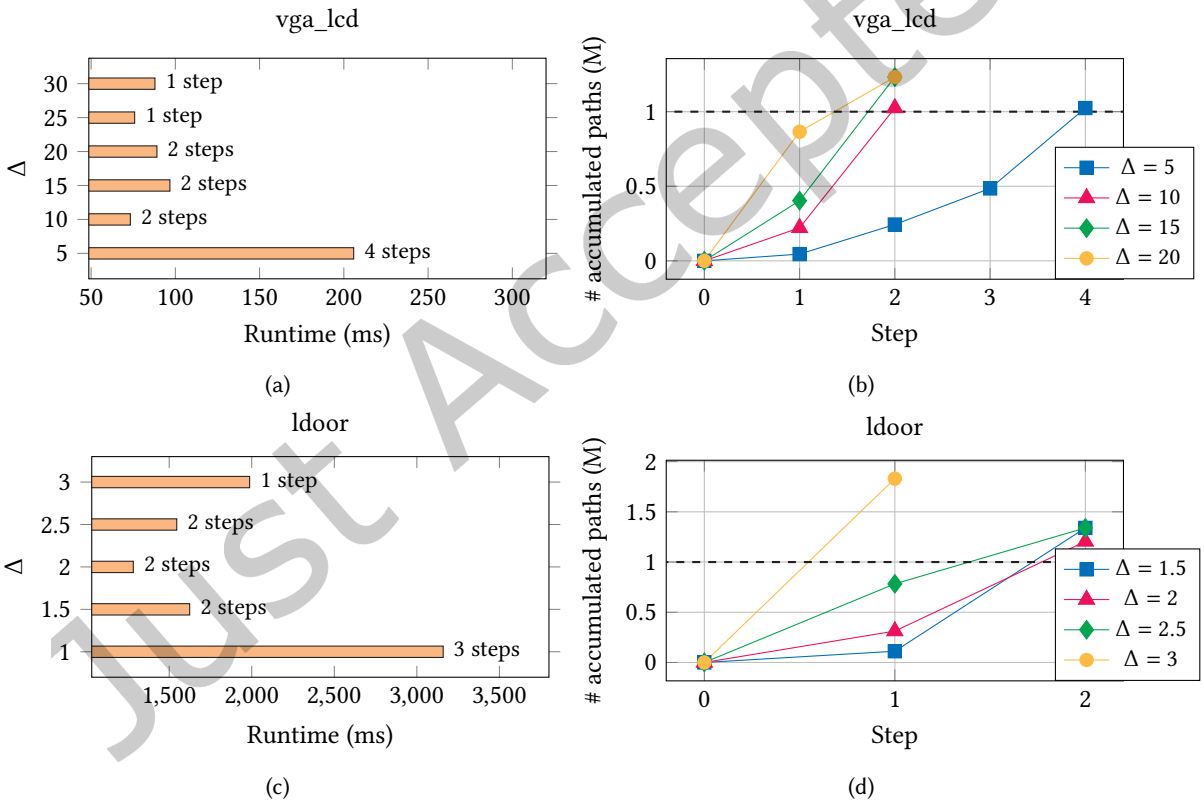


Fig. 12. Left: Pfxt runtime of C-PathGen at different step sizes (Δ) on vga_lcd and ldoor. Right: accumulated path count at each step on vga_lcd and ldoor.

In C-PathGen, the choice of step size (Δ) is important. A small Δ minimizes the number of redundant generated paths, but reduces the number of concurrently processed paths at each step. A large Δ increases the parallelism

at each step, but will likely generate more redundant paths. In this experiment, we analyze the work efficiency of C-PathGen at different Δ. We measure the work efficiency in terms of the total number of generated paths before C-PathGen terminates. Fewer generated paths indicate higher work efficiency. We only study the performance of Pfxt in this experiment since Δ does not affect Sfxt.

We select two large graphs (vga_lcd and ldoor). Fig. 12(a) and (c) plot the Pfxt runtime of C-PathGen at different Δ. Fig. 12(b) and (d) plot the accumulated path count at each step. To better observe how the workload varies at each step, we select small Δs for both graphs to avoid grouping too many similar-cost paths into one step. For vga_lcd, we select $\Delta = 5, 10, ..., 30$; for ldoor, we select $\Delta = 1, 1.5, ..., 3$. Similar to Table 2, we set k =1M.

