PathGen: An Efficient Parallel Critical Path Generation Algorithm

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

Boyang Zhang
University of Wisconsin at Madison
Madison, Wisconsin, USA

Cheng-Hsiang Chiu
University of Wisconsin at Madison
Madison, Wisconsin, USA

Dian-Lun Lin
University of Wisconsin at Madison
Madison, Wisconsin, USA

Yi-Hua Chung
University of Wisconsin at Madison
Madison, Wisconsin, USA

Wan-Luan Lee
University of Wisconsin at Madison
Madison, Wisconsin, USA

Zizheng Guo
Peking University
Beijing, China

Yibo Lin
Peking University
Beijing, China

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

Abstract

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 *PathGen*, an efficient CPU-parallel CPG algorithm. PathGen introduces a multi-level queue scheduling framework that can efficiently parallelize the search process of critical paths. Compared to a state-of-the-art single-threaded timer, PathGen is up to 7.4× faster with 16 threads and achieves nearly 100% accuracy when generating one million critical paths on large designs.

ACM Reference Format:
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 *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3658617.3697741

1 Introduction

Critical Path Generation (CPG) is an important step for static timing analysis (STA) applications to analyze the timing criticality of a circuit design [1]. As the design complexity increases, the runtime of CPG can become a bottleneck in many STA engines [28]. To solve this problem, the STA community has developed several CPG algorithms that efficiently generate top-k critical paths. For example, iTimerC [59] introduces a branch-and-bound algorithm to remove redundant path traversals; iitRace [75] introduces a pin coloring strategy to conduct path reduction; OpenTimer [33] 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 promising performance [28], *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 [33], where industrial STA applications typically issue thousands of CPG queries during timing-driven optimization.

To mitigate this runtime challenge, Guo et al. introduced a GPU-accelerated CPG algorithm [18] that expands the critical path search space in parallel. While this GPU-parallel CPG algorithm achieves 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 timing may not exhibit enough data parallelism to benefit from GPU acceleration [22]. 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 the GPU-based approach [18] out of the box due to the difference in parallelism models between GPUs and CPUs. For example, [18] counts on massive parallelism (e.g., thousands of GPU threads) and a GPU-specific path data structure to explore many critical paths at the same time. However, CPU data structure is fundamentally different from GPU and often does 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. We need a dynamic strategy to re-balance the slack priorities in each partition.

To overcome these challenges, we propose *PathGen*, an efficient CPU-parallel CPG algorithm. PathGen builds upon OpenTimer [33]

ASPDAC '25, January 20–23, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-0635-6/25/01.
https://doi.org/10.1145/3658617.3697741

but renovates its implicit path representation algorithm to support parallel expansion of its prefix tree data structure. We summarize three technical contributions of 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.
- We design a geometric slack distribution partitioning strategy to balance the number of paths in each group. By balancing the number of paths in each group, we can reduce the chance of multiple threads accessing the same group simultaneously, minimizing the contention.
- We design a redistribution strategy that dynamically transfers paths between groups to adjust the slack priorities. This strategy allows threads to process these paths in a more accurate order, improving the accuracy of generated paths.

We evaluate PathGen's performance on large circuit benchmarks generated by a popular open-source timer, OpenTimer [33]. Compared to OpenTimer's CPG algorithm, which is inherently single-threaded, PathGen is up to 7.4× faster and achieves nearly 100% accuracy when generating one million critical paths on large designs. We plan to make PathGen open-source to benefit the STA community.

2 Background

2.1 Critical Path Generation

The circuit network is input as a directed-acyclic graph $G = \{V, E\}$. V is a set of n vertices that represent pins of circuit components (e.g., logic gates, flip-flops, etc.). E is a set of m edges that represent pin-to-pin connections. Each edge e is directed from its head vertex u to tail vertex v and is associated with a delay. A path is an ordered sequence of edges $\langle e_1, e_2, ..., e_i \rangle$. The path delay is the summation of delays through all edges of that path. Given a circuit graph G and a positive integer k, a CPG query reports the top-k critical paths in ascending order of path slack (or path delay depending on how the graph is formulated [33]).

2.2 Implicit Path Representation

While several CPG algorithms [33, 59, 75] exist, we adopt the *implicit path representation algorithm* proposed by OpenTimer [33], 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 is a tree order of non-suffix tree edges. Each prefix tree node implicitly represents a path deviated from its parent path. The prefix tree root refers to the shortest path in the suffix tree. Fig. 1(b) shows an example of how a prefix tree node represents a path. The prefix tree root ϕ implicitly represents the shortest path $\langle e_{SB}, e_{BE}, e_{ED}, e_{DT} \rangle$ in the suffix tree. The prefix tree node marked by "e_{SC}" (in gray) implicitly represents the path with prefix $\langle e_{SC} \rangle$ from its parent path (which is the shortest path) deviated on e_{SC} and followed by suffix $\langle e_{CF}, e_{FT} \rangle$ from the suffix tree. Fig. 1(c) illustrates the path $\langle e_{SC}, e_{CF}, e_{FT} \rangle$ by coloring the vertices black. To retrieve the path delay, we record the "deviation cost" of each non-suffix tree edge e: $dvi[e] = dis[tail[e]] + weight[e] - dis[head[e]]$, where $dis[v]$ denotes the shortest distance from vertex v to its destination vertex. Intuitively, deviation cost measures the distance loss by deviating on edge e instead of taking the shortest path to the destination vertex. For example, in Fig. 1(b), e_{SC} has a deviation cost of $dis[tail[e_{SC}]] + weight[e_{SC}] - dis[head[e_{SC}]] = 5$ (where $tail[e_{SC}]$ is C and $head[e_{SC}]$ is S), which means by deviating on e_{SC}, we obtain a path that is 5 units longer than the shortest path from $head[e_{SC}]$ to its destination vertex. Tab. 1 lists the data fields we apply for each prefix tree node [33].

