iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis

Boyang Zhang
University of Wisconsin-Madison
bzhang523@wisc.edu

Che Chang
University of Wisconsin-Madison
cchang289@wisc.edu

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

Dian-Lun Lin
University of Wisconsin-Madison
dlin57@wisc.edu

Yang Sui
Rice University
ys155@rice.edu

Chih-Chun Chang
University of Wisconsin-Madison
chih-chun.chang@wisc.edu

Yi-Hua Chung
University of Wisconsin-Madison
yihua.chung@wisc.edu

Wan Luan Lee
University of Wisconsin-Madison
wanluan.lee@wisc.edu

Zizheng Guo
Peking University
gzz@pku.edu.cn

Yibo Lin
Peking University
yibolin@pku.edu.cn

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

ABSTRACT

Recent static timing analysis (STA) tools have utilized task dependency graph (TDG) parallelism to enhance the STA runtime performance. Although TDG parallelism shows promising speedup, the overhead of scheduling a TDG can become dominant as the TDG becomes larger. To minimize the scheduling overhead, several TDG partitioning algorithms have been proposed to reduce the TDG size without affecting its task parallelism. Despite improved performance, existing TDG partitioners all fall short of *incremental partitioning*, limiting their practical use in STA tools that support timing-driven operations. To overcome this limitation, we propose iTAP, an incremental TDG partitioner to fully leverage the power of TDG partitioning in task-parallel STA applications. Compared to a state-of-the-art full TDG partitioner, iTAP enhances the overall STA performance by up to 2.97×.

ACM Reference Format:
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 *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/10.1145/3658617.3697738

1 INTRODUCTION

Static Timing Analysis (STA) is a crucial step in the chip design flow to verify the timing constraints of a circuit. STA can be very time-consuming due to the growing circuit complexity. In response, various efforts have been made to parallelize STA [20, 31, 43, 75]. Among these efforts, the task-parallel STA introduced by OpenTimer [31, 43] has proven to be one effective approach because it is free from expensive level-by-level synchronization as in other timers [43, 75]. Specifically, OpenTimer models timing propagation as a *task dependency graph* (TDG). For example, in Figure 1, OpenTimer expresses the STA of the circuit in (a) as the TDG in (b), where each task corresponds to a specific STA task (e.g., delay calculation) and each edge denotes the dependency between two STA tasks. By delegating such a TDG to a task graph scheduling environment [35, 40], OpenTimer can efficiently execute STA in parallel across multicore CPUs.

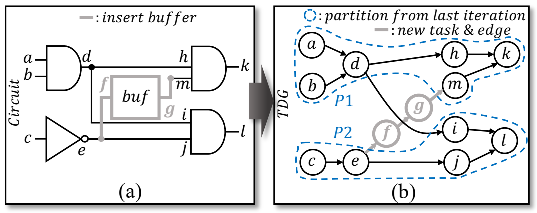

Figure 1: An example of an incremental timing update iteration in STA. The inserted buffer (marked in grey) in the circuit shown in (a) results in two tasks and three edges inserted in the TDG shown in (b). Since the TDG has been partitioned into $P1$ and $P2$ in the last iteration, the inserted tasks and edges introduce a cycle between these two partitions.

Despite the promising performance of TDG parallelism, the scheduling overhead, including building and executing a TDG, can be substantial in large task-parallel STA workloads. For instance, when analyzing a circuit with 1.5 million gates, more than 50% of the runtime is spent on building a TDG with 4 million tasks and 5 million dependencies [78]. However, the optimal parallel execution performance is achievable by only 8 to 16 CPU threads [20]. This indicates that a large TDG is not preferable given a small amount of

ASPDAC '25, January 20–23, 2025, Tokyo, Japan
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0635-6/25/01…$15.00
https://doi.org/10.1145/3658617.3697738

saturated CPU threads. Moreover, most timing propagation tasks finish real quick, typically similar to the time of scheduling a task to a CPU thread. For instance, a backward propagation task in OpenTimer [31] typically finishes within 0.5-50 us, yet scheduling this task using OpenTimer's Taskflow [40] scheduler takes 0.2-3 us [78]. Therefore, finding the right balance between the scheduling overhead and TDG granularity is crucial for improving the performance of task-parallel STA applications.

To address this issue, Zhang et al. recently introduced G-PASTA [78], a GPU-accelerated TDG partitioning algorithm that clusters tasks to reduce the TDG size. Clustered tasks within one partition run sequentially based on their topological order in the original TDG. This approach reduces the scheduling overhead because it eliminates the need to schedule tasks individually across multiple threads. Instead, an entire partition is scheduled once and executed by a single thread. Compared to other sequential TDG partitioning algorithms such as Vivek [72] and GDCA [72], G-PASTA [78] achieves fast partitioning runtime by leveraging the power of GPU to efficiently partition the TDG. Although G-PASTA [78] improves the partitioning performance, it is restricted to *full partitioning* and does not support *incremental partitioning*. However, practical STA tools require incrementality to work with timing-driven optimization [3]. Specifically, when a TDG is incrementally changed, G-PASTA [78] needs to repartition the entire TDG from scratch even when only a small fraction of the TDG needs to be repartitioned. For example, in Figure 1 (a), a buffer is inserted into the circuit by a timing-driven operation, which is represented as task and edge insertions in the corresponding TDG shown in Figure 1 (b). Since the TDG has been partitioned into $P1$ and $P2$ in the previous iteration, the inserted tasks and edges introduce a *cycle* between these two partitions, which is illegal because we cannot schedule a TDG with cycles. In this case, G-PASTA [78] has to repartition the entire TDG in (b) to remove the cycle. In practice, timing optimization may incur millions of incremental iterations. Without incremental partitioning, we cannot fully harness the power of TDG partitioning in task-parallel STA applications.

