Asynchronous Many-task System with Intelligent Scheduling

by

Cheng-Hsiang Chiu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of final oral examination: 08/21/2025

The dissertation is approved by the following members of the Final Oral Committee:
Tsung-Wei Huang, Professor, Electrical and Computer Engineering, Chair
Umit Yusuf Ogras, Professor, Electrical and Computer Engineering
Joshua San Miguel, Professor, Electrical and Computer Engineering
Yi Zhou, Professor, Computer Science Engineering, Texas A&M University

I dedicate this dissertation to all those who have supported me throughout my Ph.D. journey, especially my family and friends who provided both laughter and encouragement when I needed it most.

ACKNOWLEDGMENTS

Completing this Ph.D. has been a journey of growth, challenges, and deep gratitude. I would like to express my heartfelt appreciation to all those who supported me throughout this endeavor.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Tsung-Wei Huang. His mentorship, encouragement, and insightful guidance have been instrumental to my growth both academically and personally. I am truly thankful for his patience, high standards, and belief in my potential.

I am also sincerely thankful to my dissertation committee members, Professor Umit Yusuf Ogras, Professor Joshua San Miguel, and Professor Yi Zhou, for their valuable time, constructive feedback, and thoughtful questions that significantly helped improve the quality of this work.

In particular, I would like to thank Professor Cunxi Yu, Professor Yibo Lin, and Professor Priyank Kalla for their thoughtful feedback throughout the journey. Their expertise and support have played a significant role in the development of my work.

I am grateful to Dr. Zhuo Li and Dr. Michael Voss for providing me with the valuable opportunities to intern at Cadence and Intel during the summers. These experiences not only broadened my perspective on real-world applications but also deepened my understanding of industry challenges. Their mentorship and support from the whole group during these internships were truly invaluable.

I would like to thank my colleagues, Dr. Dian-Lun Lin, Che Chang, Wan-Lun Lee, Boyang Zhang, Chih-Chun Chang, Yi-Hua Chung, Jie Tong, Aditya Das Sarma, Shui Jiang, Chedi Morchdi, Jiaqi Yin, Yasin Zamani, Zizheng Guo, Dr. Guannan Guo, and Zhicheng Xiong, for the insightful discussions, shared frustrations, and the invaluable trust and friendship we built along the way. Your support made the difficult moments easier

and the good moments even better.

To my friends at the University of Wisconsin–Madison and the University of Utah, thank you for reminding me to laugh. From playing badminton to relaxing outdoors and sharing great food, your friendship has been a constant source of strength, joy, and refreshment throughout this experience.

I am also indebted to the administrative and technical staff at the University of Wisconsin-Madison and University of Utah. Your instant assistance has been invaluable throughout my doctoral studies.

Most importantly, I thank my family for their unconditional love, sacrifices, and encouragement. To my parents, thank you for your steadfast supports and constant faith in me. To my younger brother, thank you for being a lifelong source of strength and support. This achievement is as much yours as it is mine.

Finally, I am grateful to everyone, named or unnamed, who contributed to this journey in ways big or small. Thank you.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis addresses critical challenges in the discipline of parallel computing and task scheduling, focusing on task-parallel programming and task graph scheduling. There are six chapters in the thesis. The first three chapters are task-parallel programming models, Pipeflow, token dependency-aware Pipeflow, and AsyncTask. The last three chapters are reinforcement learning-based task graph scheduling, resource-efficient task scheduling, topological ordering for task graphs, and CUDA Graph scheduling optimization.

(1) **An Efficient Task-parallel Pipeline Programming Framework.** The pipeline is a fundamental pattern to parallelize a series of stage tasks over a sequence of data in loops. Mainstream libraries rely on data abstractions to schedule pipeline tasks, which complicates the scheduling design and is not efficient for applications with task parallelism only. To address this challenge, Pipeflow decouples task scheduling and data abstractions and introduces a lightweight scheduling policy to efficiently exploit pipeline parallelism in an application. We have demonstrated that Piepflow outperforms existing libraries up to 110.33% faster. This work significantly reduces the runtime, providing a crucial solution for pipeline applications that only exploit task parallelism.

In this work, Cheng-Hsiang Chiu was the primary contributor, responsible for the majority of the research and development efforts. Zhicheng Xiong and Zizheng Guo provided design ideas. Yibo Lin and Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the work. All authors participated in discussing the results and contributed to the preparation and review of this chapter.

(2) **A Task-parallel Pipeline Programming Model with Token Dependency.** Task-parallel pipeline framework explores pipeline parallelism in applications and is critical in many parallel and heterogeneous areas. However, existing solutions cannot deal with applications in which data dependencies exhibit in both forward and backward directions To address this need, we have extended Pipeflow to support application's bi-directional token dependencies through an expressive programming model and lightweight atomic counters in resolving token dependencies. We have demonstrated our token dependency-aware Pipeflow is 8.6% faster than existing implementation in video encoding applications. This work showcases Pipeflow's capability to explore pipeline parallelism across various pipeline applications.

In this work, Cheng-Hsiang Chiu was the primary contributor, responsible for the majority of the research and development efforts. Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the work. Wan-Luan Lee, Boyang Zhang, Yi-Hua Chung, and Che Chang provided design ideas. All authors participated in discussing the results and contributed to the preparation and review of this chapter.

(3) **Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms.** Parallelizing EDA applications that are extremely sparse, irregular, and control-flow intensive can benefit from the ability to express dynamic task parallelism across arbitrary decision-making points at runtime. However, existing libraries describe dynamic task dependencies in an indirect manner and rely on lock-based data structure to schedule tasks. To address this challenge, we have introduced AsyncTask to support the dynamic building of a computational task graph. We have demonstrated AsyncTask is up to 3.19$\times$ faster than existing libraries. This work presents a direct description of tasks, improves code readability, and develops an efficient scheduling al-

gorithm, which is extremely crucial for complex and irregular EDA applications.

In this work, Cheng-Hsiang Chiu was the primary contributor, responsible for the majority of the research and development efforts. Dian-Lun Lin provided design ideas. Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the work. All authors participated in discussing the results and contributed to the preparation and review of this chapter.

(4) **A Resource-efficient Task Scheduling System using Reinforcement Learning.** Efficiently scheduling millions of functional tasks of EDA applications in a computing environment that comprises manycore CPUs and GPUs is critically important. However, existing scheduling methods are typically hardcoded within an application that are not adaptive to the change of computing environment. To address the issue, we have introduced a novel reinforcement learning-based scheduling algorithm that can learn to adapt the performance optimization to a given runtime situation. We have demonstrated that our scheduling algorithm can achieve the same performance as the existing methods while using only 20% of CPU resources. This work highlights the capability of our algorithm to maintain the same runtime performance across concurrent workloads without experiencing performance degradation, which is crucial for EDA applications.

In this work, Cheng-Hsiang Chiu and Chedi Morchdi were both the primary contributors, responsible for the majority of the research and development efforts. Yi Zhou and Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the work. All authors participated in discussing the results and contributed to the preparation and review of this chapter.

(5) **Reinforcement Learning-generated Topological Order for Dynamic**

Task Graph Scheduling. Dynamic task graph scheduling (DTGS) allows applications to define the task graph structure on-the-fly, enabling concurrent task creations and task executions. To schedule tasks, DTGS relies on applications to define a topological order for the task graph. However, existing algorithms that generates this order primarily rely on heuristics like level-by-level sorting, which lack adaptability to dynamic computing environments. To address this need, we have introduced a novel method that leverages reinforcement learning to generate topological orders for DTGS systems. We have demonstrated that our method achieves a speedup of up to 1.52× over the existing solutions. This work is essential for task-parallel runtimes that employ diverse work stealing policies to support a broader range of applications.

In this work, Cheng-Hsiang Chiu was the primary contributor, responsible for the majority of the research and development efforts. Chedi Morchdi, Boyang Zhang and Che Chang provided design ideas. Yi Zhou and Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the work. All authors participated in discussing the results and contributed to the preparation and review of this chapter.

(6) **Optimizing CUDA Graph Scheduling with Reinforcement Learning : A Case Study in SSTA Propagation.**

CUDA Graph has shown potential in recent GPU-accelerated statistical static timing analysis (SSTA) propagation applications. However, application-given CUDA graphs are often suboptimal, as they focus on capturing circuit structures while overlooking GPU resource availability and scheduling constraints. To address this challenge, we have introduced a reinforcement learning-based framework that optimizes CUDA graphs by learning to restructure SSTA graphs through inter-

actions with the CUDA Graph runtime. We have demonstrated that our optimized CUDA graph can achieve up to a 12% runtime improvement over the application-given CUDA graph. This work is crucially important for CUDA Graph applications as our framework requires no changes to application-level algorithms, but instead restructures the given CUDA graph to guide the CUDA runtime toward better scheduling performance.

In this work, Cheng-Hsiang Chiu was the primary contributor, responsible for the majority of the research and development efforts. Chih-Chun Chang designed the problem formulation. Chedi Morchdi provided design ideas. Cunxi Yu, Yi Zhou and Tsung-Wei Huang supervised the research, providing guidance and oversight throughout the work. All authors participated in discussing the results and contributed to the preparation and review of this chapter.

PREVIOUS WORK

This thesis builds upon a substantial foundation of prior research in parallel computing, task-parallel programming, task graph scheduling, and reinforcement learning. Below, we summarize previous work relevant to each chapter of the thesis.

(1) **An Efficient Task-parallel Pipeline Programming Framework.** Pipeline programming models have received intensive research interest. Most of them are data-centric, using static template instantiation or dynamic runtime polymorphism to model data processing in a pipeline. To name a few popular examples: oneTBB [3] and TPL [108] require explicit definitions of input and output types for each stage; GrPPI [139] provides a composable abstraction for data- and stream-parallel patterns with a pluggable back-end to support task scheduling; FastFlow [10] models parallel dataflow using pre-defined sequential and parallel building blocks; TTG [18] focuses on dataflow programming using various template optimization techniques; SPar [40, 41, 56, 124] analyzes annotated attributes extracted from the data and stream parallelism domain, and automatically generates parallel patterns defined in FastFlow; Proteas [126] introduces a programming model for directive-based parallelization of linear pipeline; [148, 149] propose a self-adaptive mechanism to decide the degree of parallelism and generate the pattern compositions in FastFlow; OpenMP [5] uses task construct and depend clause to explore pipeline parallelism. Although these programming models are used in many applications, such as oneTBB in PARSEC [12], they constrain users to design pipeline algorithms using their data models, making it difficult to use, especially for applications that only need pipeline scheduling atop custom data structures. Pipeflow simply requires users to specify the pipeline structures (e.g., the number of parallel lines) and pipe callables, and

provides a scalable pipeline API for users to flexibly define the pipeline scheduling frameworks with dynamic structures based on their specific needs.