Although we use "deviation cost" and "slack" interchangeably depending on context (e.g., explaining algorithms), these two words are algorithmically equivalent in terms of ranking critical paths.

Constructor	Members
Pfx(p, e, w)	p: parent node, e: deviation edge, w: cumulative $dvi[e]$

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

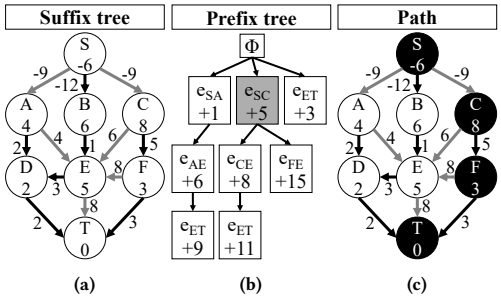

Fig. 1: Implicit path representation using suffix tree and prefix tree. Prefix $\langle e_{SC} \rangle$ + Suffix $\langle e_{CF}, e_{FT} \rangle$= Path $\langle e_{SC}, e_{CF}, e_{FT} \rangle$.

3 PathGen

Inspired by OpenTimer [33], PathGen has two stages: (1) suffix tree construction and (2) parallel prefix tree expansion. Suffix tree construction can be done efficiently with topological relaxations. Through analyzing OpenTimer's runtime breakdown, we discovered that prefix tree expansion takes up most of the runtime when dealing with large path counts. For example, prefix tree expansion takes up almost 80% of the runtime when generating 5 million paths in netcard. Therefore, we focus on the second stage: parallelizing prefix tree expansion. To be clear, to "expand" a prefix tree node means to expand its critical path search space by generating children nodes for this node.

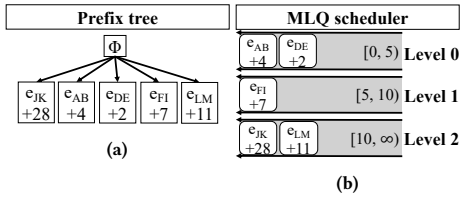

Fig. 2: Illustration of the MLQ scheduling strategy. (a) A prefix tree with five leaf nodes. (b) The MLQ scheduler distributes the prefix tree leaf nodes into three queues according to the deviation costs of these nodes.

3.1 Parallel CPG with Multi-level Queue Scheduling

To generate critical paths concurrently and accurately, the major challenge is that *we need to partition generated prefix tree nodes into multiple groups to expand in parallel while appropriately accommodating their slack priorities.* To overcome this challenge, we distribute the prefix tree nodes to multiple *concurrent queues.* A concurrent queue provides thread-safe access to insertion and deletion. Each queue manages a group of nodes within a particular range of deviation costs, ensuring that nodes with close deviation costs are grouped together. This way, the nodes in the same queue share the same priority, and we can expand them in parallel. Additionally, we arrange the queues into multiple levels, where lower-level queues manage ranges with smaller upper and lower bounds. Also, lower-level queues have higher priority than higher-level queues. This is because lower-level queues manage nodes with smaller deviation costs and must be expanded first, to generate paths more accurately. We refer to this design as *multi-level queue (MLQ) scheduling*. Fig. 2 illustrates an example. Fig. 2(a) shows a prefix tree with five leaf nodes (five nodes to be expanded). Fig. 2(b) illustrates the MLQ scheduler with three queues. Each queue is assigned a specific range of deviation costs: [0, 5), [5, 10), and [10, ∞), respectively. The MLQ scheduler distributes the leaf nodes in Fig. 2(a) to the queues according to the deviation costs of these nodes.

To facilitate the explanation of the MLQ scheduling strategy, we assume that the ranges of deviation costs are equally sized in the rest of Section 3.1. However, in Section 3.2, we will discuss another strategy that organizes the ranges into different sizes to address thread contention.

Algorithm 1: SpurMLQ(*pfx*, *d*, *MLQ*)

```
   Input: prefix tree node pfx, destination vertex d, array of
          concurrent queues MLQ
   Global: array of suffix tree successors successor
1  u ← tail[pfx.e];
2  while u ≠ d
3  |  Foreach e ∈ fanout(u)
4  |  |  if tail[e] == successor[u] then
5  |  |  |  continue;
6  |  |  pfx_new ← new Pfx(pfx, e, pfx.w + dvi[e]);
7  |  |  lvl ← determine level of queue;
8  |  |  MLQ[lvl].push(pfx_new);
9  |  u ← successor[u];
```

Having determined the core data structure to perform parallel prefix tree expansion, Algorithm 1 introduces the prefix tree expansion algorithm of PathGen. The goal of this algorithm is to expand the critical path search space by finding the children nodes for a given prefix tree node, and each child node implicitly represents a new path. Additionally, Algorithm 1 determines the appropriate queue to which each child node should be pushed. We obtain the tail vertex u of the edge associated with the prefix tree node *pfx* (line 1). With the successor array *successor* from the suffix tree, we can visit the vertices along the shortest path until we reach the destination vertex (lines 2 and 9). For each vertex u along the shortest path, we inspect the fanout edges of u (line 3). Because we are looking for non-suffix tree edges to deviate on, for each fanout edge e, we skip *tail*[e] if it is the successor of u, which indicates that e belongs to the suffix tree (line 4:5). Otherwise, we create a new prefix tree node *pfx_new* (line 6). We also record the cumulative deviation cost of *pfx_new* (line 6). We push *pfx_new* to its corresponding queue according to the cumulative deviation cost of *pfx_new* (line 7:8).