However, designing an efficient incremental TDG partitioning algorithm is challenging for the following three reasons: (1) We need to effectively identify a minimal set of tasks necessary for initiating incremental partitioning. (2) During incremental partitioning, we need to avoid traversing too many tasks to update partitions. Otherwise, it has no difference from full partitioning. (3) Incremental partitioning must resolve any potential cycles introduced by incremental operations. To overcome these challenges, we propose iTAP, an incremental TDG partitioner for task-parallel STA. Compared to existing TDG partitioners, iTAP can incrementally update partitions in response to changes in the TDG topology. We summarize iTAP's technical contributions as follows:

- We propose a strategy to effectively identify a minimal set of tasks necessary for initiating incremental partitioning.
- We design an effective incremental clustering algorithm that completes incremental partitioning by traversing a minimal number of tasks without introducing any cycles.
- We give rigorous proof to verify the correctness of our incremental cycle-free clustering algorithm.

We evaluate the performance of iTAP on a set of TDGs and incremental operations obtained from OpenTimer [31] and 2015 TAU Contest [26]. Compared to G-PASTA [78], iTAP enhances the overall STA performance by up to 2.97×. We plan to open the source of iTAP to benefit the STA community.

2 BACKGROUND

2.1 TDG Partitioning

The inputs of TDG partitioning include: (1) a TDG where each task corresponds to a specific timing propagation task (e.g., delay calculation) and each edge corresponds to the dependency between two tasks, and (2) an adjustable partition size parameter to limit the maximum number of tasks within a partition. The objective is to partition the TDG to an optimal granularity without introducing any cycles so that the partitioned TDG can deliver better runtime performance than the original TDG. TDG partitioning stands apart from conventional graph or hypergraph partitioners such as Metis [53] and Kahypar [73] by targeting partitioning a *directed acyclic graph* (DAG) to minimize scheduling overhead while maintaining the original TDG parallelism. Besides, since partitioning a TDG is conceptually identical to clustering the tasks in a TDG, in this paper, the two terms, partitioning and clustering, are interchangeable.

2.2 G-PASTA

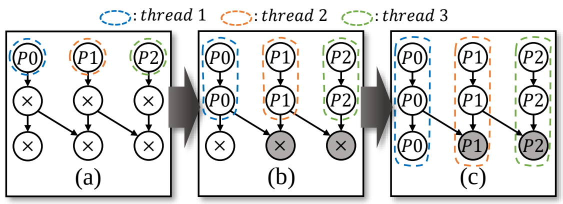

Figure 2: An example of G-PASTA's [78] partitioning process in three iterations. Threads 1, 2, and 3 traverse the TDG by parallel BFS and assign partition IDs to the tasks. Under G-PASTA's [78] constraint, each task with multiple dependents (marked in grey) is clustered into its dependent's partition with the largest partition ID (e.g., the task with dependents from $P0$ and $P1$ is clustered into $P1$).

iTAP is built on top of G-PASTA [78] because of its fast partitioning runtime and high partitioning quality. In this section, we provide a brief overview of G-PASTA [78]. By leveraging the power of GPU, G-PASTA [78] efficiently partitions the TDG by assigning partition IDs to the tasks using parallel breadth-first search (BFS). Figure 2 illustrates G-PASTA's [78] partitioning process in three iterations. In the first iteration (a), each thread assigns a source task with a new partition ID ($P0$, $P1$, or $P2$). In the following two iterations (b) and (c), each thread traverses to the next BFS level and assigns a partition ID to each available task.

However, such parallel partitioning can easily introduce cycles to the partitioned TDG. To address this, G-PASTA [78] introduces a key constraint on tasks' partition IDs, which is, *the partition ID of a task must be no smaller than the partition IDs of its dependents.*

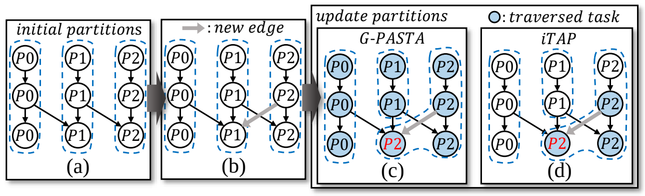

Figure 3: An example that inserting an edge into a partitioned TDG results in a cycle, thus requiring repartitioning the TDG. (a) Initial partitions. (b) A new edge is inserted, leading to a cycle between $P1$ and $P2$. To remove this cycle, (c) G-PASTA [78] traverses the entire TDG to repartition it, while (d) iTAP only traverses three tasks to update partitions.

This constraint, as proven in [78], effectively eliminates cycles by preventing backward dependencies during partitioning. Based on this constraint, in Figure 2 (b) and (c), each task with multiple dependents (marked in grey) is clustered into its dependent's partition with the largest partition ID (e.g., the task with dependents from $P0$ and $P1$ is clustered into $P1$).

Despite the fast partitioning runtime offered by GPU parallelism, G-PASTA [78] lacks the support for incrementality-*it cannot efficiently update partitions without repartitioning the entire TDG when the topology changes.* This limitation hinders G-PASTA [78] from fully utilizing its fast partitioning capabilities in incremental timing analysis, where the TDG topology is frequently modified, as shown in the example in Figure 3.