Existing pipeline scheduling algorithms typically co-design task scheduling and buffer structures to strive for the best performance. For instance, oneTBB [3] defines a per-stage buffer counter to synchronize data tokens among stages and parallel lines, coupled with a small object allocator to minimize the data allocation overhead; GRAMPS [140] designs a buffer manager with per-thread fix-sized memory pools to dynamically allocate new data and release used ones; FastFlow [10] designs a lock-free queue with a mechanism to transfer data ownership between senders and receivers, but this method can incur imbalanced load and requires non-trivial back-pressure management; HPX [93] counts on a channel data structure and standard future objects to pass data around tasks, but the creation of share states becomes expensive when the pipeline is large; Cilk-P [104] employs per-stage queues coupled with two counter types to track static and dynamic dependencies of each node, but it targets on-the-fly pipeline parallelism, which is orthogonal to our focus; FDP [142] proposes a learning-based mechanism to adapt scheduling to an environment, but it requires expensive runtime profiling that may not work well for highly irregular applications like CAD. Pipeflow leverages C++ simple atomic operations and assigns every pipe an atomic variable denoting the dependency. Since there is no data synchronization involved, the scheduling algorithm of Pipeflow is lightweight and efficient. In terms of load balancing, most pipeline schedulers leverage work stealing, which has been reported with better performance than static policies [16, 57, 104, 109, 134, 140, 141]. However, for some special cases, such as fine-grained load-imbalanced pipelines, static policies perform comparably. For example, Pipelight [129, 130] imple-

ments a load-balancing technique based on two static scheduling algorithms, DSWP [135, 136, 137] and LBPP [94]; Pipelite [128, 130] and URTS [127, 130] introduce dynamic schedulers using ticket-based synchronization and directive-based model language for linear pipelines, respectively. Although, in some special cases, work stealing [111] may not give the best runtime performance, Pipeflow and the most frameworks still adopt this algorithm as it has the best performance in most applications. While co-designing task scheduling and buffer structures has certain advantages for data-centric pipeline (e.g., data locality), the cost of managing data can be significant yet unnecessary, especially for applications that only exploit task parallelism in pipeline.

(2) **A Task-parallel Pipeline Programming Model with Token Dependency.** The state-of-the-art pipeline programming library, Pipeflow [31], introduces a programming model and an efficient task scheduling algorithm for users to explore pipeline parallelisms in applications. However, Pipeflow only deals with data dependency in forward direction, which limits Pipeflow's generalizability to pipeline applications, such as video encoding [12]. To handle the data dependency in both forward and backward directions, the most common way for existing libraries [12, 13, 31] is to reorder the execution order of data using low-level synchronization primitive, condition variable [1], and then feed the reordered data to the pipeline framework as PRASEC does [12, 13]. However, we notice three limitations of this approach: (1) Manipulating condition variable requires a deep understanding of this low-level synchronization primitive from users and is error-prone when dependency is intricate. (2) The approach is not an end-to-end implementation as users need to additionally reorder the data outside the original pipeline application. (3) The implementation could encounter deadlock when the data dependency is complex and

insufficient threads are spawned.

(3) **Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms.** Mainstream dynamic task graph programming libraries include OpenMP [5], PaRSEC [17], and OpenCilk [16, 141]. OpenMP [5] is a popular library that simplifies the development of parallel applications by adding parallelism to existing serial code through the use of compiler directives, `pragmas`, and runtime library routines. To implement a simple dynamic task graph application, OpenMP uses `#pragma omp task` construct to define a task and `depend` clause to specify that task's dependencies. Since OpenMP relies on a task's input and output data to describe a task's dependencies, applications need a data storage to store the data of every task, which is done by defining a dynamic array. Then applications use the entries in the dynamic array as the inputs and outputs for a task to define the dependencies. To schedule tasks, OpenMP implements a lock-based hash table, in which the key of each entry is the address of a task's input or output data, and the value of that entry is a list of tasks accessing that address. As scheduling tasks require frequent accessing and updating the hash table, the overhead of using mutexes is heavy and can impact the overall runtime performance when running large and complex task graphs with multiple threads.

PaRSEC [17, 58] is a task-based runtime for distributed system. It leverages Domain Specific Languages (DSL) in its dataflow model to implement applications. To program a PaRSEC implementation, applications need the following steps: (1) Initialize a Message Passing Interface (MPI) engine as PaRSEC is a runtime for distributed system. (2) Define an application data structure using PaRSEC memory allocator to correctly build up the dependencies between tasks. (3) Initialize a PaRSEC taskpool to execute the tasks. (4) Define PaRSEC tasks and their function definitions. The task scheduling algorithm of PaRSEC

is similar to OpenMP's design. Both of them rely on a lock-based hash table to manage the dependencies between tasks. The main difference is that PaRSEC additionally considers where to execute tasks that are created at a remote machine.

OpenCilk [16, 141] is a software infrastructure for task-parallel programming. A typical OpenCilk code is to spawn threads for tasks' operations and explicitly join threads for synchronization. To implement a dynamic task graph application, users spawn a thread to run a task using the `cilk_spawn` directive, and explicitly join the thread to finish a task's execution using `cilk_sync`. As OpenCilk uses explicit synchronization directives to manage dependencies between tasks, the task scheduling algorithm is to wake up a thread from its thread pool to do a task's operation, and release that thread to the thread pool. When reaching the synchronization directives, the program execution halts until all threads finish and return to the thread pool.

Although OpenMP, PaRSEC, and OpenCilk have been used in many applications, we find several limitations of using them for DTGP: (1) Describing a task's dependencies through that task's input and output data is not expressive. Applications need to figure out the dataflow between two tasks to represent the task dependency, which is an indirect description and could reduce the code readability. (2) Programming a large task graph is very verbose. Applications have to explicitly indicate a task's input and output data, such as using `in` and `out` in OpenMP or `INPUT` and `OUTPUT` in PaRSEC. For PaRSEC users, they have to additionally write `PARSEC_DTD_ARG_END` to denote the end of input arguments and `PARSEC_HOOK_RETURN_DONE` to indicate the end of function definition. (3) Relying on a lock-based hash table to schedule tasks is not efficient. Their runtimes have to acquire a mutex when accessing the hash table, which introduces non-negligible lock overheads especially when running with multiple threads to schedule

a complex task graph.

(4) **A Resource-efficient Task Scheduling System using Reinforcement Learning.** To schedule tasks, existing solutions either resort to general-purpose heuristics (e.g., work stealing [73, 75, 111, 113]) or a custom scheduling method (e.g., hardcoded [152]). These solutions are typically not adaptive to the change in the computing environment and often consume large scheduling resources due to the randomness involved in dynamic load balancing.

Recent advances in machine learning have focused on designing a new scheduling framework that learns to interact with a computing environment [36]. Despite exciting progress in learning-based scheduling solutions, most of them target *independent* job batches in a high-performance computing (HPC) cluster. These solutions are not suitable for most computer-aided design (CAD) problems where the goal is to find a *resource-efficient* scheduling plan for running dependent tasks using minimal CPU resources. Because many CAD task graphs are much larger and more complex than conventional HPC workloads.

(5) **Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling.** To schedule tasks with the task dependency constraints, dynamic task graph scheduling runtime requires applications to create tasks in a topological order of the task graph. To obtain the order, topological sorting algorithms, such as Kahn's algorithm [4], are widely used. These heuristic-based algorithms generate orders primarily based on the graph structures, such as level-by-level sorting. However, such heuristic-based approaches have limitations. First, solely relying on graph structure lacks adaptability to dynamic changes in the computing environment. This can lead to suboptimal scheduling and can consume large scheduling resources due to the

randomness involved in dynamic task graph runtime's dynamic load balancing [113]. Second, heuristic algorithms generate deterministic orders. But, topological orders of a task graph are not unique and different topological orders for the same task graph can lead to substantial performance differences.

(6) **Optimizing CUDA Graph Scheduling with Reinforcement Learning: A Case Study in SSTA Propagation.**

Statistical static timing analysis (SSTA) is a critical step in Electronic Design Automation (EDA) as it enables more accurate delay estimation than traditional static timing analysis (STA) by modeling on-chip process variation (OCV) as random variables [14, 37]. [85] uses numerical integration to estimate circuit yield by exploring device parameter combinations, while [147] models gate delays as random variables and propagates rise and fall arrival time statistically through the timing graph. As design complexity continues to grow, manufacturing variations have introduced a broad range of OCVs that SSTA algorithms must evaluate during propagation. [23, 42, 45, 49, 121] emerge as a promising solution to meet the growing performance demands of SSTA. To schedule SSTA workloads on GPUs more efficiently, the recent state-of-the-art [23] leverages *CUDA Graph* to model SSTA propagation as a *GPU task graph*. However, application-given CUDA graphs by [23] are often suboptimal because the application's CUDA graphs prioritize capturing circuit structure but overlook GPU resource availability and scheduling constraints. This oversight often results in resource contention (e.g., multiple tasks competing for limited GPU resources) and reduced task parallelism, as tasks are forced to wait instead of executing concurrently.

1 AN EFFICIENT TASK-PARALLEL PIPELINE PROGRAMMING FRAMEWORK

1.1 Abstract

The pipeline is a fundamental pattern to parallelize a series of stage tasks over a sequence of data in loops. Mainstream pipeline programming frameworks count on data abstractions to perform pipeline scheduling. Although this design is convenient for data-centric parallel applications, it is not efficient for algorithms that only exploit task parallelism in the pipeline. To address the limitation, we introduce a new task-parallel pipeline programming framework called *Pipeflow*. Pipeflow separates data abstractions and task scheduling, enabling a more efficient implementation of task-parallel pipeline algorithms than existing frameworks. We have evaluated Pipeflow on both micro-benchmarks and real-world applications. For example, in a timing analysis workload that explores pipeline parallelism to speed up the runtime performance, the Pipeflow's implementation outperforms the oneTBB's implementation up to 110.33% faster.

