An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads

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

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

***Abstract*—Many circuit analysis workloads incorporate complex execution logic under dynamic control flow, such as branch-and-bound techniques, on-the-fly pruning and recursive decomposition strategies. Parallelizing these kinds of workloads can benefit from the exploitation of dynamic task graph parallelism across arbitrary decision-making points at runtime. A recent research paper AsyncTask has introduced a new programming model that supports the dynamic construction of a computational task graph. Unlike the traditional construct-and-run programming models, AsyncTask offers programmers great flexibility to parallelize large-scale circuit analysis workloads that are extremely spare, irregular and control-flow intensive. To leverage the power of dynamic task parallelism, AsyncTask users are responsible for creating tasks in a valid topological order. This paper conducts an experimental study to investigate in the runtime difference of different topological orders of tasks on large-scale static timing analysis workloads using AsyncTask. Our result highlights the need for a new technique to get a valid topological sequence that yields a better runtime performance than heuristic-based sorting algorithms for large-scale real-world circuit analysis applications.**

***Index Terms*—Dynamic Task Graph Parallelism, Task Graph, Topological Order**

I. Introduction

Task graph programming (TGP) has inspired many new parallel and heterogeneous circuit analysis algorithms [6]–[8], [10]–[19], [22]–[25], [27], [28], [32], [36], [41], [47], [50], [54]–[61] and large-scale machine learning problems [9], [42]–[45], [51], [52]. Unlike traditional loop-based models that explore parallelism across parallel loops, TGP models a function call as a *task* and a functional dependency as an *edge* in a *task graph*. The left plot in Figure 1 illustrates an example task graph with four tasks and four edges (or dependencies). TGP models enable applications to perform top-down optimization in irregular parallel decomposition strategies that consist of many tasks and dependencies. Then, a TGP runtime is able to scale these dependent tasks across a large number of processors with dynamic load balancing [49]. As a result, the parallel computing community has introduced many successful TGP libraries in various application domains, such as OpenMP [1], Kokkos-DAG [3], PaRSEC [5], Taskflow [20], [21], [31], [33], [48], and Taro [53].

Typically, TGP has two forms: *static task graph programming* (STGP) and *dynamic task graph programming* (DTGP). For the STGP users, they define the task graph in advance and submit it to a STGP runtime for execution. The top right plot in Figure 1 illustrates the timing diagram of STGP. Since the graph structure is defined a priori, the STGP runtime is capable of performing whole-graph optimization. On the other hand, DTGP users define the task graph structure dynamically. Tasks and dependencies are created on the fly according to the runtime variables and control-flow results. As a result, DTGP allows the task creation time to overlap with the task execution time, as shown in the bottom right plot in Figure 1. Thus, DTGP is often more flexible than STGP when dealing with many circuit analysis algorithms that frequently incorporate dynamic control flow to implement irregular parallel decomposition strategies.

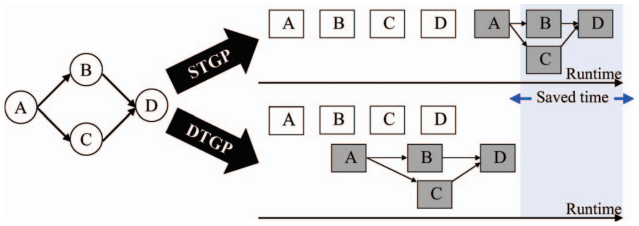

Fig. 1: An illustration of the execution diagram of a task graph. The left plot denotes a task graph. The right top plot denotes the execution diagram of STGP while the right below denotes DTGP. White rectangles represent the task creation and gray rectangles represent the task execution. Edges denote the dependencies. Task are created in the order of A-B-C-D in both STGP and DTGP. The blue area highlights the saved time of DTGP over STGP.