Fig. 12 shows that the best runtime is achieved at a moderate step size rather than at the smallest or largest Δ. When Δ is too small, C-PathGen generates few redundant paths, but it advances through the search space in more steps and exposes less parallel work in each step. For example, on vga_lcd, increasing Δ from 5 to 10 reduces the runtime from 205 ms to 73 ms while still terminating at about 1.02M generated paths. This shows that a moderate increase in Δ can improve parallelism without substantially increasing redundant work.

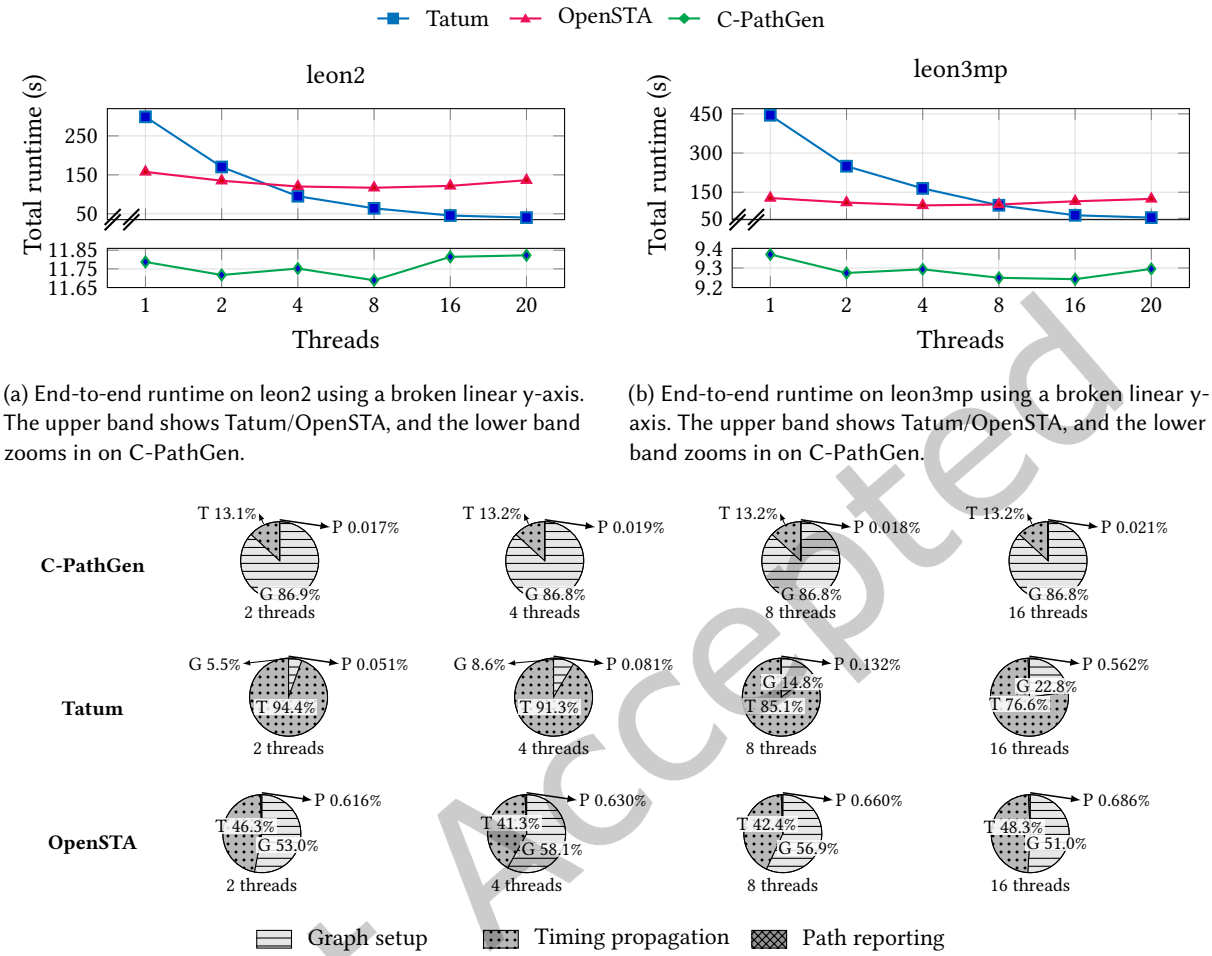
However, larger Δ values eventually reduce work efficiency. For example, on vga_lcd, increasing Δ further to 15 increases runtime to 96 ms because the total generated path count grows to about 1.2M. The same trend appears in Fig. 12(d), where an overly large Δ causes ldoor to generate about 1.8M paths, nearly twice the target k. Therefore, Δ controls a trade-off between per-step parallelism and redundant path generation, and the best performance is obtained at a moderate value that balances both effects.