1.2 Introduction

The pipeline is a fundamental parallel pattern to model parallel executions through a linear chain of stages. Each stage processes a *data token* after the previous stage, applies an abstract function to that token, and then resolves the dependency for the next stage. Multiple data tokens can be processed simultaneously across different stages whenever dependencies are met. For example, in circuit simulation [63, 76, 79, 80], some operations on a gate (e.g., NAND, OR, AND) do not depend on other gates and thus can be done at multiple logic levels simultaneously, while operations at the same levels require processing prior levels first. As pipeline parallelism widely

exists in modern computing applications [12], there is always a need for new pipeline programming frameworks to streamline the implementation complexity of pipeline algorithms.

Recently, several pipeline programming frameworks have emerged to assist developers in implementing pipeline algorithms without worrying about scheduling details, such as oneTBB [3], FastFlow [10], GrPPI [139], Cilk-P [104], SPar [40], and HPX-pipeline [93]. While each of these frameworks has its pros and cons, a common design philosophy is to perform data synchronizations using buffers between stages (i.e., *data abstraction*) in their pipeline scheduling designs, as illustrated in Figure 1.1. This design is convenient for data-centric pipeline applications but also has two limitations. Firstly, users have to design their pipeline algorithms in the data-parallel manner. However, data management is often application-dependent. Many applications exhibit pipeline parallelism among *tasks* rather than data. For example, the VLSI timing analysis application [25, 26] formulates a sequence of linearly dependent propagation tasks in a graph node and runs independent nodes in parallel to efficiently update the timing data from a custom global shared graph data structure. The real need is a *pipeline scheduling framework* to schedule and run tasks while leaving data management completely to applications.

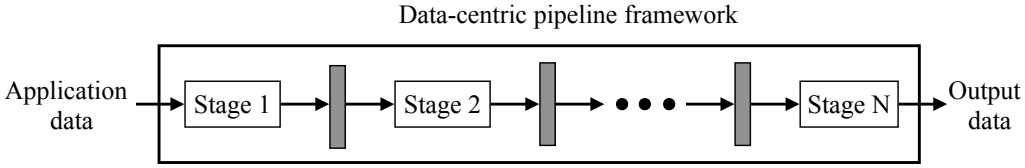


Figure 1.1: An illustration of data abstraction in a pipeline framework. Gray bars are buffers used for data synchronizations.

Secondly, scheduling algorithms involve complex synchronizations between data and buffer structures. These frameworks typically leverage object allocators and buffer structures to manage temporary data between

stages (e.g., oneTBB [3]). However, the synchronizations can be redundant in some applications. For instance, ferret [12], a pipeline benchmark of PARSEC, defines six stages (loading, segmentation, extraction, indexing, ranking, and output) in its oneTBB implementation to perform image similarity search. Every stage is defined as a derived class of oneTBB's `tbb::filter` and has an overridden operator that takes an input `void*` pointer returned from the previous stage. The pointer points to a *global* data structure `all_data`, bypassing all the data abstractions in the oneTBB pipeline.

To overcome these limitations, we introduce in this paper *Pipeflow*, a new task-parallel pipeline programming framework. We summarize our contributions as follows:

- **Task-Parallel Pipeline.** We have introduced a new *task-parallel* pipeline programming concept that separates task scheduling and data abstraction. This separation allows us to concentrate on the pipeline tasking itself, enabling a more efficient implementation of task-parallel pipeline algorithms than existing frameworks.
- **Programming Model.** We have introduced a new C++ programming model to support our concept. Unlike existing models, we do not provide yet another data abstraction but a flexible framework for users to fully control their application data atop a task-parallel pipeline scheduling framework.
- **Scheduling Algorithm.** We have introduced a new scheduling algorithm to schedule stage tasks across parallel lines. Since we do not touch data abstraction, we can avoid complex data buffer designs and synchronization mechanisms to enable more lightweight and efficient scheduling.

We have evaluated Pipeflow on both micro-benchmarks and real-world applications. For example, in a real-world VLSI static timing analysis

workload, the Pipeflow implementation outperforms the oneTBB implementation up to 110.3% faster. Right now, Pipeflow is merged into the open-source Taskflow project [72].

1.3 Background

We first review the pipeline basics and then detail the motivation of Pipeflow. We then argue a new task-parallel pipeline programming model is needed for many important industrial and research areas, e.g., circuit design [63].

Pipeline Basics

Pipeline parallelism is commonly used to parallelize various applications, such as stream processing, video processing, and dataflow systems. These applications exhibit parallelism in the form of a *linear pipeline*, where a linear sequence of abstraction functions, namely *stages*, $F = \langle f_1, f_2, \cdots, f_j \rangle$, is applied to an input sequence of data tokens, $D = \langle d_1, d_2, \cdots, d_i \rangle$. A linear pipeline can be thought of as a loop over the data tokens of D. Each iteration i processes an input token d_i by applying the stage functions F to d_i in order. Depending on the number of *parallel lines*, $L = \langle l_1, l_2, \cdots, l_k \rangle$, to process data tokens, parallelism arises when iterations overlap in time. For instance, the execution of token d_i at stage f_j of parallel line l_k, denoted as $f_j^k(d_i)$, can overlap with $f_{j-1}^{k+1}(d_{i+1})$. A stage can be a *parallel* type or a *serial* type to specify whether $f_j^k(d_i)$ can overlap with $f_j^{k+1}(d_{i+1})$ or not. Figure 1.2 shows the dependency diagram of a 3-stage (serial-serial-parallel) pipeline.

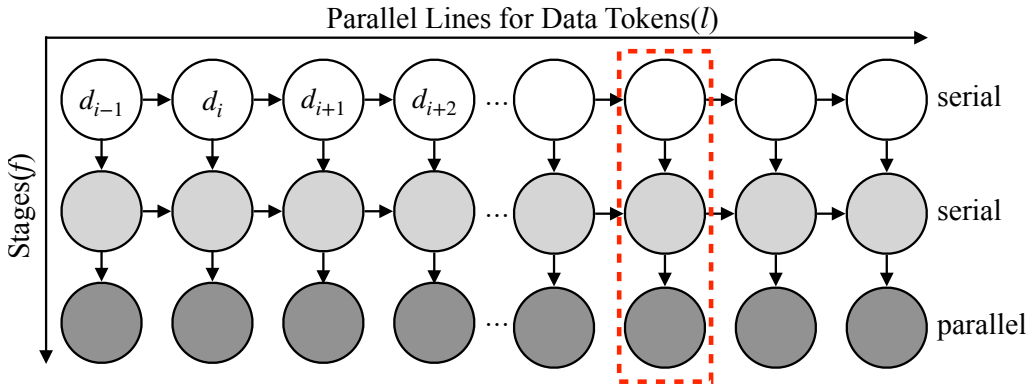


Figure 1.2: Dependency diagram of a 3-stage (serial-serial-parallel) pipeline. Each node represents a task that applies a stage function to a data token. Each edge represents a dependency between two tasks. The dashed rectangle denotes one parallel line.

Task-parallel Pipeline Parallelism

Pipeflow is motivated by our research projects on developing parallel timing analysis algorithms for very large scale integration (VLSI) computer-aided design (CAD) [43, 45, 48, 50, 61, 63, 152]. Timing analysis is a critical step in the overall CAD flow because it validates the timing performance of a digital circuit. As design complexity continues to grow exponentially, the need to efficiently analyze the timing has become the major bottleneck to the design closure flow. For instance, generating a comprehensive timing report (e.g., pessimism removal, hundreds of corners, etc.) for a multi-million-gate design can take several hours [92]. To reduce the analysis runtime, there is an increasing trend of adopting manycore parallelism by new timing analysis algorithms recently [22, 49, 51, 64, 77, 78, 81, 82, 88, 114, 120].

The most widely used strategy, including commercial timers, to parallelize timing analysis is *pipeline*. Figure 1.3 illustrates this strategy using forward timing propagation as an example [75]. The circuit graph is first levelized into a level list using topological sort. Nodes at the same level

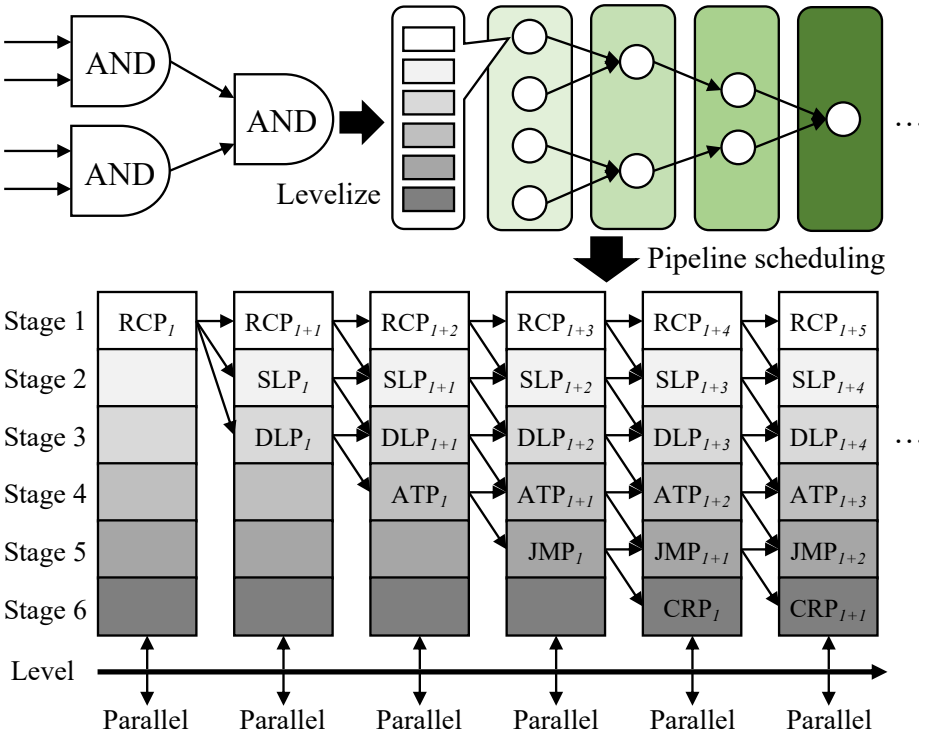


Figure 1.3: Parallel timing propagations [75]. Linearly dependent timing data (e.g., slew) is updated across graph nodes in a task-parallel pipeline fashion.

are independent of each other and can run in parallel. Each node runs a sequence of *linearly dependent* propagation tasks, including parasitics (RCP), slew (SLP), delay (DLP), arrival time (ATP), jump points (JMP), and common path pessimism reduction (CRP) to update its timing data from a *custom global* and *application-dependent* circuit graph data structure. Different propagation tasks can overlap across different levels using pipeline parallelism.