Recently, a research paper has introduced a new DTGP library called *AsyncTask* [10] to assist circuit analysis developers in quickly exploring dynamic task graph parallelism. As AsyncTask has demonstrated a promising runtime performance in large-scale timing analysis workloads, it is now integrated into the popular Taskflow library [31] and Taskflow claims the number of AsyncTask users is increasing. Among the special properties of DTGP, we find one interesting topic that could directly impact the performance of AsyncTask applications and that is, different topological orders for creating the tasks could

DOI 10.1109/ISVLSI61997.2024.00149

lead to different runtime performance. For example, in Figure 1, the creation of tasks is illustrated in the topological order of A-B-C-D. However, the other valid order A-C-B-D could achieve a better runtime performance than the order A-B-C-D in certain applications, such as multiple tasks relying on the runtime result of task C.

Consequently, in this paper we conduct an experimental study to investigate in the runtime performance effect using different topological orders, and highlight the need for a new technique to schedule tasks with valid topological orders for better runtime performance. By conducting the study, users can further optimize their applications when using AsyncTask to exploit dynamic task graph parallelism. As AsyncTask is inspired to deal with electronic design automation (EDA) applications, this paper will focus on a large-scale circuit analysis problem that is representative of many analysis-driven EDA applications.

II. Overview of AsyncTask

AsyncTask [10] is a new dynamic task graph programming (DTGP) library. It enables an efficient implementation of irregular parallel decomposition strategies through a top-down dynamic task graph. It provides an expressive programming model for applications to describe the task graphs easily, and introduces a new task scheduling algorithm to support its programming model with only atomic counters to reduce the overhead of managing the dependencies between tasks.

A. Programming Model

To enable expressive DTGP, AsyncTask directly specifies a task's dependencies with its dependent tasks in a clear graph description language, which allows programmers to easily leverage the power of DTGP. Listing 1 demonstrates the AsyncTask implementation for the task graph in Figure 1. The program creates a task graph of four tasks, A, B, C, and D. The dependency constraints state that task B and task C run after task A, and task D runs after both task B and task C. To create a task, AsyncTask uses the silent_dependent_async function. Every task defines its own lambda as the first argument followed by a list of dependent tasks. *AsyncTask users must follow the rule that the dependent tasks must be created before the current task*. For instance, task A must be created before task B. Upon returning from silent_dependent_async, AsyncTask obtains an instantiated task object. After creating all tasks, executor.wait_for_all is called to wait for all tasks to finish.

```
int main(){
  Executor executor;
  // create 4 asynchronous tasks
  auto A = executor.silent_dependent_async([](){
    printf("Task A\n"); });
  auto B = executor.silent_dependent_async([](){
    printf("Task B\n"); }, A);
  auto C = executor.silent_dependent_async([](){
    printf("Task C\n"); }, A);
  auto D = executor.silent_dependent_async([](){
    printf("Task D\n"); }, B, C);
  // wait for the task graph to finish
  executor.wait_for_all();
}
```

Listing 1: AsyncTask implementation of Figure 1.

B. Task Scheduling Algorithm

AsyncTask designs a new task scheduling algorithm to support its new programming model. There are three main designs in AsyncTask's scheduling algorithm. First, every task object keeps track of its successor tasks in its own successor list instead of in a global data structure. This design can efficiently resolve a dependency when a task finishes the execution. For example, in Figure 1, both task B and task C have task A as the dependent task, and AsyncTask inserts task B and task C in task A's successor list. When task A finishes, AsyncTask can quickly resolve the associated dependencies by directly checking task A's successor list rather than iterating every existing task to see what dependencies to resolve.

Second, every task object has a shared ownership between worker threads. As every task is an instantiated task object, it will be destroyed and returned to the operating system after it finishes the execution. To avoid inserting a new task into an empty or wrong task's successor list (the ABA problem [4]), AsyncTask leverages the idea of shared ownership to keep every task object alive throughout the whole program.

Third, every task has an atomic variable to protect its own successor list from data race. As multiple tasks could insert themselves into one task's successor list simultaneously, AsyncTask assigns every task an atomic variable and every new task must perform atomic operations before inserting itself into a task's successor list.