Algorithm 2: PathGen(k, P, *MLQ*)

```
   Input: path count k, prefix tree P, destination vertex d,
          array of concurrent queues MLQ
   Output: array of critical paths Ψ
1  atomic_num_paths ← 0;
2  Ψ ← ϕ;
3  MLQ[0].push(P.root);
4  while atomic_num_paths < k
5  |  if all queues in MLQ are empty then
6  |  |  break;
7  |  launch_async_task {
8  |  |  lvl ← 0;
9  |  |  while MLQ[lvl] is empty
10 |  |  |  if lvl == MLQ.size() - 1 then
11 |  |  |  |  break;
12 |  |  |  lvl ← lvl + 1;
13 |  |  node ← MLQ[lvl].pop();
14 |  |  if node == nullptr then
15 |  |  |  return;
16 |  |  SpurMLQ(node, d, MLQ);
17 |  |  path ← recover path from node;
18 |  |  Ψ ← Ψ ∪ path;
19 |  |  atomic_num_paths ← atomic_num_paths + 1;
20 |  };
21 sync_all_threads();
22 sort Ψ in ascending order of deviation costs;
23 return Ψ;
```

Algorithm 2 describes the most important component of PathGen. The goal of this algorithm is to search through the queues and find the non-empty lowest-level queue, then assign an idle thread to pop a node from this queue to perform expansion. We strictly select the lowest-level queue because this queue manages the nodes with higher priorities than higher-level queues. We must expand these nodes first to generate accurate path results. We initialize an atomic

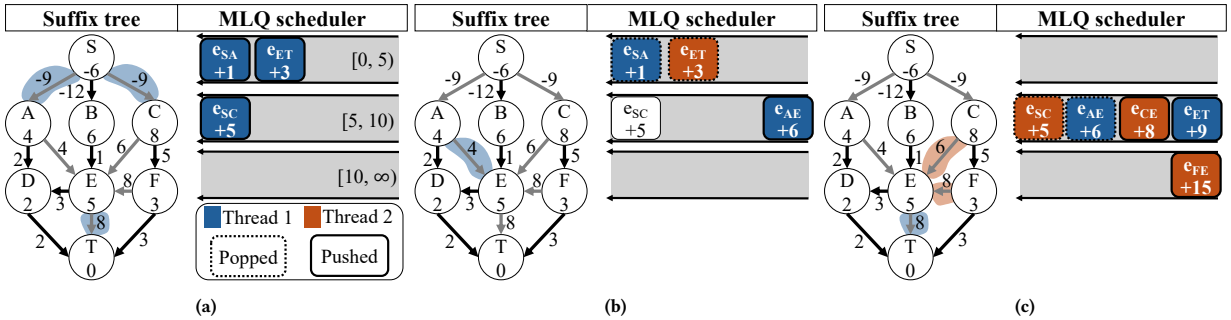

Fig. 3: Illustration of Algorithm 2. (a) A suffix tree and a MLQ scheduler that has three concurrent queues. We have already expanded the prefix tree root and obtained Pfx(e_{SA}, 1), Pfx(e_{ET}, 3), and Pfx(e_{SC}, 5). (b) Since the level-0 queue is non-empty and has the highest priority, thread 1 pops Pfx(e_{SA}, 1) from the level-0 queue to expand. Thread 1 generates Pfx(e_{AE}, 6). Pfx(e_{AE}, 6) lies within the range of the level-1 queue. Thread 2 pops Pfx(e_{ET}, 3) from the level-0 queue to expand. Thread 2 generates no node because e_{ET} already reaches the destination vertex T. (c) Since the level-0 queue is empty, we move on to the next level. Thread 2 pops Pfx(e_{SC}, 5) from the level-1 queue to expand. Thread 2 generates Pfx(e_{CE}, 8) and Pfx(e_{FE}, 15). Pfx(e_{CE}, 8) lies within the range of the level-1 queue. Pfx(e_{FE}, 15) lies within the range of the level-2 queue. Thread 1 pops Pfx(e_{AE}, 6) from the level-1 queue to expand. Thread 1 generates Pfx(e_{ET}, 9). Pfx(e_{ET}, 9) lies within the range of the level-1 queue.

counter *atomic_num_paths* to atomically keep track of the number of critical paths we have discovered (line 1). We initialize a solution set Ψ to record the critical paths (line 2). Since the prefix tree root has a deviation cost of zero, we push it to the level-0 queue to start the expansion (line 3). We then move on to the main path search loop (line 4:20). If all the queues in *MLQ* are empty, which indicates that we have no more nodes to expand, we terminate the loop (line 5:6). Until we have generated enough critical paths, we use Taskflow [38, 43, 44]'s asynchronous tasking library to launch an asynchronous task, and an idle thread will immediately pick up this task and execute (line 7:20). Inside this task, we initialize a level counter *lvl* to record which queue we will select to pop a node (line 8). We loop through the queues to find the non-empty lowest-level queue (line 9:12). Once we have decided *lvl*, we attempt to pop a node from *MLQ*[*lvl*] (line 13). However, since multiple threads are popping nodes from *MLQ*[*lvl*], these threads may have already cleared *MLQ*[*lvl*]. In this case, we get an empty node and terminate this task (line 14:15). Otherwise, we use Algorithm 1 to expand this node and push its children to their corresponding queues (line 16). Since this node only implicitly represents a critical path, we need to explicitly perform path recovery (line 17) to get the path trace, which can be done with OpenTimer [33]'s path recovery algorithm. We record the path in the solution set Ψ (line 18) and increment the atomic path counter (line 19). Finally, we synchronize all threads to ensure the completion of all the running tasks (line 21). We sort the paths in ascending order of deviation costs since multiple threads generate them out of order (line 22).