This type of *task-parallel* pipeline strategy is ubiquitous in many parallel CAD algorithms, such as logic simulation [102, 119] and physical design [65, 67, 68, 100], because computations frequently flow through circuit networks. We have observed three important properties that make

mainstream pipeline programming frameworks fall short of our needs: (1) Unlike the typical data-parallel pipeline, the pipeline parallelism in many CAD algorithms is driven by *tasks* rather than data. (2) Data is not directly involved in the pipeline but in the *graph data structure* defined by a custom algorithm. (3) From a user's standpoint, the real need is a *pipeline scheduling framework* to help schedule and run tasks on input tokens across parallel lines while leaving data management completely to applications; in our experience, users disfavor another library data abstraction to perform pipeline scheduling, as it often incurs development inconvenience and unnecessary data conversion overheads.

1.4 Pipeflow

Inspired by the need for parallel CAD algorithms, Pipeflow introduces a new task-parallel pipeline programming model for users to create a pipeline scheduling framework without data abstraction. In this section, we will dive into the technical details of Pipeflow.

Programming Model

Pipeflow leverages modern C++ and template techniques to strike a balance between expressiveness and generality. Listing 1.1 shows the Pipeflow code that implements the pipeline in Figure 1.2. Pipeflow has one API `Pipeline` that allows users to define the pipeline structure and explore the pipeline parallelism in their applications. There are two steps to create a Pipeflow application: (1) define the pipeline structure using template instantiation using the `Pipeline` API and (2) define the application data storage, if needed. In Pipeflow, the terms "pipe" and "stage" are interchangeable. For the first step, users define the number of parallel lines and the abstract function of each pipe in a `Pipeline` object. For each pipe, users define the pipe type and a pipe callable using `Pipe`. A pipe can be either a

serial type (`PipeType::SERIAL`) or a parallel type (`PipeType::PARALLEL`). The pipe callable takes an argument of `Pipeflow` type, which is created by the scheduler at runtime. A `Pipeflow` object `pf` represents a *scheduling token* and contains several methods for users to query the runtime statistics of that token, including the parallel line, pipe, and token numbers.

```
const size_t num_lines  = 4;
std::variant<float, std::string> data_type;
std::array<data_type, num_lines> buffer;
Pipeline pl(num_lines,
  // First pipe
  Pipe{PipeType::SERIAL,
    [&](Pipeflow& pf) {
      if ( !data.ready() ) {
        pf.stop();
      } else {
        // Generate a float and save it in buffer
        buffer[pf.line()] = data.get();
      }
    }
  },
  // Second pipe
  Pipe{PipeType::SERIAL,
    [&](Pipeflow& pf) {
      // Generate a string and save it in buffer
      buffer[pf.line()] =
        make_string(std::get<0>(buffer[pf.line()]));
    }
  },
  // Third pipe
  Pipe{PipeType::PARALLEL,
    [&](Pipeflow& pf) {
      std::cout << std::get<1>(buffer[pf.line()]);
    }
  }
);
```

```
// Run the pipeline object pl
pl.run();
```

Listing 1.1: Pipeflow code of Figure 1.2, assuming the first pipe generates `float` and the second pipe generates `string` outputs.

Pipeflow does not have any data abstraction but gives applications full control over data management. In our example, since the first and the second pipes generate `float` and `std::string` outputs, respectively, we create a one-dimensional (1D) array, `buffer`, as the application data storage to store data in uniform storage using `std::variant<float, std::string>`. The dimension of the array is equal to the number of parallel lines, as Pipeflow schedules only one token per parallel line. Each entry `buffer[i]` stores the data that is being processed at parallel line `i`, which can be retrieved by `Pipeflow::line`. This organization is space-efficient because we use only a 1D array to represent data processing in a two-dimensional (2D) scheduling map. Additionally, by delegating data management to applications, we can avoid dynamic data conversion between the library and the application, which typically counts on virtual function calls to convert a generic type (e.g., *void**, *std::any*) to an arbitrary user type [3, 10].

Based on the pipeline structure and data layout defined above, we instantiate a `Pipeline` object, `pl`. This template-based design enables the compiler to optimize each pipe type, such as using a fixed-layout functor to store the callable and its captured data. Finally, we call `run` to submit the object `pl` to a runtime and execute it.

```
using P = Pipe<std::function<void(Pipeflow&)>>;
// 6 pipes
std::vector<P> p(6, create_pipe());
// Pipeline of 4 parallel lines and 6 pipes
ScalablePipeline pl(4, p.begin(), p.end());
// First run
pl.run();
// Resize p to 3 pipes
```

```
p.resize(3);
// Pipeline of 4 parallel lines and 3 pipes
pl.reset(p.begin(), p.end());
// Second run
pl.run();
```

Listing 1.2: Scalable pipeline model in Pipeflow to accept variable assignments of pipes.

`Pipeline` requires instantiation of all pipes at the construction time. While this design gives compilers more freedom to optimize the layout of each pipe type, it prevents applications from varying the pipeline structure at runtime; for instance, the number of pipes might depend on the problem size, which can be runtime variables. To overcome this limitation, Pipeflow provides a scalable alternative, `ScalablePipeline`, to allow variable assignments of pipes using range iterators. In Listing 1.2, we create a scalable pipeline, `pl`, from a vector of six pipes `p`. After the first run, we reset `pl` to another range of three pipes for the second run. A scalable pipeline is thus more flexible for applications to create a pipeline scheduling framework with dynamic structures.

Compared to the programming model of existing frameworks, such as oneTBB [3], Pipeflow's programming model has the following advantages: (1) Pipeflow is expressive and easy to write. Users only need pipe type, pipe callable, and the number of parallel lines to create a pipeline scheduling framework and explore the pipeline parallelism in their applications. Moreover, as Pipeflow does not provide data abstraction, users do not need to explicitly specify the input data and output data type at every pipe definition as oneTBB's users do. (2) Pipeflow is flexible. Users are able to modify the pipeline structure at runtime based on their specific needs.

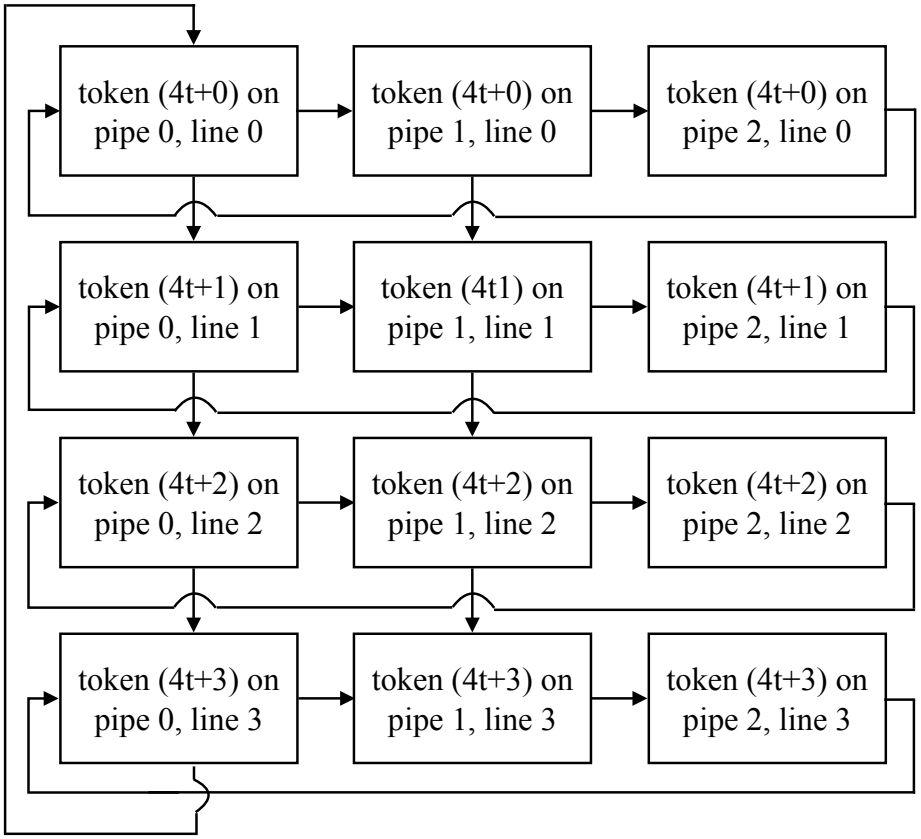


Figure 1.4: The scheduling diagram of the task-parallel pipeline in Listing 1.1. Each parallel line runs one scheduling token. Multiple parallel lines overlap tokens in a circular fashion. The text "token (4t+1) on pipe 1, line 1" means the token with ID `4t+1` runs on the pipe 1 and the parallel line 1.

Scheduling Algorithm

As Pipeflow does not touch data abstraction, we can simplify the pipeline scheduling problem to decide which scheduling token to run at which pipe and parallel line. Our scheduling algorithm places only one scheduling token per parallel line. We then process all tokens in a circular fashion across the given number of parallel lines. Figure 1.4 illustrates our pipeline scheduling idea using the pipeline in Listing 1.1. Since the pipeline sched-

ules tokens in a circular fashion, there are four edges (dependencies) from the last pipes (`pipe 2`) to the first pipes (`pipe 0`), and one edge from the first pipe of the last parallel line to the first pipe of the first parallel line. The last pipe (`pipe 2`) is a parallel type. There is no vertical edge between the last pipes of two consecutive parallel lines. Each parallel line runs only one scheduling token. Multiple parallel lines can overlap tokens whenever their dependencies are met. Even though the pipeline execution can involve many scheduling tokens, only four parallel lines are used in total.