III. Experimental Results

We first demonstrate the impact of different topological sorting sequences on the large-scale timing analysis workloads. Then we showcase the significance of DTGP over STGP on large-scale applications. The goal of the experiment is to 1) point out an optimization direction for AsyncTask users, and 2) highlight the need for a technique to generate a topological sorting sequence that yields a better runtime performance than existing heuristic-based methods.

We compiled programs using g++11.4 with -std=c++20 and -O3 enabled. We ran all the experiments on a Ubuntu 22.04.3 machine with 16 Intel i7-11700 CPU at 2.50 GHz and 125 GB RAM. All data is an average of ten runs.

A. Timing Analysis Workloads

We tested AsyncTask on an industrial static timing analysis (STA) application [24], [36] that leverages task graph parallelism to parallelize graph-based analysis (GBA). STA is a critical step in the overall EDA flow because it analyzes the timing landscape of a circuit design and reports critical paths that do not meet the given constraints (e.g., setup time and hold time).

We considered the state-of-the-art open-source STA engine, OpenTimer [2], [24], as our experimental environment. OpenTimer formulates the GBA algorithm into a task graph and

TABLE I: Task ($\|V\|$) and edge ($\|E\|$) counts of 12 circuits.

Circuits	$\|V\|$	$\|E\|$	$\|V\| + \|E\|$	Size
c432	483	925	1408	Small
c499	604	1097	1701	
s400	626	1125	1751	
wb_dma	13125	16593	29718	Medium
tv80	17038	23087	40125	
ac97_ctrl	42438	53558	95996	
aes_core	66,751	86,446	153,197	Large
des_perf	303,690	387,291	690,981	
vga_lcd	397,809	498,863	896,672	
leon3mp_iccad	3,376,832	6,277,562	9,654,394	Giant
netcard_iccad	3,999,174	7,404,006	11,403,180	
leon2_iccad	4,328,255	7,984,262	12,312,517	

schedules dependent tasks across many heterogeneous cores for parallel execution. The task graph represents the circuit graph itself and can contain millions of tasks and dependencies for large designs. Each task computes the required timing information at its corresponding node in the circuit graph (e.g., parasitics, slew, delay, arrival time), while each edge represents a dependency between two tasks. Table I lists the statistics of the 12 circuits we used. $\|V\|$ denotes the number of the tasks in a task graph generated from the circuit and $\|E\|$ denotes the number of the edges. We categorized the 12 circuits into 4 categories based on the size. For example, $\mathrm{c432}$, $\mathrm{c499}$, and $\mathrm{s400}$ are small-sized circuits and $\mathrm{leon3mp\_iccad}$, $\mathrm{netcard\_iccad}$, and $\mathrm{leon2\_iccad}$ are giant-sized circuits.

B. Methodology

To run the STA workload using AsyncTask, we first performed a topological sorting on a task graph to get a sequence of tasks and then created tasks in the topological order using AsyncTask's $\mathrm{silent\_dependent\_async}$ API. In the API, the first argument is the callable function for each task (e.g., timing operations of parasitics, slew, delay, arrival time), which is followed by a list of dependent tasks.

To get a valid topological order of a task graph, we chose three heuristic approaches, the BFS- and DFS-based topological sorting algorithm, and a random approach. For the first two algorithms, we can modify the typical BFS and DFS algorithm to get the orders. For the random approach, we repeated the steps to get the order: 1) push tasks whose dependencies have been resolved into an array, 2) randomly pop one task from that array, and 3) resolve the associated dependencies for that task.

C. Impact of Different Topological Orders

To schedule a task graph, AsyncTask must follow a topological order of that graph because AsyncTask can not associate a dependency with a non-existing task. For example, in Listing 1, AsyncTask implements Figure 1 in the A-B-C-D order. AsyncTask can not schedule the task graph in the D-A-B-C order as we were trying to associate task D with non-existing task B and task C. For a task graph, there are multiple valid topological orders. For instance, both A-B-C-D and A-C-B-D are valid in Figure 1. Different valid orders could lead to