5.10 Comparison with Tatum and OpenSTA

In addition to comparing C-PathGen with its predecessor PathGen and OpenTimer, we conduct the same performance evaluation against two state-of-the-art CPU-parallel STA engines, Tatum [87] and OpenSTA [12]. Beyond end-to-end runtime, we analyze the runtime breakdown of each engine across three major STA stages, *graph setup*, *timing propagation*, and *path reporting*, to identify the sources of performance differences. Graph setup includes design parsing, graph construction, and levelization. Timing propagation includes the computation of the arrival time and required time. Path reporting measures the time to extract the timing paths after propagation. For C-PathGen, the breakdown uses suffix tree construction as the timing propagation stage and prefix tree expansion as the path reporting stage. All measurements use k = 10K paths.

Please note that end-to-end runtime comparisons with standalone STA engines should be interpreted carefully, as they include different software tradeoffs beyond the core CPG algorithm. For example, standalone engines like OpenSTA incorporate additional functionalities, including design parsing infrastructure, complex internal data structures, scripting interfaces, and runtime management mechanisms, which contribute to overall execution time but are orthogonal to the CPG algorithm itself. Therefore, while the end-to-end comparison provides a practical perspective on the overall runtime, the path reporting stage offers a more direct evaluation of the algorithmic advantage of C-PathGen. We encourage readers to focus on this stage when assessing the effectiveness of our proposed path-generation approach.

Fig. 13(a) and (b) plot the end-to-end runtime of Tatum, OpenSTA, and C-PathGen on leon2 and leon3mp. Since they run at different time scales, we break the y-axis to two parts: the upper part shows Tatum and OpenSTA, while the lower part zooms in on C-PathGen. The runtimes of Tatum and OpenSTA remain higher than C-PathGen at all thread counts. For example, on leon3mp at 16 threads, C-PathGen takes 9.24 s, compared with 61.68 s for Tatum and 115.30 s for OpenSTA. As discussed above, this difference does not indicate one is more efficient than the other; it only reflects the algorithmic scope of each tool: Tatum and OpenSTA build rich timing data and propagate timing through the graph, while C-PathGen focuses on using light timing data to perform shortest-path workload in Sfxt and MLQ scheduling workload in Pfxt.



(a) End-to-end runtime on leon2 using a broken linear y-axis. The upper band shows Tatum/OpenSTA, and the lower band zooms in on C-PathGen.

(b) End-to-end runtime on leon3mp using a broken linear y-axis. The upper band shows Tatum/OpenSTA, and the lower band zooms in on C-PathGen.

(c) Runtime breakdown of C-PathGen, Tatum, and OpenSTA on leon3mp. Each row plots the runtime breakdown at different thread counts: graph setup (G), timing propagation (T), and path reporting (P).

Fig. 13

Tatum's performance scales better with thread count. For example, on leon3mp, its runtime decreases from 249.64 s at two threads to 61.68 s at 16 threads, while OpenSTA changes from 110.55 s to 115.30 s. We attribute this difference to where each engine spends its time. Tatum's runtime is dominated by parallel timing propagation, which benefits from more threads in this experiment. OpenSTA spends a larger portion of its runtime in setup and reporting, so its end-to-end runtime does not improve much with additional threads. As a result, Tatum overtakes OpenSTA between 4 and 8 threads: OpenSTA is faster at 4 threads (99.57 s vs. 164.61 s), but Tatum is slightly faster at 8 threads (100.14 s vs. 103.11 s).

In Fig. 13(c), to study how the runtime of each stage changes at different thread counts, the pie charts plot the runtime breakdown of each engine: graph setup (G), timing propagation (T), and path reporting (P) at different thread counts. For Tatum, timing propagation decreases from 94.4% of runtime at 2 threads to 76.6% at 16 threads,

while graph setup grows from 5.5% to 22.8%. This is because, as discussed earlier, the parallel propagation stage becomes faster, so setup occupies a larger portion of the remaining runtime. On the other hand, OpenSTA's timing propagation portion is more stable, from 46.3% to 48.3%. This is because its threading model on this workload does not improve the performance of any stage much. Finally, Fig. 13(c) shows that path reporting is a small fraction of end-to-end runtime for all three engines at $k = 10$K. This means the total-runtime curves in Fig. 13(a) and (b) are dominated by graph setup and timing propagation. We therefore study path reporting runtime separately in Fig. 14.