Pseudocode

Based on the idea discussed in Section 1.4, we formulate each parallel line as a task, which defines a function object to run by a thread in the thread pool. Each task (1) deals with one scheduling token per parallel line and (2) decides which adjacent task to run on its next parallel line and pipe. Algorithm 1 implements such a task using efficient atomic operations. When a task is scheduled, we need to know which pipe at which parallel line for the scheduling token to work. We keep the parallel line and pipe information in a Pipeflow object. Each task owns a Pipeflow object `pf` of a specific parallel line (line 1). Once a scheduling token is done, there are two cases for its corresponding task to proceed: (1) for a parallel type, the task moves to the next pipe at the same parallel line; (2) for a serial type, the task additionally checks if it can move to the next parallel line. To carry out such a dependency constraint, each pipe keeps a join counter of an *atomic* integer to represent its dependency value. The values of a serial pipe and a parallel pipe can be up to 2 and 1, respectively. We create a 2D array `join_counters` to store the join counter of each pipe at each parallel line. Line 2 initializes these join counters to either 2 or 1 based on the corresponding pipe types that are enumerated on integer constants, 2 (serial) and 1 (parallel). At the first pipe (line 3), the Pipeflow object

updates its token number (line 4) and checks if the pipe callable requests to stop the pipeline (line 5:8). If continued, we increment the number of scheduled tokens by one (line 9). For other pipes, we simply invoke the pipe callables (line 11:13). After the pipe callable returns, we call `schedule_next_task` (line 14, define in Algorithm 2) to schedule the next task.

Algorithm 1: define_task(l)

```
   global: pipeflows: a vector of Pipeflow objects
   global: join_counters: a 2D array of join counters
   global: num_tokens: the number of tokens
   Input: l: a parallel line id
 1 pf ← pipeflows[l];
 2 AtomicStore(join_counters[pf.line][pf.pipe], pf.join_counter);
 3 if pf.pipe == 0 then
 4 |   pf.token ← num_tokens;
 5 |   invoke_pipe_callable(pf);
 6 |   if pf.stop == True then
 7 |   |   return;
 8 |   end
 9 |   num_tokens = num_tokens + 1;
10 end
11 if pf.pipe != 0 then
12 |   invoke_pipe_callable(pf);
13 end
14 schedule_next_task(pf);
```

Algorithm 2 implements how we schedule the next task after the pipe callable returns in line 12 in Algorithm 1. We update the join counters based on the pipe type and determine the next possible tasks to run (line 1:11). When the join counter of a pipe becomes 0, we bookmark this pipe as a task to run next (line 7 and line 10). If two tasks exist (line 12), the current task informs the scheduler to call a worker thread to run the task at the next parallel line (line 13) and reiterates itself on the next pipe (line 14). The idea here is to facilitate data locality as applications tend to deal with the next pipe at the same parallel line as soon as possible. If there is only one task available, the current task directly runs the next task with the updated `pf` object (line 16:21).

Algorithm 2: schedule_next_task(pf)

global: pipeflows: a vector of Pipeflow objects
global: join_counters: a 2D array of join counters
global: num_lines: the number of parallel lines
global: num_pipes: the number of pipes
Input: pf: a pipeflow object

```
1  curr_pipe ← pf.pipe;
2  next_pipe ← (pf.pipe + 1)%num_pipes;
3  next_line ← (pf.line + 1)%num_lines;
4  pf.pipe ← next_pipe;
5  next_tasks = {};
6  if curr_pipe is SERIAL and
     AtomicDecrement(join_counters[next_line][curr_pipe]) == 0
     then
7  |  next_tasks.insert(1);
8  end
9  if AtomicDecrement(join_counters[pf.line][next_pipe]) == 0
     then
10 |  next_tasks.insert(0);
11 end
12 if next_tasks.size == 2 then
13 |  call_scheduler(task_of_next_line);
14 |  goto Line 2 in Algorithm 1;
15 end
16 if next_tasks.size == 1 then
17 |  if next_tasks[0] == 1 then
18 |  |  pf ← pipeflows[next_line];
19 |  end
20 |  goto Line 2 in Algorithm 1;
21 end
```

Compared to existing algorithms, such as oneTBB [3], that count on non-trivial synchronization between tasks and internal data buffers, our algorithm focuses on the task parallelism of a pipeline itself. This design largely reduces the scheduling complexity of pipeline by using lightweight atomic operations without complex data buffer management.

Proof

We draw the following lemmas and sketch their proofs to highlight the correctness of our scheduling algorithm:

Lemma 1.1. *Only one task runs a pipe callable* (*line 5 and line 12 in Algorithm 1*) *on a scheduling token.*

Proof. Assume two tasks are running the same pipe callable, which means one task reiterates its execution from the previous pipe, and the other task comes from the previous parallel line. This is not possible in a parallel pipe as there is no dependency from the previous parallel line; only one runtime task decrements the join counter to 0 (line 9 in Algorithm 2). Take Figure 1.4 for example. There is no vertical edge pointing to token `4t+1` from token `4t+0` for pipe `2`. Only the task that runs token `4t+1` on pipe `1` gets to decrement the join counter for task `4t+1` on pipe `2`. In a serial pipe, this is also not possible because the dependency is resolved using atomic operations; only one task will acquire the zero value of the join counter (line 6 in Algorithm 2). For example, in Figure 1.4, either the task running token `4t+0` on pipe `1` or the task running token `4t+1` on pipe `0` decrements token `4t+1` on pipe `1` to zero and then runs the pipe. □

Lemma 1.2. *The scheduler does not miss any pipe.*

Proof. We consider the situation where one task moves to the next parallel line (line 18 in Algorithm 2) instead of the next pipe at the same parallel line. Under this circumstance, we need to make sure one task will run

that next pipe. Take Figure 1.4 for example. Suppose a task finishes token `4t+1` at pipe `0` and precedes to token `4t+2` at pipe `0`, meaning that the join counter of token `4t+1` at pipe `1` is not 0 yet. Another task that works on token `4t+0` at pipe `1` will eventually decrement the join counter to run it (line 14 in Algorithm 2) or invoke another worker thread to run it (line 13 in Algorithm 2). □

1.5 Experimental Results

We implemented Pipeflow using C++17 and evaluated the performance of Pipeflow on a micro-benchmark and a real-world industrial CAD application. We studied the performance across memory (RSS), runtime, and throughput. We did not use conventional pipeline benchmarks (e.g., PARSEC's ferret [12] has only six pipes, loading, segmentation, extraction, indexing, ranking, and output) as their problem sizes are relatively small compared to CAD, and the runtime difference between Pipeflow and the baseline is not obvious on small pipelines. We compiled all programs using g++12 with `-std=c++17` and `-O3` enabled. We ran all the experiments on a Ubuntu Linux 19.10 (Eoan Ermine) machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz and 256 GB RAM. All data is an average of ten runs.

Baseline

Given a large number of pipeline programming frameworks, it is infeasible to compare Pipeflow with all of them. We considered oneTBB [3] as our baseline for two reasons. First, oneTBB is the only library that provides a single pipeline API for users to explore pipeline parallelism. Others require combining several library-specific constructs to achieve this goal. For example, in Cilk-P users need `pipe_while`, `pipe_stage`, and `pipe_stage_wait` to add pipeline parallelism in applications. Second,

Pipeflow is inspired by our CAD applications, and oneTBB is widely used in the CAD community due to its absolute speed and robustness. For a fair comparison, we implemented the same work-stealing strategy as oneTBB in our thread pool, in particular, `call_scheduler` in line 13 in Algorithm 2.

Micro-benchmark

The purpose of micro-benchmarks is to measure the pure scheduling performance without much computation bias from applications. We compared the memory, runtime, and throughput between Pipeflow and oneTBB for completing pipelines of different numbers of serial pipes, scheduling tokens, and threads. We did not use parallel pipes as their callable can be absorbed into the previous serial pipe. Each pipe performs a nominal work of constant space and time complexity (e.g., small matrix multiplications) and forwards a scheduling token to the next pipe.

Figure 1.5 illustrates the maximum RSS between Pipeflow and oneTBB with different scheduling tokens and threads. The number of parallel lines and pipes of a pipeline is equal to the number of threads. We can see that oneTBB starts to consume more memory than Pipeflow as we increase the pipeline size. For example, with 2^{10} scheduling tokens Pipeflow needs 1.97% and 11.68% less memory than oneTBB when running with 16 and 64 threads, respectively. The same trend is also observed in the plot of processing 2^{15} scheduling tokens. In terms of memory usage, oneTBB is consistently higher than Pipeflow (e.g., 97.72% higher with 64 threads and 2^{15} scheduling tokens) because we do not manage any data buffers but focus on the task scheduling itself. That is, oneTBB needs to allocate space for its internal data buffer structures to perform pipeline scheduling. We can see the overhead of using data abstractions in pipeline scheduling next.

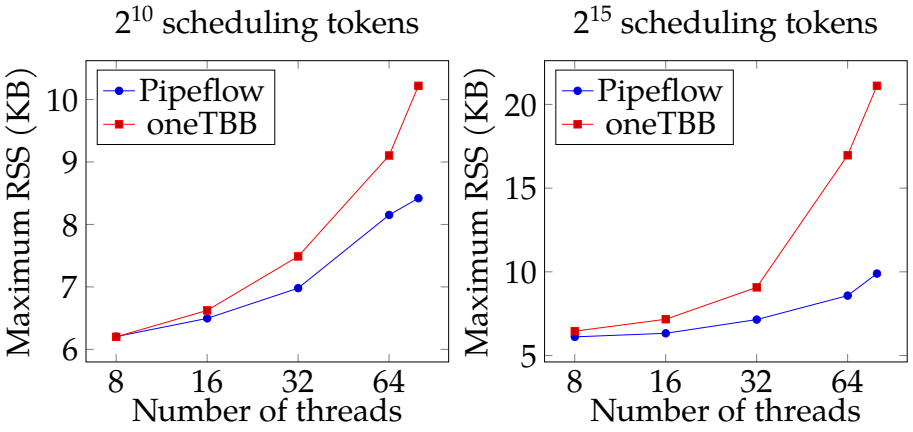


Figure 1.5: Maximum RSS comparison between Pipeflow and oneTBB with different threads and two scheduling tokens (2^{10} and 2^{15}) for the micro-benchmark. The number of threads is the same as the number of pipes in the pipeline.