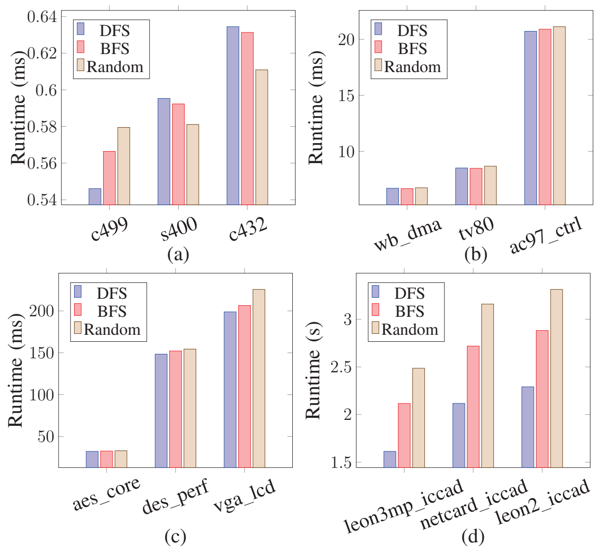

Fig. 2: Runtime comparison of AsyncTask scheduling tasks in the Depth First Search (DFS)-based, Breadth First Search (BFS)-based topological, and the random orders on 12 circuits, respectively. (a) Runtime of small-sized circuits. (b) Runtime of medium-sized circuits. (c) Runtime of large-sized circuits. (d) Runtime of giant-sized circuits.

distinct runtime performance. Take a task graph with a very deep branch as an example. It is better to schedule tasks with DFS-based topological order than BFS-based order because of higher data locality DFS can bring in the scenario. In this section, we discuss the runtime performance of AsyncTask scheduling tasks based on the DFS- and BFS-based topological sorting algorithm, and one random approach.

Figure 2 shows the runtime comparison of AsyncTask scheduling tasks in the DFS-based, BFS-based and a random orders. We can see that the implementation with a BFS order runs faster on $\mathrm{wb\_dma}$ and $\mathrm{tv80}$. For example, BFS order is 0.4% faster than DFS for $\mathrm{tv80}$. The implementation with a DFS order runs faster on another 8 circuits. For example, DFS order is 0.9%, 2.5%, and 27% faster than BFS for $\mathrm{ac97\_ctrl}$, $\mathrm{des\_perf}$, and $\mathrm{netcard\_iccad}$, respectively. The runtime difference between BFS and DFS order comes from the variation of the graph architecture. For $\mathrm{wb\_dma}$ and $\mathrm{tv80}$, their graphs have a wide diameter, meaning a task has more neighboring tasks at the same depth level than descendant tasks in a branch. As a result, in BFS order, AsyncTask can schedule a bunch of tasks at the same depth level at once and assign these tasks across all execution units for higher load balancing. On the other hand, for graphs that are deep in a branch, such as $\mathrm{des\_perf}$, AsyncTask is able to take advantage of data locality, increase the cache hit rate, and get better performance.

We also find out that for large- and giant-sized circuits, such as $\mathrm{aes\_core}$ and $\mathrm{netcard\_iccad}$, DFS order always gives the better performance. We attribute the finding to the reason below. In modern circuit designs a circuit gate does not have a very high fan-out (the number of output pins connecting to the next gates) because a high fan-out means a high current flow required to meet the total needs of all the connected next gates. A high current flow on a circuit gate could easily lead to reliability issue. As a result, modern circuits are usually deep in shape, especially for large and complex circuits, and the benefit of taking the advantage of data locality outperforms the benefit of load balancing. This finding gives AsyncTask users a good optimization hint when dealing with large-scale timing analysis applications.

In Figure 2, we also find out the random approach gives the best performance despite the graph structure when running $\mathrm{c432}$ and $\mathrm{s400}$. For example, the random approach is 3.86% and 3.33% faster than DFS and BFS when running $\mathrm{c432}$, respectively. This finding puts AsyncTask users in a difficult position because they can not choose DFS- or BFS-based order simply based on the graph structure when the random approach could finish the fastest. Moreover, for STA engineers, they need to iterate the same analysis multiple times and the random approach could give the best order in one iteration while the worst in other iterations. Through this finding, we highlight the need for a new technique that can help users to make the decision about the sorting algorithm or generate a valid topological order that yields a better runtime performance than the three heuristic-based approaches.