Fig. 14 compares the performance of the path reporting stage of C-PathGen, Tatum, and OpenSTA. For all three engines, runtime generally increases with larger k because more paths must be selected and processed. However, the scaling behavior differs across engines. For example, on leon3mp, C-PathGen increases from 0.24 ms at $k = 1$ to 1.74 ms at $k = 10$K, Tatum increases from 22.68 ms to 123.28 ms, and OpenSTA increases from 102.77 ms to 628.94 ms. We attribute this scaling difference to the amount of timing information each engine must recover and materialize during path reporting. C-PathGen reports top-k paths from its suffix-tree/prefix-tree CPG representation, so each reported path is recovered from compact node representations with very little STA metadata to construct. Tatum collects the worst setup paths from its abstract timing graph and timing-tag data structures, which requires constructing `TimingPath` objects from the analyzed timing state. OpenSTA reports gate-level timing checks through `report_checks`, so each reported path also carries detailed STA metadata such as slew, capacitance, input pins, nets, fanout, and slack. As k grows, these additional per-path recovery and construction costs make Tatum and OpenSTA scale more steeply than C-PathGen.

For C-PathGen, the runtime curves are not strictly monotonic at small k values. For example, $k = 1$ is slower than $k = 10$ on both circuits: 0.23 ms vs. 0.026 ms on leon2 and 0.24 ms vs. 0.020 ms on leon3mp. This is because for both path counts, C-PathGen needed to initialize and complete one path expansion step. Both $k = 1$ and $k = 10$ finish in one step on both circuits, so the runtime is dominated by the workload in the first step rather than by k. Once k grows beyond this one-step region, the trend becomes more regular from $k = 100$ to $k = 10$K. On the other hand, the runtimes of Tatum and OpenSTA show higher increases at high k values. For example, Tatum's runtime is nearly flat from $k = 10$ to $k = 100$ (29.61–29.68 ms on leon2 and 20.17–19.74 ms on leon3mp), but increases sharply when k becomes large. From $k = 1$K to $k = 10$K, Tatum rises from 41.53 ms to 150.27 ms on leon2 and from 27.66 ms to 123.28 ms on leon3mp. This is because small-k queries are dominated by reporting overhead that is paid once per query, including setting up the path collection state and accessing Tatum's timing graph and timing tags. At large k, the per-path cost becomes visible because the engine must collect, construct, and store many `TimingPath` objects. Additionally, we see OpenSTA's runtime increases more steeply than Tatum's. For example, on leon3mp, OpenSTA grows from 150.57 ms at $k = 1$K to 628.94 ms at $k = 10$K. This is because the OpenSTA runtime includes a richer reporting flow: `report_checks` emits gate-level timing reports with fields such as slew, capacitance, input pins, nets, and fanout. These additional per-path report construction costs contribute to a longer path reporting runtime than Tatum.

5.11 Performance Comparison with GPU-Parallel CPG Algorithms

To evaluate the relative benefits of CPU and GPU parallelism for CPG applications, we compare the performance of C-PathGen with a state-of-the-art GPU-parallel CPG algorithm, G-PathGen [4]. G-PathGen is an exact GPU-parallel CPG algorithm that generates path candidates on the GPU using a threshold-based exploration strategy. It exploits massive data parallelism to process many candidates concurrently, while still guaranteeing that all paths that may belong to the top-k set are generated before the search advances to less critical regions.

Fig. 15 shows the path generation performance comparison between C-PathGen and G-PathGen. For small and medium path counts, C-PathGen is faster than G-PathGen. For example, on leon2, C-PathGen takes 0.03 ms at $k = 10$ and 1.76 ms at $k = 10$K, while G-PathGen takes 0.67 ms and 5.37 ms. This is because C-PathGen does not