Figure 1.6 draws the runtime comparisons between Pipeflow and oneTBB under different scheduling tokens and thread counts. The number of parallel lines and pipes of a pipeline is equal to the number of threads. We can see that the runtime gap between Pipeflow and oneTBB starts to increase as we increase the pipeline size. For example, at 2^{15} scheduling tokens Pipeflow runs 10.13%, 10.98%, 124.18%, and 201.38% faster than oneTBB with 8, 16, 64, and 80 threads, respectively. Furthermore, Pipeflow has better runtime performance than oneTBB in all situations. We attribute the performance improvements of Pipeflow over oneTBB to the reason that oneTBB relies on its internal data buffer to perform pipeline scheduling while Pipeflow only uses lightweight atomic operations. Moreover, as the number of threads is equal to the number of parallel lines and pipes in the experiment, the pipeline running with 80 threads has a bigger structure than the pipeline running with 8 threads. As a result, the former pipeline exhibits higher task scheduling overhead than the latter and thus spends more time to finish. Although the micro-benchmark only demonstrates

the pure scheduling performance and forwards the scheduling token between pipes, the overhead of data abstraction design of oneTBB results in a significant runtime difference, especially in a large pipeline.

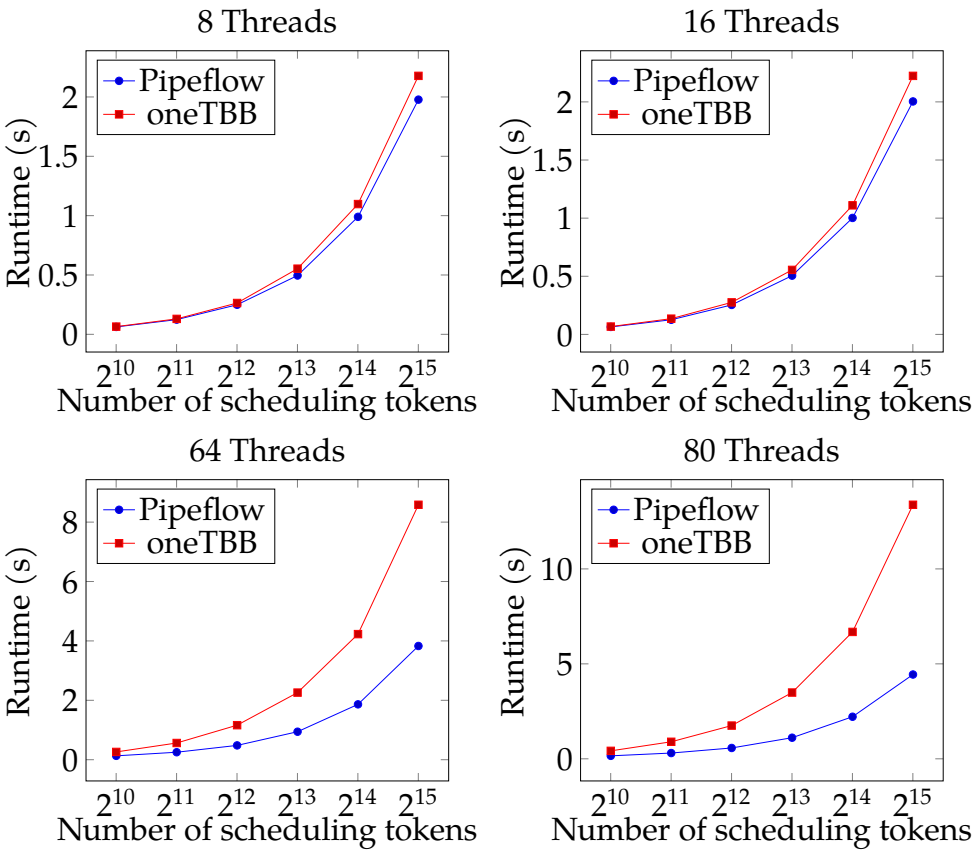


Figure 1.6: Runtime comparison between Pipeflow and oneTBB with different scheduling tokens and threads for the micro-benchmark. The number of threads is the same as the number of pipes in the pipeline.

Figure 1.7 compares the throughput by corunning the same program up to 8 times. Corunning a program at different configurations is very common in some applications, such as [92]. The experiment emulates a server-like environment where different pipeline applications compete for the same resources. We use the weighted speedup to measure the system

throughput, which is the sum of the individual speedup of each process over a baseline execution time [33]. A throughput of one implies that the corun throughput is the same as if those processes run consecutively. On the left plot, the pipeline has 16 pipes and 16 parallel lines and runs with 16 threads. On the right plot, the pipeline has 80 pipes and 80 parallel lines and runs with 80 threads. Both of them run 2^{15} scheduling tokens. We can see that Pipeflow outperforms oneTBB in all coruns. For example, at 8 coruns Pipeflow is 1.2x and 3.31x better than oneTBB with 16 and 80 pipes, respectively. Besides, Pipeflow remains around one up to 5 coruns while oneTBB decreases after 2 coruns. We attribute the finding to the reason that we use lightweight atomic operations rather than complex data buffer synchronizations to do the task scheduling. As Pipeflow is more lightweight than oneTBB in pipeline scheduling, corunning Pipeflow thus has better throughput than corunning oneTBB.

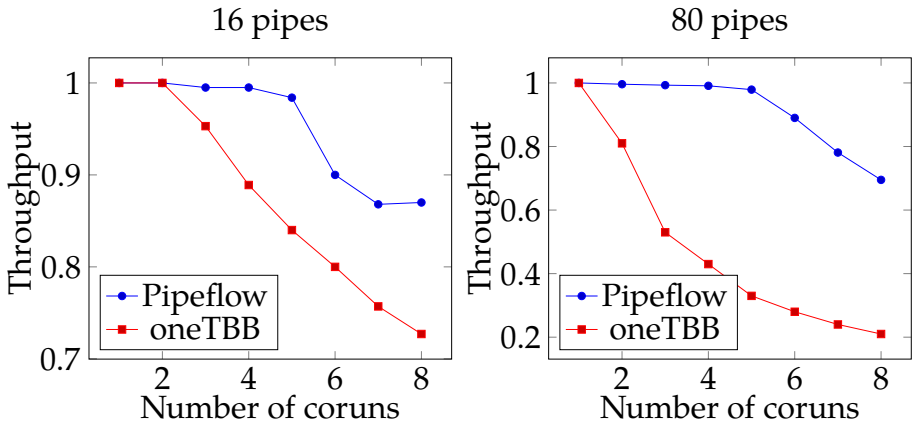


Figure 1.7: Throughput of corunning micro-benchmark programs with 16 and 80 pipes and 2^{15} scheduling tokens.

The above experiments were running with the configuration in which the number of pipes is the same as the number of threads. However, this configuration does not always guarantee the best runtime performance because of different applications and hardware environments. Next, we

see how the number of threads could impact the performance. Figure 1.8 shows the impact of Pipeflow and oneTBB running with different numbers of threads in a small (16 pipes) and a big (80 pipes) pipeline in which 2^{10} and 2^{15} tokens were scheduled. For 16 pipes, we can see that the runtime trends of running 2^{10} and 2^{15} are similar. oneTBB has the best runtime performance with 32 threads; Pipeflow has the best performance at 16 threads. For 80 pipes, the runtime trends are the same. We find out that Pipeflow has the best performance with 80 threads while oneTBB with 32 threads. In this micro-benchmark, we know the selection of identical pipes and threads is the best option for Pipeflow but not for oneTBB. Selecting the best thread number is an important factor and the best thread number for one application may not be the best choice for another application.

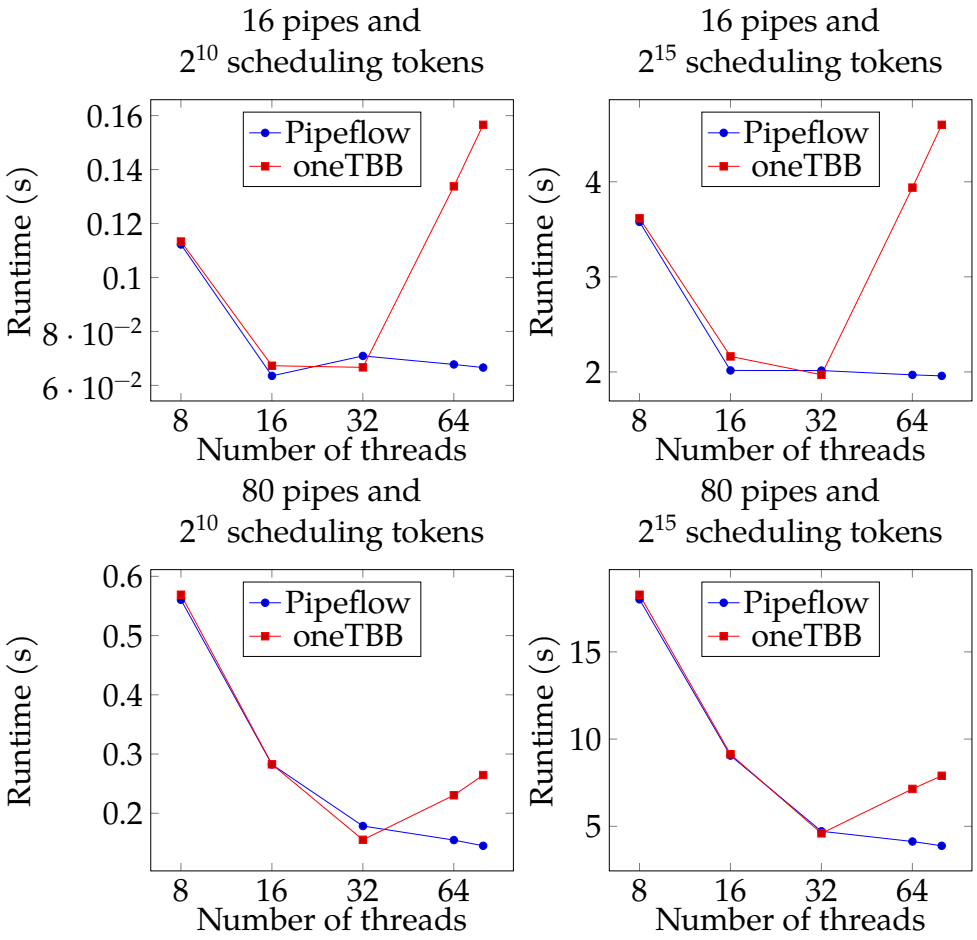


Figure 1.8: Impacts of selecting the number of threads on the runtime performance for the micro-benchmark. The number of pipes is not identical to the number of threads. The numbers of pipes are 16 and 80. The pipeline processes 2^{10} and 2^{15} scheduling tokens.