Fig. 3 illustrates an example of Algorithm 2. We use two threads and three queues in this example. We assign the ranges of deviation costs [0, 5), [5, 10), and [10, ∞) to the level-0, -1, and -2 queues, respectively. "e_{ij}" refers to the edge pointing from vertex i to vertex j. The numbers on the prefix tree nodes in the queues represent the deviation costs of the nodes. For ease of reading, we refer to a prefix tree node that is associated with edge e_{ij} and deviation cost N as "Pfx(e_{ij}, N)". Fig. 3(a) shows a suffix tree on the left. On the right, the level-0 queue manages Pfx(e_{SA}, 1) and Pfx(e_{ET}, 3), and the level-1 queue manages Pfx(e_{SC}, 5). Fig. 3(b) shows that since the level-0 queue is non-empty and has the highest priority, thread 1 pops Pfx(e_{SA}, 1) from the level-0 queue to expand and generates a new node Pfx(e_{AE}, 6). Pfx(e_{AE}, 6) lies within the range assigned to the level-1 queue, so we push it to the level-1 queue. Thread 2 pops Pfx(e_{ET}, 3) from the level-0 queue to expand. This expansion generates no node because e_{ET} already reaches the destination vertex T. Fig. 3(c) repeats the same procedure, except that since the level-0 queue is empty, we move on to the level-1 queue. Full description is in the caption of Fig. 3.

3.2 Partition of Slack Distribution

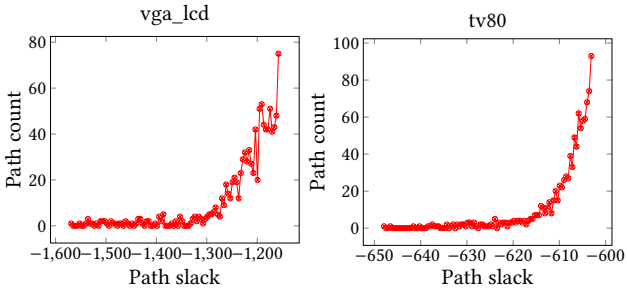

Fig. 4: Top-1K path slack distribution reported by OpenTimer [33].

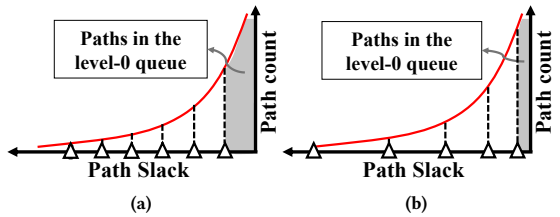

Fig. 5: Two slack distribution partitioning strategies. (a) Equal partitioning results in an unbalanced number of paths in each range. (b) Geometric partitioning results in a more balanced number of paths in each range.

With the proposed MLQ scheduling, our next step is to decide the distribution of the prefix tree nodes across all queues. Specifically,

we need to decide for each queue its range of deviation costs by partitioning slack distribution. However, the design of the partitioning strategy affects the balance of path counts in each queue. When path counts become unbalanced, certain queues experience high thread contention. We need a strategy that *balances the path counts in each queue to minimize contention.*

To this end, we need to analyze the slack distribution. Fig. 4 plots the top-1K path slack distribution for vga_lcd and tv80, reported by OpenTimer [33]. Both circuits exhibit highly localized distribution. Taking vga_lcd for example, over 50% of the slacks lie within the range from -1150 to -1250. Atop this analysis, Fig. 5 illustrates two slack distribution partitioning strategies. The white triangles represent the bounds of the partitions. The red curve in Fig. 5 approximates the path counts as a function of the path slacks. With the distribution curve, we can use the area (in gray) under the curve to approximate the path counts in a certain range. As shown in Fig. 5(a), a naive strategy is to partition the slacks into equal ranges. This strategy has a drawback: the rightmost range (assigned to the level-0 queue) manages significantly more paths than other ranges. This causes the level-0 queue to experience higher-frequency thread access than other queues, leading to high thread contention. To solve this issue, we introduce the geometric slack distribution partitioning strategy. As shown in Fig. 5(b), the geometric partitioning strategy organizes ranges based on a geometric sequence, which grows the ranges as we get further away from the most critical slack (rightmost on the x-axis). As will be discussed in Section 4.3, this results in a more balanced number of paths in each range (a more balanced area under the curve within each range) than the equal partitioning strategy, which minimizes thread contention.