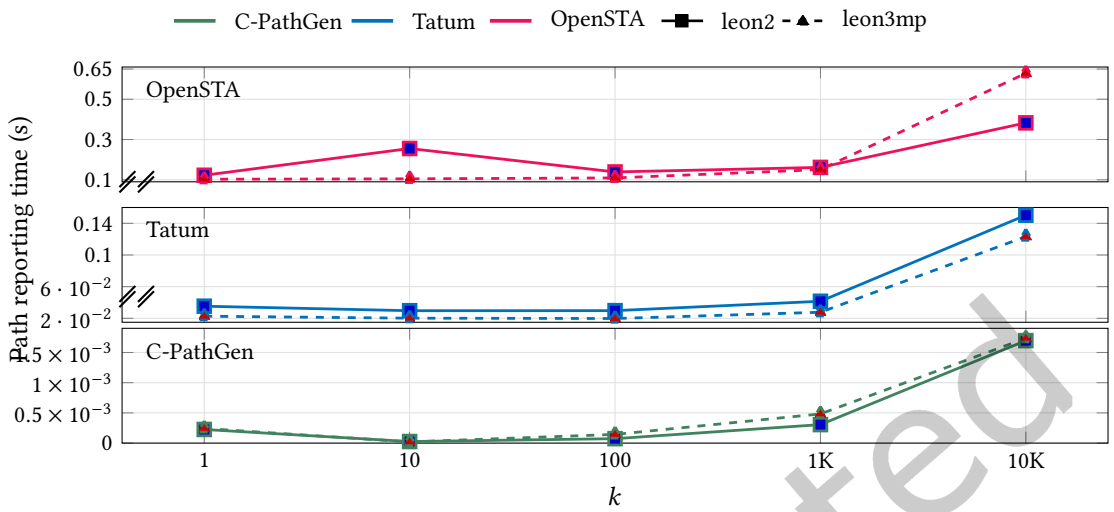


Fig. 14. Path reporting runtime comparison of C-PathGen, Tatum, and OpenSTA at different path counts on leon2 and leon3mp. The three bands use separate y-axis ranges for OpenSTA, Tatum, and C-PathGen. Colors distinguish engines; marker shapes and line styles distinguish circuits.

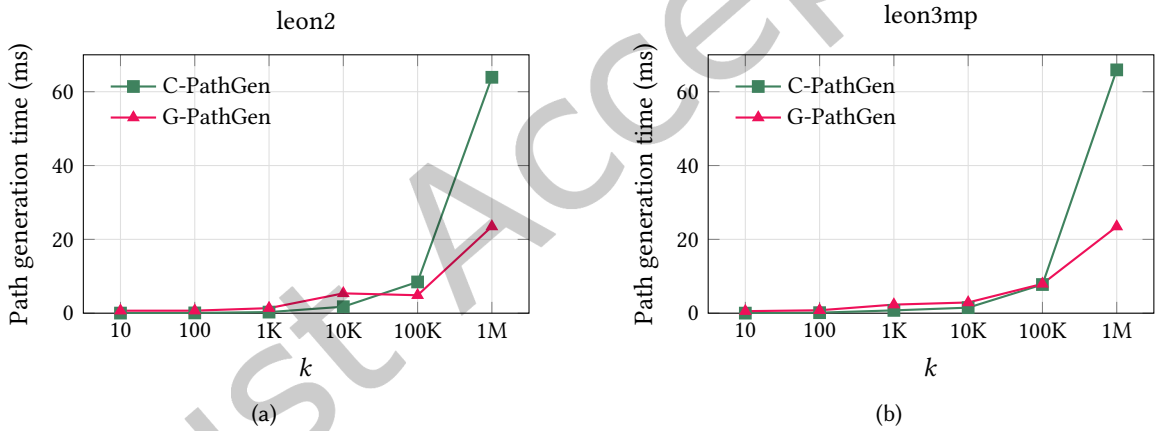


Fig. 15. Path generation performance comparison between C-PathGen and G-PathGen at different path counts on (a) leon2 and (b) leon3mp.

incur GPU-specific overhead such as data movement, kernel launch, and CUDA runtime scheduling. However, when k becomes large enough, G-PathGen starts to outperform C-PathGen, showing the efficiency advantage of GPU execution for sufficiently large path queries. For example, on leon2, the crossover occurs between k = 10K and k = 100K, at which G-PathGen is slower at k = 10K (5.37 ms vs. 1.76 ms), but faster at k = 100K (4.86 ms vs. 8.46 ms) and k = 1M (23.48 ms vs. 63.91 ms). On leon3mp, the crossover occurs later, where C-PathGen is still slightly faster at k = 100K (7.77 ms vs. 7.95 ms), while G-PathGen becomes faster at k = 1M (23.47 ms vs. 65.92 ms). This is because, at large path counts, the path-generation workload exhibits sufficient data parallelism for thousands of GPU threads to exploit, allowing the resulting throughput gains to outweigh the GPU-related

overheads. In summary, this shows that CPU-parallel CPG is advantageous for small and moderate path queries, while GPU-parallel CPG becomes advantageous for very large path queries.

5.12 Choosing Between CPU-Parallel, GPU-Parallel, and Incremental CPG

Based on the above discussions, we summarize the architectural trade-offs among C-PathGen, GPU-parallel CPG [4], and incremental CPG [3, 5] in Table 4, including supported workloads, exactness, integration requirements, and preprocessing/transfer overheads.