VLSI Static Timing Analysis Algorithm

We applied Pipeflow to solve a large-scale VLSI static timing analysis (STA) problem. The goal is to analyze the timing landscape of a circuit design and report critical paths that do not meet the given constraints (e.g., setup and hold). As presented in Figure 1.3, modern STA engines

leverage pipeline parallelism to speed up the timing propagations. However, nearly all of them count on OpenMP-based loop parallelism with layer-by-layer synchronization [75]. With Pipeflow, we can directly formulate the problem as a task-parallel pipeline to improve task asynchrony. As the analysis complexity continues to increase, more analysis tasks (e.g., RC, delay calculators, pessimism reduction) are incorporated into each node in the STA graph. These tasks can be encapsulated in a sequence of pipe functions to overlap in the graph across parallel lines. We modified a large circuit design of 1.5 million nodes and 3.5 million edges from [9, 75] and studied the performance under different pipe counts. Each node has a pipe task to calculate delay values at a specific configuration using linear interpolation. We levelized the STA graph and ran the nodes at the same level in parallel, such that different analysis tasks overlaps across different levels using pipeline parallelism, as illustrated in Figure 1.3.

Figure 1.9 evaluates the memory usage between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts. The number of pipes and lines in the pipeline is identical to the number of threads. As the pipeline size grows, the gap of memory usage starts to increase in both 1.5 million and 5 million graph sizes. For instance, with 1.5 million graph size Pipeflow needs 0.07% and 5.6% less than oneTBB at 32 and 80 threads, respectively. We can observe a similar trend when we process 5 million graph size. Since Pipeflow delegates data management directly to applications without touching data abstractions, Pipeflow does not allocate as much memory as oneTBB to perform pipeline scheduling.

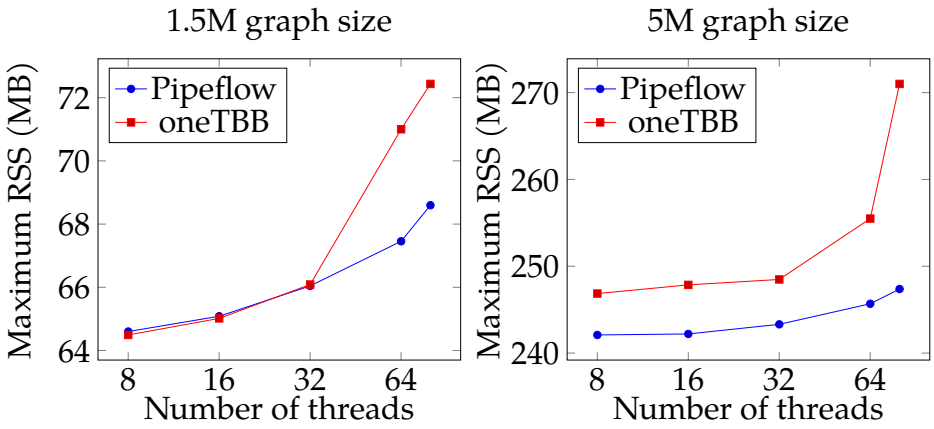


Figure 1.9: Maximum RSS comparison between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts for the timing analysis workload. The number of threads is identical to the number of pipes in the pipeline.

Figure 1.10 compares the runtime performance between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts. The number of pipes and parallel lines of the pipeline is the same as the number of threads in this experiment. We can see that Pipeflow outperforms oneTBB when we increase the graph size. Taking 16 threads for example, Pipeflow runs 62.02%, 57.44%, and 46.08% faster than oneTBB with 1, 3, and 5 million graph size, respectively. The performance improvements reduce because the overhead of setting up internal data buffers in oneTBB is gradually amortized when we increase the graph size in this workload. We also notice the runtime gap decreases when we use more threads. For example, at 5 million graph size Pipeflow is 110.33%, 46.08%, and 20.08% faster with 8, 16, and 64 threads, respectively. The improvements also reduce because the cost of data buffers is amortized gradually as the pipeline size grows. Despite the runtime improvements gradually decrease when the pipeline size grows or the graph size increases, Pipeflow still outperforms oneTBB in all cases in the workload. Since our scheduling

algorithm does not deal with data passing between pipes, we can process scheduling tokens more efficiently than oneTBB. In Pipeflow all pipe tasks perform computations directly on a global graph data structure captured in the pipe callable instead of passing data between successive pipes using buffers. The data passing interface between successive pipes in oneTBB thus becomes an unnecessary overhead.

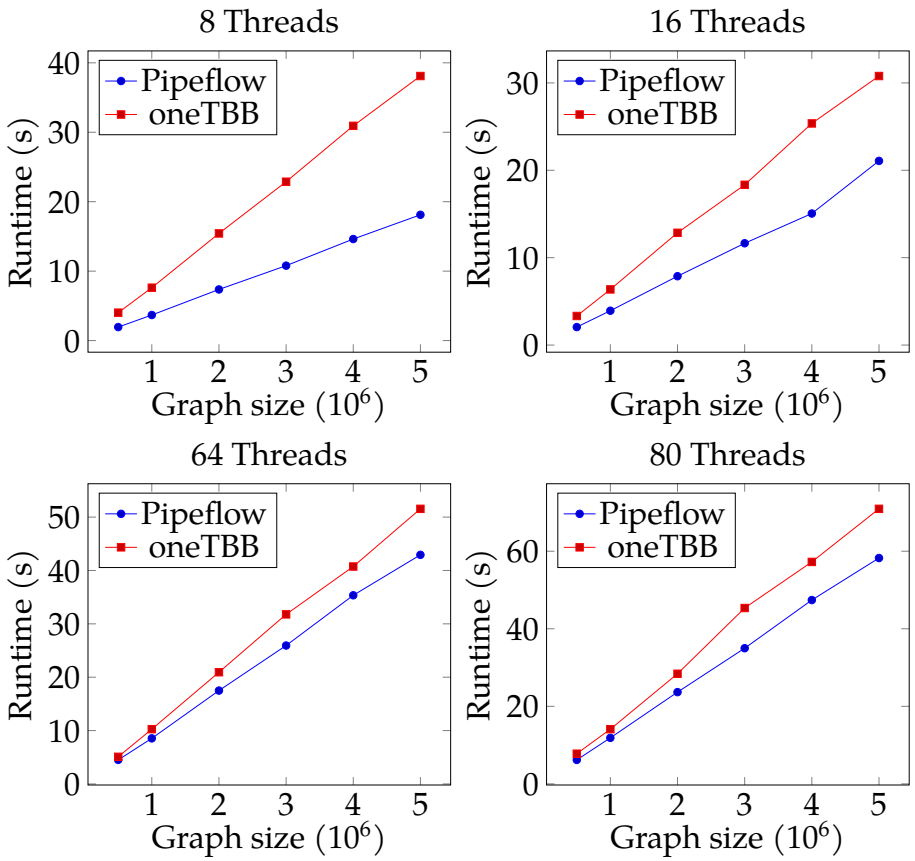


Figure 1.10: Runtime comparison between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts for the timing analysis workload. The number of threads is identical to the number of pipes in the pipeline.

Next, we compare the throughput by corunning the same program

up to 8 times. Corunning the STA program is very common for reporting the timing data of a design at different input library files [92]. The effect of pipeline scheduling propagates to all simultaneous processes. Hence, throughput is a good measurement for the inter-operability of a pipeline-based STA algorithm. We corun the same analysis program up to 8 processes that compete for 40 cores. Again, we use the weighted speedup to measure the throughput. Figure 1.11 plots the throughput across 8 coruns at 16 and 80 pipes. The number of pipes is identical to the number of threads. We can see that Pipeflow outperforms oneTBB at all coruns. For instance, at 8 coruns Pipeflow is 1.04x and 1.14x better than oneTBB with 16 and 80 pipes, respectively. This is because Pipeflow leverages lightweight atomic operations and oneTBB relies on complex data buffer management in pipeline scheduling. Corunning a lightweight program has a higher throughput than corunning a heavy program. Besides, with more coruns both Pipeflow and oneTBB have a decreasing throughput.

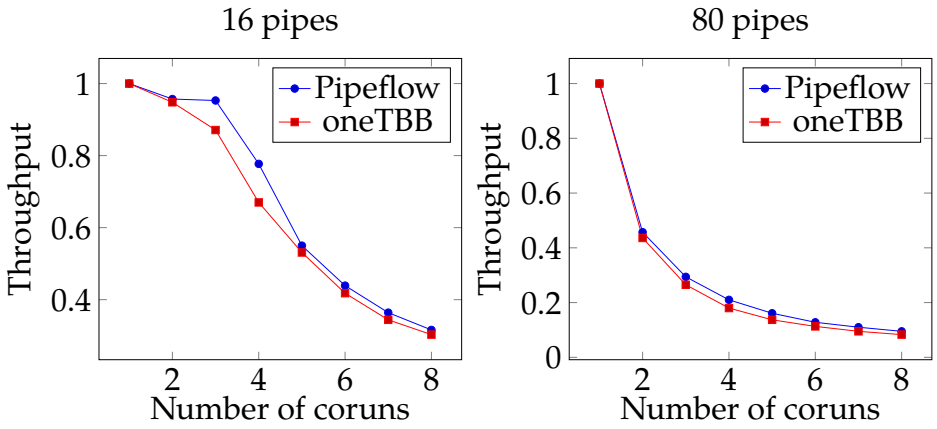


Figure 1.11: Throughput of corunning STA programs with 16 and 80 pipes and 1.5 million graph size($\|V\| + \|E\|$).

So far, we ran the experiments of this workload using the number of threads same as the number of pipes. This configuration may not

always give us the best runtime performance because of different hardware environment and workloads. Next, we demonstrate the importance of selecting the number of threads in this workload. Figure 1.12 shows the runtime performance of Pipeflow and oneTBB processing 1.5 million and 5 million graph size with different numbers of threads in 16-pipe and 80-pipe pipelines. For 16 pipes, we observe that both Pipeflow and oneTBB have a similar runtime trend, and both achieve the best performance with 64 threads for 1.5 million graph size. With 5 million graph size Pipeflow has the best performance with 80 threads while oneTBB with 64 threads. For 80 pipes, both Pipeflow and oneTBB have the best performance with 32 threads. From Figure 1.12 we learn that the alignment of threads and pipes does not achieve the best runtime performance in the workload. Hence, selecting the thread counts is an important factor while exploring pipeline parallelism in applications.