D. Significance of DTGP on Large-Scale Workloads

We show the significance of DTGP over STGP on large-scale timing analysis workloads. As OpenTimer adopts STGP, we used it for the evaluation. Figure 3 shows the runtime breakdown of OpenTimer. We can see that the task creation time for STGP increases as the graph grows in size. For example, the task creation time increases from 5.65% in a small-sized circuit to 15.66% in a giant-sized circuit. That means, as the circuit graph grows, we can not neglect the overhead of task creations anymore. As the circuit design is becoming larger in size and more complex in functionality in the current technology, STGP is introducing more overhead in task creations, which standouts the significance of using DTGP in large-scale timing analysis applications.

IV. Conclusion

In this paper, we have conducted an experimental study to investigate in the impact of different topological orders on running large-scale timing analysis workloads using recently-published DTGP library, AsyncTask. We have also demonstrated the significance of DTGP over STGP. In the end, we have highlighted the need for a new and clever technique to generate a valid topological order that yields a better runtime performance than heuristic-based algorithms. Future work will focus on applying AsyncTask to other EDA applications, such as distributed computing [26], [29], [30], [38], macro modeling [46], and path-based analysis [34], [35], [37], [39], [40].

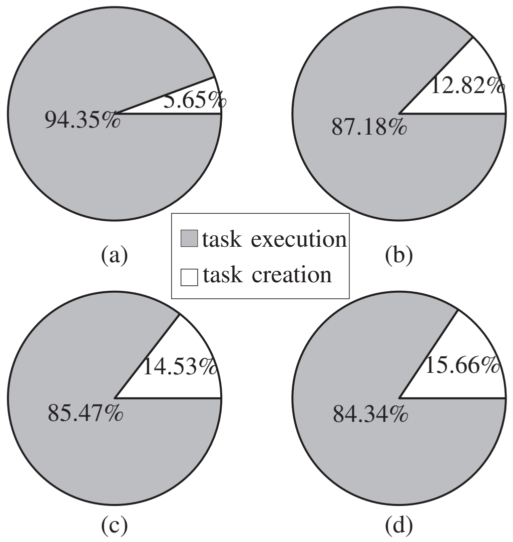

Fig. 3: Runtime breakdown of OpenTimer's construct-and-run model on four selected circuits. (a) Chart of a small-sized circuit, $\mathrm{c432}$. (b) Chart of a medium-sized circuit, $\mathrm{ac97\_ctrl}$. (c) Chart of a large-sized circuit, $\mathrm{vga\_lcd}$. (d) Chart of a giant-sized circuit, $\mathrm{leon2\_iccad}$.

Acknowledgment

We are grateful for the NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

[1] OpenMP. https://www.openmp.org/
[2] OpenTimer. https://github.com/OpenTimer/OpenTimer
[3] Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. In: Journal of Parallel and Distributed Computing. pp. 3202–3216 (2014)
[4] (ABA Problem), https://en.wikipedia.org/wiki/ABA_problem
[5] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J.J.: Parsec: Exploiting heterogeneity to enhance scalability. In: Computing in Science Engineering. pp. 36–45 (2013)
[6] Chang, C., Huang, T.W., Lin, D.L., Guo, G., Lin, S.: Ink: Efficient Incremental k-Critical Path Generation. In: ACM/IEEE DAC (2024)
[7] Chiu, C.H., Huang, T.W.: Composing Pipeline Parallelism Using Control Taskflow Graph. In: ACM HPDC. p. 283–284 (2022)
[8] Chiu, C.H., Huang, T.W.: Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results. In: ACM/IEEE DAC. p. 1388–1389 (2022)
[9] Chiu, C.H., Lin, D.L., Huang, T.W.: An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In: Euro-Par Workshop (2022)
[10] Chiu, C.H., Lin, D.L., Huang, T.W.: Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In: IEEE/ACM ICCAD (2023)
[11] Chiu, C.H., Xiong, Z., Guo, Z., Huang, T.W., Lin, Y.: An efficient task-parallel pipeline programming framework. In: ACM International Conference on High-performance Computing in Asia-Pacific Region (HPC Asia) (2024)
[12] Dzaka, E., Lin, D.L., Huang, T.W.: Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In: IEEE IPDPSw. pp. 923–929 (2023)
[13] Guo, G., Huang, T.W., Lin, C.X., Wong, M.: An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In: ACM/IEEE DAC. pp. 1–6 (2020)