C-PathGen is the most direct fit for multicore CPU STA flows: it does not require accelerator hardware, avoids host-device data movement, and preserves the same general CPU-side graph and timing data organization used by existing timers. GPU-parallel CPG provides a different trade-off. Fig. 15 shows that G-PathGen's GPU kernel becomes faster than C-PathGen at very large path counts; absolute efficiency of GPU over CPU occurs at $k = 1$M. However, this GPU kernel performance advantage does not capture the full end-to-end cost of a GPU flow. In our profiling, practical GPU runs include CPU preprocessing and host-device setup costs of about 8.2–11.3 s, while the GPU path generation kernel is only 0.55–5.4 ms for $k \leq 10$K and 13.5–23.5 ms at $k = 1$M. Therefore, GPU-parallel CPG is most advantageous when the path generation workload is very large and exhibits sufficient data parallelism, or when repeated GPU-resident queries can amortize the setup and data transfer overhead.

In terms of memory usage, Table 5 reports the measured memory footprint of C-PathGen and G-PathGen on large circuit graphs. In particular, the GPU comparison includes both the CPU-side maximum RSS (graph preparation) and the GPU-side allocation peak (path generation GPU kernels), so the table reflects the total memory footprint needed to run G-PathGen. This table shows that G-PathGen uses less total memory than C-PathGen on these workloads. This is mainly because G-PathGen stores the graph and paths in compact array-based GPU data structures, while C-PathGen adopts a richer CPU-side representation with graph objects, timing data, suffix tree/prefix-tree state, scheduler metadata, and allocator overhead.

On the other hand, incremental CPG targets a different class of workloads, such as timing-driven optimization and ECO, where the design evolves incrementally and many paths can be reused across iterations. Unlike the CPU/GPU break-even point shown in Fig. 15, its break-even point depends on the locality of design modifiers and an algorithm's ability to maintain reusable path data across modified circuits. Since we focus on the evaluation of one-shot CPG queries rather than repeated design updates, incremental CPG is not a directly comparable baseline in these figures. Therefore, we only discuss incremental CPG briefly in Table 4.

6 Conclusion

In this article, we have introduced C-PathGen, an exact and efficient CPU-parallel CPG algorithm. Building on our prior CPU-parallel CPG algorithm, PathGen, C-PathGen overcomes its inaccuracy by filtering out non-critical paths. Additionally, to eliminate contention caused by atomic operations on generated paths, we introduce a new exploration strategy that processes the paths directly in place. Compared to PathGen, C-PathGen is 6.8×–10.3× faster while producing exact results when dealing with large path queries on industrial circuit graphs. Future work includes extending C-PathGen to other task-graph heterogeneous workloads inspired by our prior research [3, 5–11, 13–33, 36–51, 53, 54, 56–62, 64–66, 68–86, 90–99].

Acknowledgment

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141. The authors would like to thank the reviewers' time and effort in improving this manuscript.

Table 4. Trade-offs among CPU-parallel C-PathGen, GPU-parallel G-PathGen, and incremental CPG.

Property	**C-PathGen on multicore CPUs**	**GPU-parallel CPG [4]**	**Incremental CPG [3, 5]**
Supported workload	Large path queries.	Large path queries.	Repeated path queries after localized design updates.
Exactness	Exact.	Exact for G-PathGen [4].	Exact.
Hardware requirements	Multicore CPU.	CUDA-capable GPU.	Multicore CPU.
Integration complexity	Low: stays in the CPU STA data flow.	High: requires GPU data structures, kernels, and runtime support.	Low: stays in the CPU STA flow and tracks only reusable timing/path state.
Preprocessing and transfer overhead	Low: CPU-side graph preparation; no host-device data movement.	Moderate: CPU-side graph preparation and host-device data movement.	Depends on the size of updated region: low preprocessing overhead for localized changes; high for widespread updates.

Table 5. Memory usage comparison between C-PathGen and G-PathGen on large circuit graphs at $k = 1$M.

Graph	$\|V\|$	$\|E\|$	**C-PathGen total (GB)**	**G-PathGen CPU RSS (GB)**	**G-PathGen GPU peak (GB)**	**G-PathGen total (GB)**
leon3mp	3.3M	4.1M	2.871	1.063	0.529	1.592
leon3mp_x8	27.0M	32.3M	22.415	6.868	2.945	9.813
netcard	3.9M	4.9M	3.291	1.172	0.533	1.705
netcard_x8	32.0M	38.4M	26.260	7.954	3.525	11.480
leon2	4.3M	5.2M	3.595	1.303	0.641	1.944
leon2_x8	34.6M	41.0M	28.687	8.464	3.727	12.191

CPU RSS is the maximum resident set size measured by `/usr/bin/time -v`. GPU peak is the CUDA allocation peak reported by the NVIDIA profiler [88]. G-PathGen total is CPU RSS plus GPU peak.