3.3 Node Redistribution

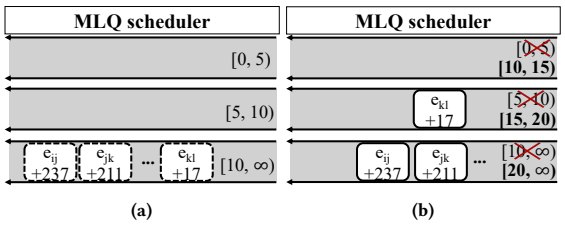

Fig. 6: Illustration of the node redistribution strategy. (a) Less accurate expansion order at the highest-level queue. Pfx(e_{ij}, 237) and Pfx(e_{jk}, 211) are expanded earlier than Pfx(e_{kl}, 17). (b) Node redistribution re-assigns Pfx(e_{kl}, 17) to the level-1 queue, so it is expanded earlier than Pfx(e_{ij}, 237) and Pfx(e_{jk}, 211).

By examining the experimental results, we discovered that Algorithm 2 produces very inaccurate results, especially when we use too few queues. For example, Algorithm 2 has a minimum accuracy of less than 85% in leon2 (see Fig. 8 in Section 4) when using 40 queues. The reason for such inaccuracy is that the highest-level queue's assigned range has an infinite upper bound. This indicates that the highest-level queue will manage nodes with very unbalanced deviation costs. In this case, if we treat all the nodes in the highest-level queue as sharing the same priority, PathGen may expand the nodes with high deviation costs first instead of the nodes with low deviation costs, leading to inaccurate path results. Fig. 6(a) illustrates the mentioned situation with three queues. Pfx(e_{ij}, 237) and Pfx(e_{jk}, 211) are in the front of the highest-level queue, so they are expanded earlier than Pfx(e_{kl}, 17). However, to get higher accuracy, we are supposed to expand Pfx(e_{kl}, 17) earlier.

To solve this issue, we introduce the node redistribution strategy. The goal of this strategy is to update the deviation cost range assigned to each queue, so we can re-balance the deviation costs by distributing the nodes in the highest-level queue to lower-level queues. In other words, *we treat nodes in the highest-level queue as not having priorities.* This way, by distributing them to lower-level queues, we re-assign priorities to these nodes. Fig. 6(b) illustrates the node redistribution strategy. Once any thread reaches the highest-level queue, we update the range of deviation cost assigned to each queue. Since from this point onwards, the prefix tree expansion only generates nodes with deviation costs larger than or equal to 10, we update the level-0 queue's range from [0, 5) to [10, 15). We update the level-1 queue's range from [5, 10) to [15, 20). We update the range of the level-2 queue from [10, ∞) to [20, ∞). After updating the ranges, we re-assign Pfx(e_k, 17) to the level-1 queue, so it is expanded earlier than Pfx(e_i, 237) and Pfx(e_j, 211). This strategy expands the nodes in a more accurate order.

Algorithm 3: NR(*lvl*, *MLQ*)

```
Input: selected queue level lvl, array of concurrent queues
       MLQ
Global: atomic boolean variable is_updating, is_redistr
1  if lvl == MLQ.size() - 1 then
2      if !is_updating.atomic_exchange(true) then
3          update ranges of deviation costs of MLQ;
4          is_updating ← false;
5      if !is_redistr.atomic_exchange(true) then
6          tmp_q ← move from MLQ[lvl];
7          while tmp_q is not empty
8              node ← tmp_q.pop();
9              while is_updating == true
10                 wait;
11             new_lvl ← determine level of queue;
12             MLQ[new_lvl].push(node);
13         is_redistr ← false;
14     return;
```

Algorithm 3 describes the node redistribution strategy. We use two global atomic boolean variables, *is_updating* and *is_redistr*, both initialized to false. *is_updating* checks if any thread is updating the ranges of deviation costs. *is_redistr* checks if any thread is redistributing the nodes. To avoid data race, we disallow threads from simultaneously updating the ranges. If any thread reaches the highest-level queue (line 1), we perform an atomic exchange operation on *is_updating* to set it to true. Note that atomic exchange returns the value of *is_updating* before the operation. If the return value is false, no threads are updating the range. *is_updating* is set to true (line 2) and we can safely update the ranges of deviation costs (line 3). After updating the ranges, we reset *is_updating* to false (line 4), allowing other threads to update the ranges. We disallow threads from simultaneously performing node redistribution because this process can be very time-consuming. We prefer that one thread

redistributes the nodes and others continue expanding the nodes in lower-level queues. Similarly, we perform an atomic exchange operation on *is_redistr* to set it to true. If the return value is false, that means no thread is performing node redistribution. *is_redsitr* is set to true (line 5) and we can safely perform node redistribution (line 6:12). We move all the nodes from the highest-level queue to a temporary queue *tmp_q* (line 6), to prevent interference with other threads inserting nodes to the highest-level queue. Until we have cleared *tmp_q* (line 7), we continuously pop *node* from *tmp_q* (line 8). We check if any thread is updating the ranges (line 9) to avoid data race. If so, we wait until the update is completed (line 10). Otherwise, we push *node* to the queue it belongs to (line 11:12). After node redistribution, we reset *is_redistr* to false (line 13), allowing other threads to perform node redistribution.

4 Experimental Results

We implemented PathGen in C++ and compiled it with GCC 11.4.0 on a 4.8-GHz 64-bit Linux machine of an Intel Core i5-13500 Processor. We enable the optimization flag `-O3` and C++17 standard `-std=c++17`. We use Taskflow [**?**] and the Moodycamel concurrent queue [2] to implement the proposed algorithms. We select seven large circuits generated by OpenTimer [33] to evaluate PathGen's performance. We only measure the runtime of the prefix tree expansion algorithm in PathGen and OpenTimer since we use the same suffix tree construction algorithm as OpenTimer. For large CPG queries, prefix tree expansion takes the majority of the runtime. For example, prefix tree expansion takes up almost 80% of the runtime when generating 5 million paths in netcard. We implemented the sequential CPG algorithm based on OpenTimer. OpenTimer is our sole comparison target because it outperforms existing methods in both time and space complexities. All data is an average of 15 runs.