3 iTAP

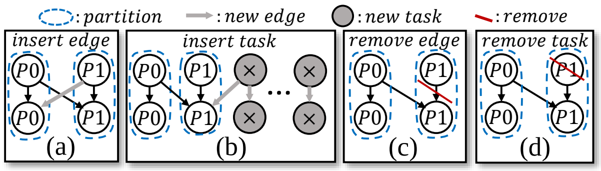

Figure 4: Four types of incremental operations on a TDG.

To achieve incremental partitioning, we first examine four types of incremental operations that can be applied to a previously partitioned TDG: *inserting edges*, *inserting tasks*, *removing edges*, and *removing tasks*, as shown in Figure 4. We apply incremental partitioning when inserting edges and inserting tasks because (1) inserting edges between partitioned tasks can introduce illegal cycles (see Figure 4 (a)), and (2) continuously inserting tasks over many incremental iterations without partitioning them can lead to too many tasks in the TDG (see Figure 4 (b)), thus increasing the scheduling overhead. However, removing edges (Figure 4 (c)) and removing tasks (Figure 4 (d)) do not necessitate incremental partitioning, as they neither introduce cycles nor increase the scheduling overhead. To eliminate the potential cycles caused by edge insertions between partitioned tasks, we identify all the partitioned tasks affected by edge insertions and define them as *type-1 frontiers.* On the other hand, to mitigate the increased scheduling overhead from task insertions, we identify all the newly inserted tasks and their connected partitioned tasks and define them as *type-2 frontiers.*

In this paper, iTAP focuses on resolving potential cycles and mitigating the increased scheduling overhead, which is caused by edge and task insertions, by partitioning these two types of frontiers. As shown in Algorithm 1, iTAP has two stages: (1) *identify a small set of tasks from type-1 and type-2 frontiers, defined as initial frontiers, to initiate incremental partitioning* (lines 1-2), and (2) *apply our incremental cycle-free clustering algorithm on the frontiers* (lines 3-4).

```
Algorithm 1: Overview of iTAP
  Input: list of partitioned tasks T, list of newly inserted
         edges E_n, list of newly inserted tasks T_n
  Output: T with partition ID par_id assigned to each task
1 type1_ftr_queue ← identify_initial_type1_frontiers(T, E_n);
2 type2_ftr_queue ← identify_initial_type2_frontiers(T, T_n);
3 T ← handle_type1_frontiers(type1_ftr_queue);
4 T ← handle_type2_frontiers(type2_ftr_queue);
```

3.1 Identify Initial Frontiers

To initiate incremental partitioning, we need to identify a set of initial frontiers from type-1 and type-2 frontiers separately as the starting point for our incremental cycle-free clustering algorithm.

```
Algorithm 2: identify_initial_type1_frontiers(T, E_n)
  Input: T, E_n
  Output: type1_ftr_queue filled with initial type-1 frontiers
1 foreach e ∈ E_n
2     if e.from_task.partitioned & e.to_task.partitioned
3         type1_ftr_queue.push(e.from_task);
```

(1) *Identify initial type-1 frontiers.* Algorithm 2 shows that when an edge is inserted between two partitioned tasks, the source task of the edge is considered as an initial type-1 frontier as it is the starting point where a cycle may be introduced. As shown in Figure 5 (a), a cycle is introduced between $P0$ and $P1$ when an edge is inserted between tasks 1 and 2. In this case, task 1 is an initial type-1 frontier and pushed into type-1 frontier queue (*type1_ftr_queue*).

```
Algorithm 3: identify_initial_type2_frontiers(T, T_n)
  Input: T, T_n
  Output: type2_ftr_queue filled with initial type-2 frontiers
1 foreach t ∈ T_n
2     if t.fanin.size = 0
3         type2_ftr_queue.push(t);
4     else if all t.dependents are partitioned
5         type2_ftr_queue.push(t);
```

(2) *Identify initial type-2 frontiers.* Algorithm 3 shows that the inserted (unpartitioned) tasks that have no dependents (lines 2-3) along with those that have only partitioned dependents (lines 4-5) are initial type-2 frontiers. As illustrated in Figure 5 (b), among unpartitioned tasks, tasks 5 and 8 have no dependents, and task 7 has only partitioned dependents, so they are initial type-2 frontiers and pushed into type-2 frontier queue (*type2_ftr_queue*).

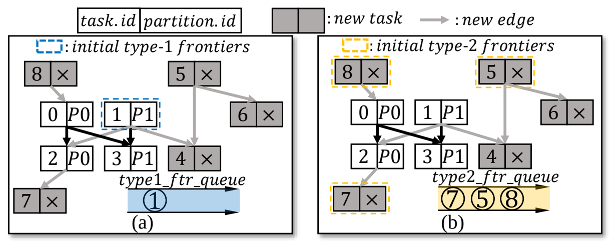

Figure 5: An example of initial type-1 frontiers (a) and initial type-2 frontiers (b). (a) The source task of a new edge (task 1) is an initial type-1 frontier (marked in blue). (b) The new (unpartitioned) tasks that have no dependents (tasks 5 and 8) along with those that have only partitioned dependents (task 7) are initial type-2 frontiers (marked in yellow).