[14] Guo, G., Huang, T.W., Lin, Y., Wong, M.: GPU-accelerated Critical Path Generation with Path Constraints. In: IEEE/ACM ICCAD. pp. 1–9 (2021)

[15] Guo, G., Huang, T.W., Wong, M.: Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In: IEEE/ACM DATE. pp. 1–6 (2023)

[16] Guo, Z., Huang, T.W., Lin, Y.: GPU-Accelerated Static Timing Analysis. In: IEEE/ACM ICCAD (2020)

[17] Guo, Z., Huang, T.W., Lin, Y.: A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In: ACM/IEEE DAC. pp. 715–720 (2021)

[18] Guo, Z., Huang, T.W., Lin, Y.: HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In: IEEE/ACM ICCAD. pp. 1–9 (2021)

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

[20] Huang, T.W., Lin, C.X., Guo, G., Wong, M.D.F.: Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. pp. 974–983 (2019)

[21] Huang, T.W.: A General-Purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In: IEEE/ACM ICCAD (2020)

[22] Huang, T.W.: Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In: IEEE HPEC. pp. 1–2 (2022)

[23] Huang, T.W.: qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In: IEEE IPDPS. pp. 746–756 (2023)

[24] Huang, T.W., Guo, G., Lin, C.X., Wong, M.: OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In: IEEE TCAD. pp. 776–789 (2021)

[25] Huang, T.W., Hwang, L.: Task-Parallel Programming with Constrained Parallelism. In: IEEE HPEC. pp. 1–7 (2022)

[26] Huang, T.W., Lin, C.X., Guo, G., Wong, M.D.F.: A General-Purpose Distributed Programming System Using Data-Parallel Streams. In: ACM MM. p. 1360–1363 (2018)

[27] Huang, T.W., Lin, C.X., Guo, G., Wong, M.D.F.: Essential Building Blocks for Creating an Open-Source EDA Project. In: ACM/IEEE DAC (2019)

[28] Huang, T.W., Lin, C.X., Wong, M.: OpenTimer v2: A Parallel Incremental Timing Analysis Engine. IEEE Design and Test (DAT) (2021)

[29] Huang, T.W., Lin, C.X., Wong, M.D.F.: DtCraft: A distributed execution engine for compute-intensive applications. In: IEEE/ACM ICCAD. pp. 757–765 (2017)

[30] Huang, T.W., Lin, C.X., Wong, M.D.F.: DtCraft: A High-Performance Distributed Execution Engine at Scale. IEEE ICAD **38**(6), 1070–1083 (2019)

[31] Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In: IEEE TPDS. pp. 1303–1320 (2022)

[32] Huang, T.W., Lin, D.L., Lin, Y., Lin, C.X.: Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System. IEEE TCAD **41**(5), 1448–1452 (2022)

[33] Huang, T.W., Lin, Y., Lin, C.X., Guo, G., Wong, M.D.F.: Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale. IEEE TCAD **40**(8), 1687–1700 (2021)

[34] Huang, T.W., Wong, M.D.F.: Accelerated path-based timing analysis with mapreduce. In: ACM ISPD. p. 103–110 (2015)

[35] Huang, T.W., Wong, M.D.F.: On fast timing closure: speeding up incremental path-based timing analysis with mapreduce. In: ACM/IEEE SLIP. pp. 1–6 (2015)

[36] Huang, T.W., Wong, M.D.F.: OpenTimer: A High-Performance Timing Analysis Tool. In: IEEE/ACM ICCAD. p. 895–902 (2015)