We do not compare with the GPU-parallel approach [18] as it is an orthogonal direction. Such a comparison is also not fair due to different architecture and application needs.

4.1 Overall Performance Comparison

Tab. 2 compares the runtime, memory usage, and accuracy between OpenTimer and PathGen. We measure the accuracy by comparing the deviation costs generated by OpenTimer and PathGen, using OpenTimer as the golden reference. As shown in Tab. 2, PathGen outperforms OpenTimer in all circuits. For example, PathGen is 7.3× and 7.4× faster than OpenTimer in netcard and leon2. PathGen is almost as accurate as OpenTimer. For example, PathGen achieves 99.9% and 100% accuracy in wb_dma and leon2. PathGen consumes more memory as it uses multiple concurrent queues, as opposed to OpenTimer using only one priority queue.

4.2 Performance at Different Thread Counts

Fig. 7 shows the speedup of PathGen over OpenTimer at different thread counts. As we increase the thread count, the speedup increases. Taking netcard for example, the speedup increases from 4× to over 7×. This is because the more threads we use, the faster we clear a queue and move on to another one. However, there is a tradeoff between thread count and thread contention. As we use more threads, they need to wait longer for their turn to access the queue. Therefore, using the maximum thread count does not always yield the optimal speedup. Taking netcard for example, using 16 threads yields the optimal speedup (over 7×). On the other hand, We see that the speedup at 20 threads for leon2 (over 5×) has not yet reached the optimal.

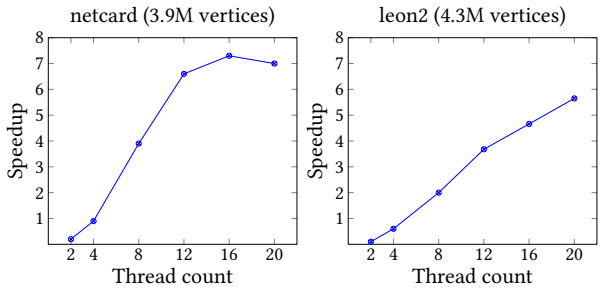

Fig. 7: Speedup of PathGen over OpenTimer at different thread count.

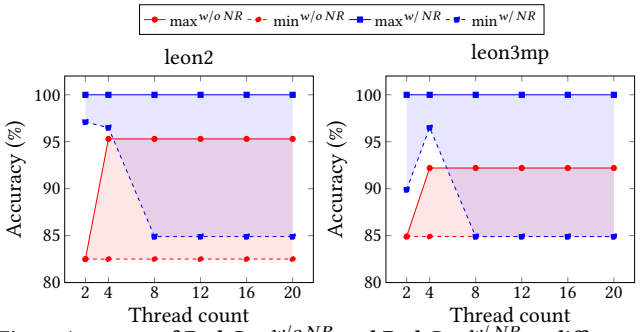

Fig. 8: Accuracy of PathGen$^{w/o\ NR}$ and PathGen$^{w/\ NR}$ at different thread counts.

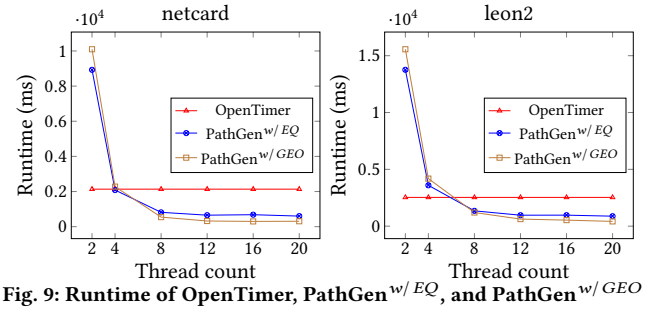

Fig. 9: Runtime of OpenTimer, PathGen$^{w/\ EQ}$, and PathGen$^{w/\ GEO}$ at different thread counts with different partitioning strategies.

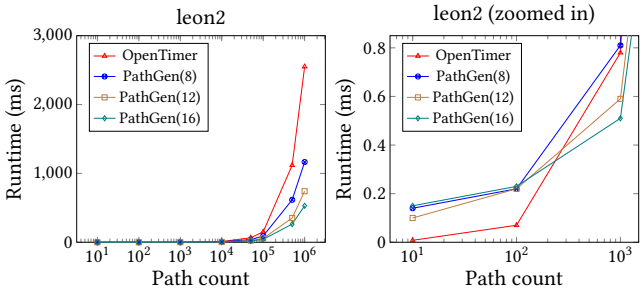

Fig. 10: Runtime vs. path count at different thread counts.

Tab. 2: Overall performance comparison between OpenTimer [33] (sequential CPG) and PathGen (parallel CPG).