3.2 Incremental Cycle-free Clustering Algorithm

Once we have identified these two types of initial frontiers, we can repartition the TDG using our incremental cycle-free clustering algorithm. Here, we have two challenges: (1) We need to minimize the number of traversed tasks during incremental partitioning. (2) Incremental partitioning must resolve any potential cycles introduced by incremental operations. To avoid cycles, iTAP adopts G-PASTA's [78] cycle-free constraint: *the partition ID of a task must be no smaller than the partition IDs of its dependents*. However, maintaining the cycle-free constraint in incremental partitioning is nontrivial because incremental operations are randomly applied to the TDG and we do not know which ones of these operations can introduce cycles. To overcome these two challenges, our algorithm operates in two phases: (1) *handle type-1 frontiers*, and (2) *handle type-2 frontiers*.

(1) *Handle type-1 frontiers*. The goal of this phase is to eliminate the potential cycles due to inserted edges between the partitioned tasks. This is done by updating the partition IDs of type-1 frontiers to meet the cycle-free constraint. As presented in Algorithm 4, with initial type-1 frontiers stored in *type1_ftr_queue* by Algorithm 2, we initiate a BFS starting from initial type-1 frontiers. Every time we pop a frontier from *type1_ftr_queue*, we examine its partitioned successors. If a successor's partition ID is smaller than that of the frontier, we update the partition ID of this successor with the frontier's partition ID (lines 5-6). Finally, we push this successor into *type1_ftr_queue* as it will be the type-1 frontier in the next incremental partitioning iteration. Figure 6 (a)-(b) illustrate phase 1 of our algorithm in two iterations.

```
Algorithm 4: handle_type1_frontiers(type1_ftr_queue)
  Input: type1_ftr_queue filled with initial type-1 frontiers
1 while !type1_ftr_queue.empty()
2     cur_task ← type1_ftr_queue.pop();
3     foreach suc ∈ cur_task.fanouts
4         if suc.partitioned & suc.par_id < cur_task.par_id
5             suc.par_id ← cur_task.par_id;
6             type1_ftr_queue.push(suc);
```

(2) *Handle type-2 frontiers*. The goal of this phase is to partition the newly inserted tasks to mitigate the increased scheduling overhead caused by task insertions without introducing any cycles. This is done by clustering type-2 frontiers under the cycle-free constraint. As shown in Algorithm 5, with initial type-2 frontiers stored in *type2_ftr_queue* by Algorithm 3, we first cluster a frontier popped from the queue in the current partitioning iteration, then traverse its successors by topological sorting to identify the next frontiers for the next partitioning iteration.

```
Algorithm 5: handle_type2_frontiers(type2_ftr_queue)
  Input: type2_ftr_queue filled with initial type-2 frontiers
1 while !type2_ftr_queue.empty()
2     cur_task ← type2_ftr_queue.pop();
3     cluster_current_frontiers(cur_task);
4     identify_next_frontiers(cur_task);
```

Algorithm 6 shows how we cluster type-2 frontiers. To ensure the partition IDs of these frontiers satisfy the cycle-free constraint, we first obtain the largest partition ID from its partitioned dependents (*Lpd*) and the smallest partition ID from its partitioned successors (*Sps*). To maintain the cycle-free constraint, the partition ID of this frontier should be no smaller than *Lpd* and no larger than *Sps*. We then categorize type-2 frontiers into the following four cases and cluster them accordingly:

```
Algorithm 6: cluster_current_frontiers(cur_task)
  Input: type-2 frontier cur_task in the current partitioning
         iteration
  Global: partition size P_s, current maximum partition ID
          max_par_id
1  [Lpd, Sps] ← get_Lpd_and_Sps(cur_task);
2  if Lpd does not exist & Sps exists // case Ⓐ
3      cur_task.par_id ← Sps;
4  else if Lpd does not exist & Sps does not exists // case Ⓑ
5      cur_task.par_id ← ++max_par_id;
6  else if Lpd exists & Sps does not exist // case Ⓒ
7      if number of tasks in partition Lpd < P_s
8          cur_task.par_id ← Lpd;
9      else
10         cur_task.par_id ← ++max_par_id;
11 else if Lpd exists & Sps exists // case Ⓓ
12     cur_task.par_id ← Lpd;
13 mark cur_task as partitioned;
```

Ⓐ Type-2 frontier does not have partitioned dependents but has partitioned successors, i.e., *Lpd* does not exist but *Sps* exists (lines 2-3). In this case, we cluster this frontier into partition *Sps* since its partition ID should be no larger than *Sps*.

Ⓑ Type-2 frontier has neither partitioned dependents nor partitioned successors, i.e., Neither *Lpd* nor *Sps* exists (lines 4-5). In this case, we assign a new partition ID to this frontier since its partition ID is not constrained by *Lpd* or *Sps*.