References

[1] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge Workshop*. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science.

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

[3] Che Chang, Cheng-Hsiang Chiu, Boyang Zhang, and Tsung-Wei Huang. 2024. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 771–774.

[4] Che Chang, Yi-Hua Chung, Cheng-Hsiang Chiu, Wan-Luan Lee, Boyang Zhang, Ulf Schlichtmann, Ing-Chao Lin, Xiangyao Yu, , and Tsung-Wei Huang. 2026. G-PathGen: An Efficient GPU-Parallel k-Critical Path Generation Algorithm. In *ACM International Conference on Supercomputing (ICS)*.

[5] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[6] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[7] Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Boyang Zhang, , and Tsung-Wei Huang. 2026. SSTA-X: GPU-Accelerated First-Order Block-Based Statistical Static Timing Analysis. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*.

[8] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*. 1–7.

[9] Chih-Chun Chang and Tsung-Wei Huang. 2025. Statistical Timing Graph Scheduling Algorithm for GPU Computation. In *ACM/IEEE Design Automation Conference (DAC)*.

[10] Chih-Chun Chang, Aditya Das Sarma, Cheng-Hsiang Chiu, and Tsung-Wei Huang. 2027. Can ML Frameworks Go Beyond ML? A Case Study in GPU-accelerated VLSI Timing Analysis. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[11] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM International Conference on Parallel Processing (ICPP)*. 565–575.

[12] James Cherry and OpenSTA Contributors. 2026. OpenSTA: Parallax Static Timing Analyzer. https://github.com/The-OpenROAD-Project/OpenSTA. Accessed: 2026-07-10.

[13] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 283–284.

[14] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*. 1388–1389.

[15] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2024. An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 766–770.

[16] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*. 468–479.

[17] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–8.

[18] Cheng-Hsiang Chiu, Chedi Morchdi, Chih-Chun Chang, Cunxi Yu, Yi Zhou, and Tsung-Wei Huang. 2025. Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*.

[19] Cheng-Hsiang Chiu, Chedi Morchdi, Yi Zhou, Boyang Zhang, Che Chang, and Tsung-Wei Huang. 2024. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[20] Yi-Hua Chung, Shui Jiang, Wan Luan Lee, Yanqing Zhang, Haoxing Ren, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

[21] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*.

[22] Serhan Gener, Sahil Hassan, Liangliang Chang, Chaitali Chakrabarti, Tsung-Wei Huang, Umit Ograss, , and Ali Akoglu. 2025. A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*.