Circuit	$\|V\|$	$\|E\|$	Path count (K)	OpenTimer [33] Runtime (ms)	OpenTimer [33] Mem. (MB)	PathGen (16 threads) Runtime (ms)	PathGen (16 threads) Mem. (MB)	PathGen (16 threads) Avg. accuracy (%)
wb_dma	13124	16593	20	4.3	19.3	4.1 (**1.03**×)	24.5	99.9
des_perf	303690	387291	500	341.6	347.9	122.3 (**2.7**×)	461.3	100
vga_lcd	397816	498873	1000	1076.5	552.4	401.9 (**2.6**×)	954.5	100
leon3mp	3376842	4148798	1000	2243.1	3261.7	400.5 (**5.6**×)	4540.9	100
netcard	3999174	4903397	1000	2135.8	3574.6	292.5 (**7.3**×)	4199.7	100
leon2	4328285	5273106	1000	2552.8	4065.4	344.9 (**7.4**×)	5774.3	100

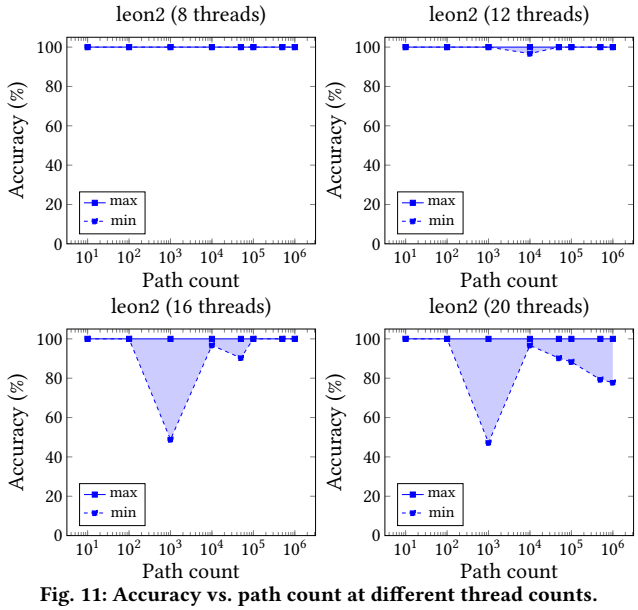

Fig. 11: Accuracy vs. path count at different thread counts.

4.3 Effectiveness of Node Redistribution

Fig. 8 plots the accuracy of PathGen at different thread counts for leon2 and leon3mp using 40 queues. We use 40 queues to investigate the effectiveness of node redistribution because low accuracy occurs especially when we use too few queues. "$w/o\ NR$" refers to the proposed algorithm without node redistribution. "$w/\ NR$" refers to the proposed algorithm with node redistribution. PathGen with the same thread count can produce varying accuracy. This is because threads may fail to complete all the atomic push operations before the algorithm terminates, and these atomic push operations may include nodes with higher priorities. If the algorithm terminates before threads finish processing these higher-priority nodes, the accuracy drops. Therefore, we measure both the minimum and maximum accuracy to investigate the accuracy distribution. PathGen$^{w/\ NR}$ is more accurate than PathGen$^{w/o\ NR}$ in terms of both the minimum and maximum accuracy. Taking leon2 for example, The red area shows that the accuracy of PathGen$^{w/o\ NR}$ ranges from 82% to 95%, whereas the blue area shows that the accuracy of PathGen$^{w/\ NR}$ ranges from 86% to 100%. This is because our node redistribution strategy updates the range of deviation costs assigned to each queue. We also redistribute the nodes from the highest-level queue to the lower-level queues. Therefore, threads can prioritize the higher-priority nodes, improving overall accuracy.

4.4 Performance of Different Slack Distribution Partitioning Strategies

Fig. 9 plots the runtime of PathGen and OpenTimer at different thread counts with different partitioning strategies for netcard and leon2. "PathGen$^{w/\ EQ}$" refers to the proposed algorithm with each queue assigned an equal range of deviation costs. "PathGen$^{w/\ GEO}$" refers to the proposed algorithm with each queue assigned a range of deviation costs based on a geometric sequence. PathGen$^{w/\ GEO}$ and PathGen$^{w/\ EQ}$ are both slower than OpenTimer at two and four threads. Taking netcard for example, The runtimes of PathGen$^{w/\ GEO}$ and PathGen$^{w/\ EQ}$ are 10.1 and 8.9 seconds at two threads (4.6× and 4.1× slower than OpenTimer). Similar to what we discussed in Section 4.2, this is due to PathGen using many queues. The fewer threads we use, the slower we clear the queues. PathGen$^{w/\ GEO}$ and PathGen$^{w/\ EQ}$ both outperform OpenTimer starting from eight threads. Taking leon2 for example, PathGen$^{w/\ GEO}$ is 3.9×, 6.6×, and 7.1× faster than OpenTimer at eight, 16, and 20 threads. On the other hand, PathGen$^{w/\ EQ}$ is 2.6×, 3.2×, and 3.1× faster than OpenTimer at eight, 16, and 20 threads. This shows that the more threads we use, the faster we clear the queues, thereby increasing PathGen's performance. PathGen$^{w/\ GEO}$ outperforms PathGen$^{w/\ EQ}$ at all thread counts. Taking leon2 for example, PathGen$^{w/\ GEO}$ is 1.5×, 2×, and 2.2× faster than PathGen$^{w/\ EQ}$ at eight threads, 12 threads, and 16 threads. This is because by observing the path slack distribution of the circuits using OpenTimer, we see that the slack distribution is highly localized (i.e., over 50% of the slacks belong to a certain range). Specifically, many deviation costs fall within the range of the lower-level queues. If we assign equal deviation cost ranges to the queues, those at lower levels will suffer from high thread contention due to many concurrent push operations, thereby hampering performance. Conversely, by assigning deviation cost ranges based on a geometric sequence, we can reduce thread contention by distributing some push operations from the lower-level queues to higher-level queues, thereby improving performance.