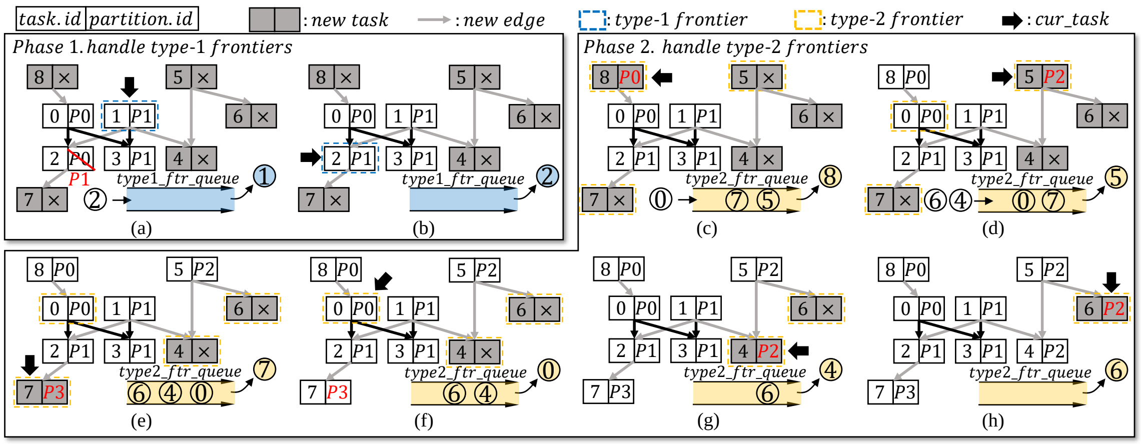

Figure 6: An example of incremental cycle-free clustering algorithm in eight iterations under the partition size of 3. Phases 1 and 2 of our algorithm are shown in (a)-(b) and (c)-(h) respectively. The initial frontiers in the queues are obtained by Algorithm 2 and Algorithm 3 from Figure 5. In phase 1, (a) we pop task 1 and traverse its partitioned successors, tasks 2 and 3. Since task 2's partition ID ($P0$) is smaller than task 1's ($P1$), we update task 2's partition ID as $P1$ and push it into $type1\_ftr\_queue$. (b) We pop task 2 and since it has no partitioned successors, $type1\_ftr\_queue$ is empty. So phase 1 ends. In phase 2, (c) we pop task 8, which falls under case Ⓐ, so we cluster task 8 into $P0$. Then we traverse task 8's successors by topological sorting and push task 0 into $type2\_ftr\_queue$. (d) We pop task 5, which falls under case Ⓑ, so we assign it with a new partition $P2$. Then we push task 5's successors, tasks 4 and 6 into $type2\_ftr\_queue$. (e) We pop task 7, which falls under case Ⓒ. Since $P1$ already has three tasks, we assign task 7 with a new partition $P3$. (f) We pop task 0, which falls under case Ⓓ, so it stays in $P0$. We do not identify any of task 0's successors as a frontier since they all satisfy the cycle-free constraint. Finally, in (g)-(h), we pop tasks 4 and 6 in turns. They both fall under case Ⓒ. Since $P2$ still has space, we cluster them into $P2$. Since $type2\_ftr\_queue$ is empty, phase 2 ends.

Ⓒ Type-2 frontier has partitioned dependents but does not have partitioned successors, i.e., *Lpd* exists but *Sps* does not exist (lines 6-10). In this case, since the partition ID of this frontier should be no smaller than *Lpd*, we cluster it into partition *Lpd* only if there is still space in partition *Lpd* (lines 7-8). Otherwise, we assign it with a new partition ID (lines 9-10).

Ⓓ Type-2 frontier has both partitioned dependents and partitioned successors, i.e., both *Lpd* and *Sps* exist (lines 11-12). In this case, we cluster this frontier into partition *Lpd* since it is possible that *Lpd* > *Sps*, leaving us no choice but to cluster this frontier into partition *Lpd* and update its successors in later partitioning iterations to preserve the cycle-free constraint.

After we cluster a type-2 frontier in the current partitioning iteration, we traverse its successors by topological sorting to identify the frontiers for the next partitioning iteration. As presented in Algorithm 7, we apply a topological sorting on the successors of a type-2 frontier in the current iteration by using a dependency counter *dep_cnt* to count the visited fanin dependencies of its successors. Once *dep_cnt* of a successor equals the size of its fanin, this successor is identified as the frontier in the next iteration and pushed into *type2_ftr_queue*. Additionally, if a successor is partitioned and its partition ID is smaller than that of the current frontier (lines 6-7), we also push it into *type2_ftr_queue* to update its partition ID in the subsequent iterations.

Figure 6 (c)-(h) shows phase 2 of our algorithm under the partition size of 3 in six iterations. During each iteration, we pop a type-2 frontier from *type2_frt_queue*, decide which of the above four cases it can be applied to (Algorithm 6), and cluster this frontier

Algorithm 7: identify_next_frontiers(*cur_task*)

```
Input: type-2 frontier cur_task in the current partitioning
       iteration
1 foreach suc ∈ cur_task.fanout
2     update suc.dep_cnt;
3     if suc.dep_cnt = suc.num_fanin_dependents
4         type2_ftr_queue.push(suc);
5         continue;
6     if suc.partitioned & suc.par_id < cur_task.par_id
7         type2_ftr_queue.push(suc);
```

accordingly. Next, we traverse its successors and identify which successors will be the next frontiers by Algorithm 7. These identified successors are then pushed into *type2_ftr_queue* for the subsequent partitioning iterations.

Our incremental clustering algorithm resolves the potential cycles due to incremental operations by maintaining the cycle-free constraint. More importantly, our algorithm largely reduces the number of tasks we need to traverse to complete the partitioning, thus providing faster partitioning runtime than G-PASTA [78].