[23] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[24] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. 4219–4232.
[25] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.
[26] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*. 721–726.
[27] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.
[28] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[29] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*. 3466–3478.
[30] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.
[31] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*. 4973–4984.
[32] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.
[33] Zizheng Guo, Zuodong Zhang, Wuxi Li, Tsung-Wei Huang, Xizhe Shi, Yufan Du, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–9.
[34] Jin Hu, Greg Schaeffer, and Vibhor Garg. 2015. TAU 2015 contest on incremental timing analysis. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 882–889.
[35] Jin Hu, Debjit Sinha, and Igor Keller. 2014. TAU 2014 contest on removing common path pessimism during timing analysis: Special session paper: Common path pessimism removal (CPPR). In *ACM/IEEE ICCAD*.
[36] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–2.
[37] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*. 1–6.
[38] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.
[39] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 746–756.
[40] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.
[41] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.
[42] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*. 1–2.
[43] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*. 1360–1363.
[44] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
[45] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*. 1–4.
[46] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 757–764.
[47] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 1070–1083.
[48] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. In *IEEE Design and Test (DAT)*.
[49] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 1303–1320.
[50] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[51] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*. 588–597.
[52] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 895–902.
[53] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*. 1–6.
[54] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*. 51–59.
[55] Intel. 2025. oneAPI Threading Building Blocks (oneTBB). https://github.com/uxlfoundation/oneTBB. Accessed: 2025-08-10.
[56] Shui Jiang, Hengrui Chen, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. FlatDD: Parallel Quantum Circuit Simulation using Decision Diagram and Flat Array. In *ACM Transactions on Quantum Computing (TQC)*.
[57] Shui Jiang, Hengrui Chen, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. Q-TranSim: Batch Quantum Circuit Simulation using Tensor Transpilation. In *ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
[58] Shui Jiang, Wen Cheng, Yi-Hua Chung, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. QDP: Worst-case Fidelity-aware Qubit Mapping and Routing using Dynamic Programming. In *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*.
[59] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[60] Shui Jiang, Rongliang Fu, Lukas Burgholzer, Robert Wille, Tsung-Yi Ho, and Tsung-Wei Huang. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM International Conference on Parallel Processing (ICPP)*. 388–399.
[61] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[62] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*. 51–61.
[63] Andrew B. Kahng. 2022. The OpenROAD Project: Foundations and Realization of Open and Accessible Design. In *Proceedings of the 2022 International Symposium on Physical Design*. ACM.
[64] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*.
[65] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 278–283.
[66] T.-Y. Lai, Tsung-Wei Huang, , and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*. 1–6.
[67] 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 *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 890–894.
[68] Wan-Luan Lee, Shui Jiang, Dian-Lun Lin, Che Chang, Boyang Zhang, Yi-Hua Chung, Ulf Schlichtmann, Tsung-Yi Ho, , and Tsung-Wei Huang. 2025. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.
[69] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. 2025. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[70] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[71] Wan-Luan Lee, Aditya Das Sarma, Che Chang, Chih-Chun Chang, and Tsung-Wei Huang. 2026. iHyperG: Incremental Hypergraph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.
[72] Zhengxiong Li, Tsung-Wei Huang, , and Umit Ogras. 2026. SET: Stream-Event-Triggered Scheduling for Efficient CUDA Graph Pipelines. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.
[73] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*. 2284–2287.
[74] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[75] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 64–71.
[76] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*. 183–188.
[77] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[78] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*. 435–450.
[79] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 3041–3052.
[80] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*. 151–166.
[81] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing (ICPP)*. 1–12.
[82] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[83] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[84] Pingchuan Ma, Ziang Yin, Qi Jing, Zhengqi Gao, Nicholas Gangi, Boyang Zhang, Tsung-Wei Huang, Rena Huang, Duane Boning, Yu Yao, and Jiaqi Gu. 2026. SP2RINT: Spatially-Decoupled Physics-Constrained Progressive Inverse Optimization for Diffractive Optical Neural Network Training. In *ACM/IEEE Design Automation Conference (DAC)*.
[85] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 89–95.
[86] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*.
[87] Kevin E. Murray and Vaughn Betz. 2018. Tatum: Parallel Timing Analysis for Faster Design Cycles and Improved Optimization. In *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, Naha, Japan, 110–117.
[88] NVIDIA Corporation. 2026. NVIDIA Nsight Systems User Guide. https://docs.nvidia.com/nsight-systems/UserGuide/index.html. Accessed: 2026-07-12.
[89] 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 *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 903–909. doi:10.1109/ICCAD.2015.7372667
[90] Xiangyu Ren, Yuexun Huang, Zhaohui Yang, Yuchen Zhu, Tsung-Wei Huang, Zhiding Liang Tsung-Yi Ho, and Antonio Barbalace. 2026. A Spatial-Topology Preserving Compiler for Quantum Many-Body Systems Simulation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[91] Aditya Das Sarma, Shui Jiang, Wan-Luan Lee, Tsung-Yi Ho, and Tsung-Wei Huang. 2026. TIMBER: A Fast Algorithm for Timing and Power Optimization using Multi-bit Flip-flops. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[92] Aditya Das Sarma, Wan-Luan Lee, Shui Jiang, Boyang Zhang, and Tsung-Wei Huang. 2026. A Differentiable Approach to Task Graph Partitioning: A Case Study in RTL Simulation. In *ACM/IEEE Design Automation Conference (DAC)*.
[93] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei Huang. 2024. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 789–793.
[94] Jie Tong, Zhengxiong Li, Umit Yusuf Ogras, and Tsung-Wei Huang. 2025. A Scalable Code Generation Flow for Heterogeneous Parallel RTL Simulation using MLIR. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.
[95] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.
[96] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[97] Boyang Zhang, Chih-Chun Chang, Yi-Hua Chung, Che Chang, Cheng-Hsiang Chiu, Aditya Das Sarma, , and Tsung-Wei Huang. 2026. G-STAR: GPU-Accelerated Statistical Static Timing Analysis using Level-by-level Replication. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.
[98] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[99] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*. 190–195.

Received 30 April 2026; revised 18 July 2026; accepted 21 August 2026