[37] Huang, T.W., Wong, M.D.F.: UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR. IEEE TCAD **35**(11), 1862–1875 (2016)

[38] Huang, T.W., Wong, M.D.F., Sinha, D., Kalafala, K., Venkateswaran, N.: A distributed timing analysis framework for large designs. In: ACM/IEEE DAC. pp. 1–6 (2016)

[39] Huang, T.W., Wu, P.C., Wong, M.D.F.: Fast Path-Based Timing Analysis for CPPR. In: IEEE/ACM ICCAD. p. 596–599 (2014)

[40] Huang, T.W., Wu, P.C., Wong, M.D.F.: UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm. In: IEEE/ACM ICCAD. p. 758–765 (2014)

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

[42] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M.X., Chen, D., Lee, H., Ngiam, J., Le, Q.V., Wu, Y., Chen, Z.: GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In: Advances in Neural Information Processing Systems. pp. 103–112 (2019)

[43] Jia, Z., Lin, S., Qi, C.R., Aiken, A.: Exploring hidden dimensions in accelerating convolutional neural networks. In: Proceedings of the 35th International Conference on Machine Learning. pp. 2274–2283 (2018)

[44] Jia, Z., Zahari, M., Aiken, A.: Beyond data and model parallelism for deep neural networks. In: Proceedings of Machine Learning and Systems. pp. 1–13 (2019)

[45] Jiang, S., Huang, T.W., Yu, B., Ho, T.Y.: SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In: ACM ICPP (2023)

[46] Lai, T.Y., Huang, T.W., Wong, M.D.F.: LibAbs: An Efficient and Accurate Timing Macro-Modeling Algorithm for Large Hierarchical Designs. In: ACM/IEEE DAC (2017)

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

[48] Lin, C.X., Huang, T.W., Guo, G., Wong, M.D.F.: A Modern C++ Parallel Task Programming Library. In: ACM MM. p. 2284–2287 (2019)

[49] Lin, C.X., Huang, T.W., Wong, M.D.F.: An efficient work-stealing scheduler for task dependency graph. In: IEEE ICPADS. pp. 64–71 (2020)

[50] Lin, C.X., Huang, T.W., Yu, T., Wong, M.D.F.: A distributed power grid analysis framework from sequential stream graph. In: GLVLSI. p. 183–188. GLSVLSI '18 (2018)

[51] Lin, D.L., Huang, T.W.: Efficient GPU Computation Using Task Graph Parallelism. In: Euro-Par (2021)

[52] Lin, D.L., Huang, T.W.: Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. IEEE TPDS **33**(11), 3041–3052 (2022)

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

[54] Lin, D.L., Ren, H., Zhang, Y., Khailany, B., Huang, T.W.: From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus. In: Proceedings of the 51st International Conference on Parallel Processing. pp. 1–12 (2023)

[55] Lin, D.L., Zhang, Y., Ren, H., Wang, S.H., Khailany, B., Huang, T.W.: GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In: ACM/IEEE DAC (2023)

[56] Lin, S., Guo, G., Huang, T.W., Sheng, W., Young, E., Wong, M.: G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In: ACM/IEEE DAC (2024)

[57] Lin, Y., Li, W., Gu, J., Ren, H., Khailany, B., Pan, D.: Abcdplace: Accelerated batch-based concurrent detailed placement on multithreaded cpus and gpus. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. pp. 5083–5096 (2020)

[58] Liu, S., Liao, P., Zhang, R., Chen, Z., Lv, W., Lin, Y., Yu, B.: FastGR: global routing on CPU-GPU with heterogeneous task graph scheduler. In: Proceedings of the 2022 Conference and Exhibition on Design, Automation and Test in Europe. pp. 760–765 (2022)

[59] Morchdi, C., Chiu, C.H., Zhou, Y., Huang, T.W.: A Resource-efficient Task Scheduling System using Reinforcement Learning. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC) (2024)

[60] Zamani, Y., Huang, T.W.: A High-Performance Heterogeneous Critical Path Analysis Framework. In: IEEE HPEC. pp. 1–7 (2021)

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