3.3 Proof

In this section, we give rigorous proof that our incremental cycle-free algorithm does not introduce any cycles to the partitioned TDG by showing that it satisfies the cycle-free constraint: *the partition ID of a task must be no smaller than the partition IDs of its dependents.*

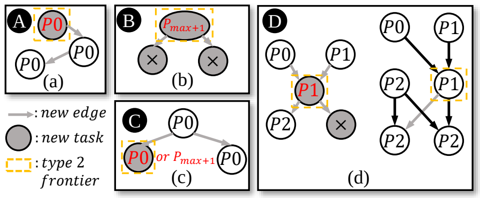

Figure 7: Illustrations of A, B, C, and D in Algorithm 6. *max* refers to *max_par_id*, which is the current maximum partition ID of the TDG.

Since our algorithm has two phases (Algorithm 4 and Algorithm 5), we prove them separately:

Theorem 1. *Phase 1 of our algorithm (Algorithm 4) does not introduce any cycles during the partition process.*

Proof. In algorithm 4, we identify the source task of an inserted edge between partitioned tasks as a frontier. From this frontier, we initiate a BFS to traverse its successors in the subsequent BFS levels to maintain the cycle-free constraint. This operation is applied for every inserted edge between partitioned tasks sequentially. Therefore, the cycle-free constraint is preserved in phase 1, thus no cycles will be introduced. □

For ease of proving the correctness of phase 2 of our algorithm (Algorithm 5), given a TDG G and a type-2 frontier $ftr_i \in G$, we define the set of dependents of ftr_i as $D(i)$, the set of its partitioned dependents as $PD(i)$, and the set of its partitioned successors as $PS(i)$. We also define the partition ID of ftr_i as $par\_id(i)$ and the current maximum partition ID as max_par_id. We then propose:

Theorem 2. *Phase 2 of our algorithm (Algorithm 5) does not introduce any cycles during the partitioning process.*

Proof. Algorithm 5 has two steps: (1) cluster current frontiers (Algorithm 6), and (2) identify next frontiers (Algorithm 7). We first prove the correctness of Algorithm 6 by examining the four cases below:

A $PD(i) = \emptyset$ and $PS(i) \neq \emptyset$, $par\_id(i) = Sps$, as visualized in Figure 7 (a). Since Algorithm 5 operates by topological sorting, all dependents of all ftr_i must be partitioned, i.e., $\forall ftr_i \in G, D(i) = PD(i)$. So $D(i) = \emptyset$. Therefore, regardless of the value of $par\_id(i)$, the cycle-free constraint is satisfied.

B $PD(i) = \emptyset$ and $PS(i) = \emptyset$, $par\_id(i) = max\_par\_id + 1$, as visualized in Figure 7 (b). Similar to case A, the cycle-free constraint is satisfied.

C $PD(i) \neq \emptyset$ and $PS(i) = \emptyset$, $par\_id = Lpd$ or $max\_par\_id + 1$, as visualized in Figure 7 (c). Since $D(i) = PD(i)$, here Lpd is the largest partition ID of all dependents. Therefore, the cycle-free constraint is satisfied.

D $PD(i) \neq \emptyset$ and $PS(i) \neq \emptyset$, $par\_id = Lpd$, as visualized in Figure 7 (d). Similar to case C, the cycle-free constraint is satisfied.

After clustering the current frontiers based on these four cases, Algorithm 5 examines their successors by topological sorting. If the partition IDs of these successors need to be updated, they will be the frontiers in the next partitioning iteration (Algorithm 7), which again, will fall under the above four cases. Therefore, we prove Theorem 2 is correct. With Theorem 1 and Theorem 2, we prove the correctness of our incremental cycle-free clustering algorithm. □

4 EXPERIMENTAL RESULTS

We implemented iTAP in C++ and compiled it using g++ v11.4 with `-O2` and `-std=c++17` enabled. We performed experiments on a 4.8 GHz 64-bit Linux machine equipped with an Intel Core i5-13500 CPU and an Nvidia RTX A4000 GPU. We consider G-PASTA [78] as our baseline due to its fast partitioning runtime and high partitioning quality compared with other existing TDG partitioners [1, 72]. We implemented G-PASTA [78] in CUDA and compiled it using nvcc v12.2 with `-O2` enabled.

To conduct the experiments, we perform one initial full timing update and 1K subsequent incremental timing updates on six industrial circuits. We generate the TDGs of these circuits using OpenTimer [31], where each task performs a timing propagation. In the initial full-timing update, we use G-PASTA [78] to partition the TDGs. For each subsequent incremental timing update, we apply a mix of 40 random incremental operations, including removing edges, removing tasks, inserting edges, and inserting tasks based on the setting of 2015 TAU Contest [26]. We then partition the TDGs with G-PASTA [78] and iTAP independently. Since iTAP requires users to specify a partition size, we use the value that yields optimal performance in G-PASTA [78]. The statistics of the circuit TDGs and their partitioning results are presented in Table 2. All data is an average of 10 runs.

4.1 Partitioning Performance Comparison

Table 1 compares the overall partitioning performance per incremental timing update iteration between G-PASTA [78] and iTAP with respect to the partitioning runtime as well as their improvement on building time and runtime of partitioned TDGs. The entries under T_{Build}, T_{Run}, and T_{Total} show TDG building time, TDG runtime, and their aggregate total time before partitioning per incremental timing update. The entries under T_{Build^P} and T_{Run^P} show TDG building time and TDG runtime after partitioning per incremental timing update. The entries under the $T_{Partition}$ show the partitioning runtime of G-PASTA [78] and iTAP. The entries under T_{Total^P} show the aggregate total time including T_{Build^P}, T_{Run^P}, and $T_{Partition}$ after partitioning, along with the speedup over T_{Total}.