4.5 Performance at Different Path Counts

Fig. 10 plots the runtime of PathGen at different path counts and thread counts for leon2. "PathGen(N)" refers to PathGen running with N threads. When the path count is smaller than 1K, PathGen is slower than OpenTimer at all thread counts. For example, in the zoomed-in figure (right of Fig. 10), PathGen(8) is about 3× slower than OpenTimer when generating 100 paths. This is because when the path count is small, the scheduling overhead of threads

dominates the runtime, therefore PathGen has no benefit. On the other hand, We start to see the benefit of PathGen when generating 1K paths. For example, in the zoomed-in figure (right of Fig. 10), PathGen(12) and PathGen(16) are 1.3× and 2.5× faster than OpenTimer. When querying 10K or more paths, PathGen outperforms OpenTimer at all thread counts. For example, when generating one million paths, PathGen(8), PathGen(12), PathGen(16) are 2.1×, 3.4×, and 4.7× faster than OpenTimer, respectively.

4.6 Accuracy at Different Path Counts

Fig. 11 plots PathGen's accuracy at different path counts and thread counts for leon2. We plot the minimum and maximum accuracy to investigate how path counts affect the accuracy distribution. As we increase the thread count, we see more drops in minimum accuracy across all path counts. This indicates that using more threads results in a broader distribution of accuracy. For example, we see one drop in minimum accuracy at 10K paths using 12 threads. We see three drops in minimum accuracy at 1K, 10K, and 50K paths using 16 threads. This is because, in Algorithm 2, as the thread count increases, the prefix tree expansion runtime becomes smaller while the number of pending atomic push operations we need to perform increases. To achieve 100% accuracy, we must complete all the pending atomic operations within the prefix tree expansion period. Therefore, as we increase the thread count, we are more likely to fail to complete all these pending atomic operations within the required period, leading to accuracy drops.

5 Conclusion

In this paper, we have introduced PathGen, a parallel CPG algorithm that efficiently groups generated critical paths into multiple concurrent queues of slack priorities. Compared to a popular open-source single-threaded timer, PathGen is up to 7.4× faster and nearly 100% accurate when generating one million paths on large designs. Inspired by the success of GPU computing in graph processing [3–17, 19–21, 23–27, 29–32, 34–37, 39–42, 44–58, 60–74, 76–80], we plan to enhance the performance of PathGen using GPU computing.

Acknowledgement

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.

References

[1] Jayaram Bhasker and Rakesh Chadha. 2009. *Static Timing Analysis for Nanometer Designs: A Practical Approach.* Springer.
[2] Cameron. 2014. A fast multi-producer, multi-consumer lock-free concurrent queue for C++11. https://github.com/cameron314/concurrentqueue.
[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).*
[4] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE DAC.*
[5] 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).*
[6] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC).*
[7] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM ICPP.* 565–575.
[8] 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).*
[9] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC).*
[10] 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).*
[11] 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).*
[12] 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).*
[13] 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).*
[14] 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).*
[15] 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).*
[16] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).
[17] 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).*
[18] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE, 721–726.
[19] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC).*
[20] 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).*
[21] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD).*
[22] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-Accelerated Static Timing Analysis. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* 1–9.
[23] 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).*
[24] 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).*
[25] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).
[26] 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).*
[27] 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).*
[28] 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. https://doi.org/10.1109/ICCAD.2015.7372664
[29] 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).*
[30] 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).*
[31] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing*

Conference (HPEC).

[32] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[33] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 4 (2021), 776–789. https://doi.org/10.1109/TCAD.2020.3007319

[34] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[35] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC).*

[36] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC).*

[37] 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).*

[38] 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).*

[39] 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).*

[40] 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).*

[41] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[42] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test (DAT)* (2021).

[43] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).

[44] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2022).

[45] 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).*

[46] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).*

[47] Tsung-Wei Huang and Martin Wong. 2016. UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2016).

[48] 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).*

[49] Tsung-Wei Huang, P.-C. Wu, and Martin Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM ICCAD.*

[50] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Route: An Ultra-Fast Incremental Maze Routing Algorithm. In *ACM System Level Interconnect Prediction Workshop (SLIP).* 1–8.

[51] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE/ACM ICCAD.*

[52] 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).*

[53] 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).*

[54] 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).*

[55] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM ICPP.* 388–399.

[56] 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).*

[57] 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).*

[58] 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).*

[59] 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. https://doi.org/10.1109/ICCAD.2015.7372665

[60] 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).*

[61] 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).*

[62] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM).*

[63] 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).*

[64] 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).*

[65] 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).*

[66] 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).*

[67] 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).*

[68] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).

[69] 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).*

[70] 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).*

[71] 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).*

[72] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC.*

[73] 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).*

[74] 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).*

[75] 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. https://doi.org/10.1109/ICCAD.2015.7372667

[76] 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).*

[77] Sheng-Han Yeh, Jia-Wen Chang, Tsung-Wei Huang, Shang-Tsung Yu, and Tsung-Yi Ho. 2014. Voltage-Aware Chip-Level Design for Reliability-Driven Pin-Constrained EWOD Chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 33, 9 (2014), 1302–1315.

[78] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC).*

[79] 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 DAC.*

[80] 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).*