Overall, for all circuits, G-PASTA [78] and iTAP both enhance the STA performance by reducing the TDG size and scheduling overhead. Nonetheless, iTAP outperforms G-PASTA [78] across all circuit benchmarks. For instance, iTAP improves the STA performance by 1.86-2.97×, whereas G-PASTA [78] is 1.11-1.98×. Regarding the partitioning runtime, iTAP is significantly faster than G-PASTA [78] because iTAP supports incrementality, i.e., incrementally updating partitions instead of repartitioning the entire TDG. For example, in the largest circuit, leon2, the speedup of iTAP over G-PASTA [78] on the partitioning runtime is 29.80×.

Figure 8 visualizes the partitioning performance by breaking down the runtime for one incremental timing update iteration. We can clearly see that iTAP further reduces STA runtime than G-PASTA [78] due to faster partitioning runtime. We do observe that as the circuit size grows larger, for one incremental timing

Table 1: Comparison of overall performance per incremental timing update iteration between G-PASTA [78] and iTAP, along with their improvements on TDGs generated from circuit benchmarks in OpenTimer [31].

circuit	T_{Build} (ms)	T_{Run} (ms)	T_{Total} (ms)	T_{Build^P} (ms)		T_{Run^P} (ms)		$T_{Partition}$ (ms)		T_{Total^P} (ms)	
				G-PASTA	iTAP	G-PASTA	iTAP	G-PASTA	iTAP	G-PASTA	iTAP
wb_dma	0.79	1.22	2.01	0.06	0.06	0.87	0.77	0.87	0.03	1.8 (1.11×)	0.86 (2.33×)
ac97_ctrl	3.59	3.45	7.04	0.65	0.61	1.95	1.57	1.66	0.82	4.26 (1.65×)	3.00 (2.34×)
aes_core	5.91	5.34	11.25	1.49	1.43	2.37	2.01	2.28	0.34	6.14 (1.83×)	3.78 (2.97×)
des_perf	25.08	25.16	50.24	11.21	11.40	10.30	9.90	3.76	0.56	25.27 (1.98×)	21.86 (2.29×)
vga_lcd	32.51	32.93	65.44	17.05	16.66	14.34	13.7	4.07	0.58	35.46 (1.84×)	30.94 (2.11×)
leon2	335.27	350.51	685.78	219.13	223.20	144.44	142.96	24.74	0.83	388.31 (1.76×)	366.99 (1.86×)

T_{Build}, T_{Run}: TDG building time and runtime before partitioning $T_{Total} = T_{Build} + T_{Run}$

T_{Build^P}, T_{Run^P}: TDG building time and runtime after partitioning $T_{Partition}$: partitioning runtime $T_{Total^P} = T_{Build^P} + T_{Run^P} + T_{Partition}$

Table 2: The number of tasks and dependencies in the original TDGs generated by OpenTimer [31] and the partitioning results by G-PASTA [78] and iTAP. The reduction ratio (×) is with respect to the original TDG.

circuit	Origin		G-PASTA		iTAP	
	#tasks	#deps	#tasks	#deps	#tasks	#deps
wb_dma	13.1K	33.2K	6.4K (2.0×)	8.9K (3.7×)	3.3K (3.9×)	10.1K (3.2×)
ac97_ctrl	42.4K	107.1K	15.1K (2.8×)	38.9K (2.7×)	11.7K (3.6×)	42.2K (2.5×)
aes_core	66.8K	86.4K	17.6K (3.7×)	59.7K (1.4×)	13.8K (4.8×)	63.6K (1.3×)
des_perf	303.7K	387.3K	67.1K (4.5×)	285.1K (1.3×)	62.4K (4.8×)	288.2K (1.3×)
vga_lcd	397.8K	498.9K	87.7K (4.5×)	361.3K (1.3×)	83.9K (4.7×)	366.9K (1.3×)
leon2	4.3M	5.3M	750.0K (5.7×)	3.4M (1.5×)	750.2K (5.7×)	3.4M (1.5×)

update iteration, the partitioning runtime improvement by iTAP does not significantly reduce the overall STA runtime. However, practical timing-driven optimization often involves millions of iterations [3]. For a circuit such as leon2, reducing 20 milliseconds per iteration can accumulate to several hours of runtime reduction over millions of iterations. As we shall show in Figure 9, this per-iteration improvement will contribute significantly to the overall STA performance.

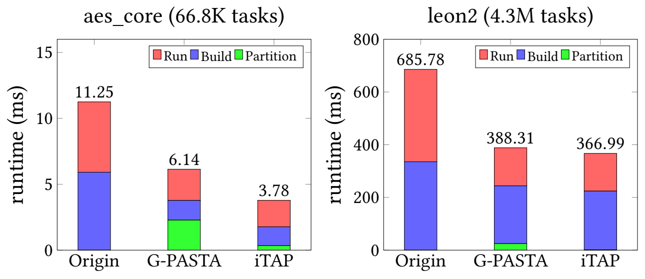

Figure 8: Runtime breakdown of one incremental timing update iteration including partitioning, building and running the TDG. Origin indicates the STA runtime without any partitioning.

4.2 STA Runtime over Incremental Iterations

Figure 9 compares the overall STA runtime of G-PASTA [78] and iTAP over 1K incremental timing update iterations, where the two partitioners are invoked separately to partition the modified TDG.

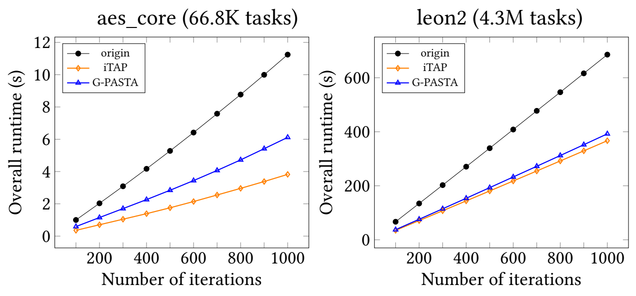

Figure 9: Comparison of STA runtime improvement between G-PASTA [78] and iTAP over 1K incremental timing iterations. The black lines represents the original runtime without any partitioning.

The overall STA runtime comprises the time spent on TDG partitioning as well as building and running the partitioned TDG. The black lines indicate the original STA runtime without any partitioning. We can see from Figure 9 that compared to G-PASTA [78], iTAP further improves the performance of STA over incremental timing iterations due to its capability of incremental partitioning. The performance advantage of iTAP over G-PASTA [78] continues to increase over iterations.

4.3 Partitioning Runtime Over Incremental Iterations

Figure 10 compares the partitioning runtime of G-PASTA [78] and iTAP over 1K incremental timing update iterations. In general, iTAP

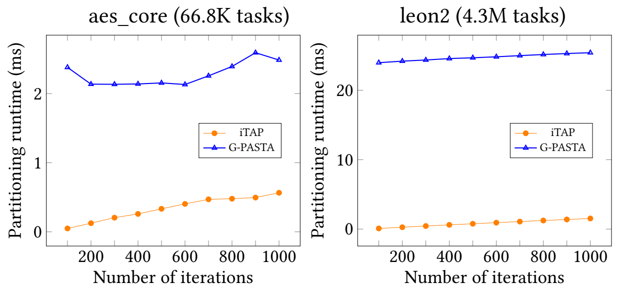

Figure 10: Partitioning runtime comparison between iTAP and G-PASTA [78] over 1K incremental timing update iterations.

significantly outperforms G-PASTA [78] in terms of the partitioning runtime, demonstrating a substantial advantage regardless of the circuit size. We do observe that over iterations, the partitioning runtime of iTAP increases slightly, and that of G-PASTA [78] remains stable. Still, such fluctuations cannot bridge the significant speedup gap that iTAP holds over G-PASTA [78], which is brought by incrementally updating the partitions without traversing the entire TDG.

5 CONCLUSION

In this paper, we have proposed iTAP, an incremental TDG partitioner for task-parallel static timing analysis (STA). iTAP introduces an efficient incremental partitioning algorithm that completes incremental partitioning by traversing a minimal amount of tasks without introducing any cycles. Compared to a state-of-the-art full TDG partitioner, iTAP enhances the overall STA performance by up to 2.97×. Inspired by the success of GPU computing in graph processing [2–19, 21–25, 27–34, 36–39, 41–52, 54–71, 74, 76, 77, 79, 80], we plan to extend iTAP to GPU task graph applications.

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] Bérenger Bramas and Alain Ketterlin. 2020. Improving parallel executions by increasing task granularity in task-based runtime systems using acyclic DAG clustering. *PeerJ Computer Science* 6 (2020), e247.
[2] 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).*
[3] 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.*
[4] 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).*
[5] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC).*
[6] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM ICPP.* 565–575.
[7] 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).*
[8] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC).*
[9] 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).*
[10] 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).*
[11] 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).*
[12] 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).*
[13] 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).*
[14] 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).*
[15] 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).
[16] 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).*
[17] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC).*
[18] 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).*
[19] 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).*
[20] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. Gpu-accelerated static timing analysis. In *Proceedings of the 39th international conference on computer-aided design.* 1–9.
[21] 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).*
[22] 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).*
[23] 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).
[24] 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).*
[25] 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).*
[26] 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).* IEEE, 882–889.
[27] 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).*
[28] 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).*
[29] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC).*
[30] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS).*
[31] 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).

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

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

[34] 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)*.

[35] 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)*.

[36] 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)*.

[37] 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)*.

[38] 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).

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

[40] 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).

[41] 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).

[42] 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)*.

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

[44] 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).

[45] 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)*.

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

[47] 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.

[48] 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*.

[49] 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)*.

[50] 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)*.

[51] 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)*.

[52] 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.

[53] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1997. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th annual Design Automation Conference*. 526–529.

[54] 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)*.

[55] 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)*.

[56] 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)*.

[57] 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)*.

[58] 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)*.

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

[60] 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)*.

[61] 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)*.

[62] 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)*.

[63] 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)*.

[64] 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)*.

[65] 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).

[66] 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)*.

[67] 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)*.

[68] 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)*.

[69] 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*.

[70] 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)*.

[71] 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)*.

[72] Vivek Sarkar and John Hennessy. 1986. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. 202–211.

[73] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2023. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithmics* 27 (2023), 1–39.

[74] 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)*.

[75] Yu-Ming Yang, Yu-Wei Chang, and Iris Hui-Ru Jiang. 2014. iTimerC: Common path pessimism removal using effective reduction methods. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 600–605.

[76] 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.

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

[78] Boyang Zhang, Lin Dian-Lun, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Lee Wan Luan, Chang Chih-Chun, Fang Donghao, and Huang Tsung-Wei. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *DAC*. 6 pages. https://doi.org/10.1145/3649329.3656230

[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)*.