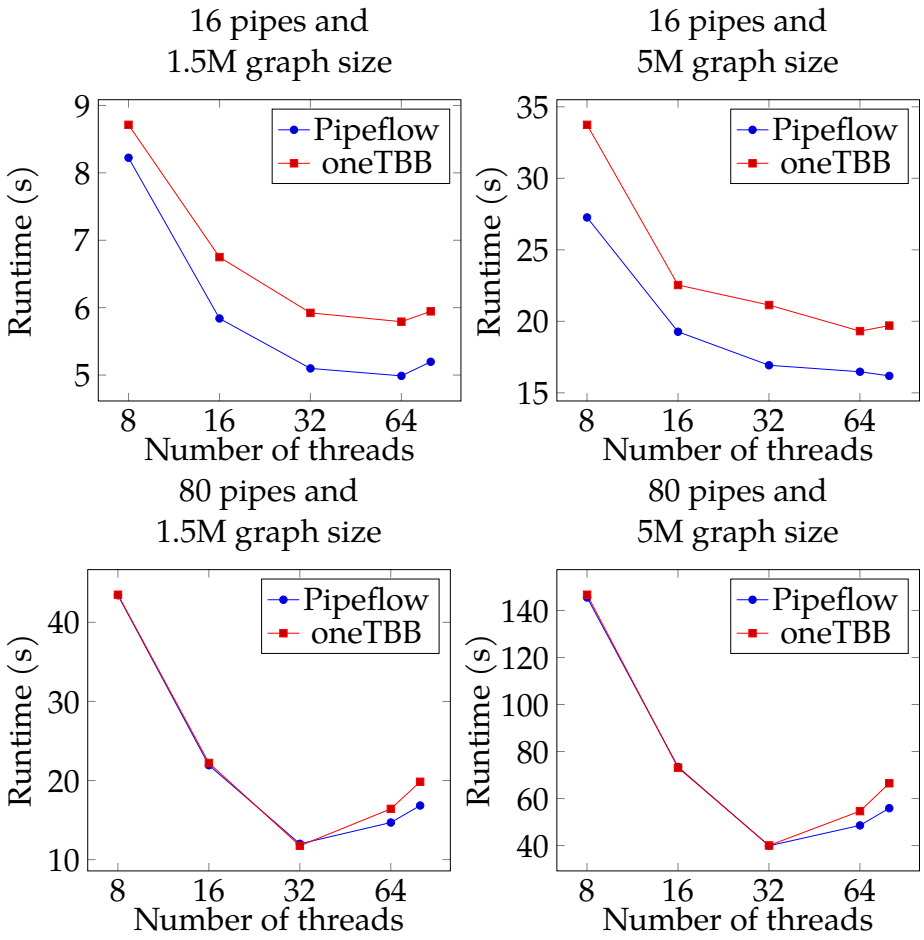


Figure 1.12: Impacts of selecting the number of threads on the runtime performance for the timing analysis workload. The number of pipes is not identical to the number of threads. The numbers of pipes are 16 and 80. The graph sizes ($\|V\| + \|E\|$) are 1.5 million and 5 million.

Importance of Task-Parallel Pipeline

As experienced parallel CAD researchers, Pipeflow has assisted us in overcoming many programming challenges. For example, in the previous experiments, the data is explicitly managed by the application algorithms and there is no need to go through any data abstraction. The real need is a

task-parallel pipeline programming framework that (1) gives applications full control over data and (2) allows applications to probe each scheduled task. For instance, when implementing the STA algorithm, we captured the data from a global STA graph structure in each pipe callable and used the `pipeflow` variable to get the parallel line number of a scheduled task to index the corresponding entry in a result vector. However, oneTBB abstracts these components out, and we have to implement another mapping strategy to get these data from each filter, both of which incur significant yet unnecessary runtime overheads. Similar situations exist in other libraries too.

Selection of the Number of Parallel Lines

Selecting the number of parallel lines (or threads) for the best performance is application-dependent. For example, Figure 1.8 illustrates that Pipeflow obtains the best performance while aligning the pipe sizes and thread counts and oneTBB should run with 32 threads in the micro-benchmark. From Figure 1.12 when running the STA workload, both Pipeflow and oneTBB obtain the best runtime results with 32 threads in an 80-pipe pipeline. From the micro-benchmark and STA algorithm, we learn that the selection of the number of parallel lines (or threads) is a critical factor regarding the runtime performance. Moreover, as the performance of an application tends to saturate or peak at a certain limit, increasing the number of parallel lines exceeds the limit could negatively affect the runtime. As a result, Pipeflow makes the number of parallel lines a tunable parameter (similarly in oneTBB). Based on our experience, most applications can obtain decent performance when the number of parallel lines is equal to the number or twice the number of the cores.

1.6 Conclusion

In this chapter, we have introduced Pipeflow, an efficient task-parallel pipeline programming framework to explore pipeline parallelism in applications. We have introduced a simple yet efficient scheduling algorithm based on our work-stealing runtime with dynamic load balancing. We have evaluated the performance of Pipeflow on a micro-benchmark and an industrial application. For example, in a VLSI static timing analysis workload that adopts pipeline parallelism to speed up the runtime performance, the Pipeflow's implementation is up to 110.33% faster than the oneTBB's implementation. Our future plans are to (1) apply Pipeflow to other applications than CAD applications to bring interdisciplinary ideas to the parallel computing community and (2) extend Pipeflow to task-parallel GPU computing platforms [28, 29, 89, 115, 116, 117] and distributed environment [66, 69, 70].

2 A TASK-PARALLEL PIPELINE PROGRAMMING MODEL WITH TOKEN DEPENDENCY

2.1 Abstract

Task-parallel pipeline framework explores pipeline parallelism in applications and is critical in many parallel and heterogeneous areas, such as VLSI static timing analysis and data similarity search. However, existing solutions only deal with certain types of applications in which data dependency exists between preceding data and succeeding data in a forward direction. Some applications, such as video encoding, exhibit data dependency in both forward and backward directions and cannot be processed with existing solutions. To address the limitation, we introduce a token dependency-aware pipeline framework. Our framework associates each data element with a token as its identifier, supports explicit definitions of forward and backward token dependency with an expressive programming model, resolves token dependency using simple data structures, and schedules tokens with lightweight atomic counters. We have evaluated the framework on applications that exhibit both forward and backward token dependency. For example, our framework is 8.6% faster than PARSEC's implementation in x.264 video encoding applications.

2.2 Introduction

Task-parallel pipeline framework (TPF) explores pipeline parallelism in applications and plays a critical role in parallel and heterogeneous computing workloads, such as static timing analysis [8, 26, 31, 43, 45, 50, 63, 67, 72, 76, 81, 113, 154], data similarity search workload (ferret benchmark [12, 13]), quantum circuit simulation [91], and others [24, 89, 106, 115, 153, 154]. TPF models a pipeline application as a *task graph* that describes a function

call as a task and a functional dependency as an edge. Through task graph scheduling, pipeline parallelism arises when multiple tasks are scheduled and executed concurrently once the dependency constraints are met. As a result, recently Chiu et al. introduced the state-of-the-art task-parallel pipeline framework, Pipeflow [31] (or Chapter 1).

Although Pipeflow has demonstrated good runtime performance [31], we find out Pipeflow only deals with specific applications in which data dependencies exist between preceding data and succeeding data in a forward direction. However, some applications exhibit data dependency from succeeding data back to preceding data. For instance, in video encoding applications, frames would reference encoded frames to reduce stream size for online transmission [138]. In real-world applications, three frame types are employed, intra (I) frame, predicted (P) frame, and bi-directional (B) frame. I frames are encoded independently without reference to other frames. P frames require references from a preceding I frame. B frames require references from both a preceding and a succeeding (future) I or P frames. Figure 2.1 illustrates such frame dependency in a video encoding application. The presence of B frames introduces bi-directional dependency, where the encoding of a frame relies on information from both past and future frames. This characteristic poses a significant challenge for existing pipeline frameworks, including Pipeflow, which primarily focus on uni-directional dependency. As a result, Pipeflow cannot effectively schedule the encoding of B frames, limiting its applicability in real-world video encoding scenarios.

Figure 2.1: A sample dependency diagram in a video encoding application of an x.264 standard. Edges denote the dependencies between two frames. I denotes frames, P denotes predicted, B denotes bi-directional frames.

To handle the data dependency in both forward and backward directions, the most common way is to reorder the execution order of data using low-level synchronization primitive, *condition variable* [1], and then feed the reordered data to the pipeline framework as PRASEC does [12, 13]. However, we notice three limitations of this approach: (1) Manipulating *condition variable* requires a deep understanding of this low-level synchronization primitive from users and is error-prone when dependency is intricate. (2) The approach is not an end-to-end implementation as users need to additionally reorder the data outside the original pipeline application. (3) The implementation could encounter deadlock when the data dependency is complex and insufficient threads are spawned.

To overcome the limitations, we have associated each data element with a token as its identifier and introduced a new task-parallel pipeline framework with token dependency enabled on top of Pipeflow [31]. We summarize our technical contributions as follows:

- **New Programming Model.** We have developed a new programming model for applications to explicitly define generalized bi-directional token dependency. With our programming model, applications can easily specify the token dependency with a single and intuitive API and do not need to touch low-level synchronization primitives.

- **New Scheduling Algorithm.** We have designed a new scheduling algorithm to support our new programming model. Our scheduling algorithm leverages simple data structures to efficiently determine the execution order of tokens and to avoid potential deadlock when dealing with intricate token dependency and insufficient threads are spawned.

- **End-to-end Implementation.** We have integrated the step of reordering tokens into the pipeline to achieve an end-to-end implementation. With our seamless integration, users can eliminate the need for ex-

ternal token reordering mechanisms, simplifying the overall system design while providing an end-to-end solution for scheduling tokens with bi-directional dependency within the pipeline itself.

We have evaluated the framework on applications that exhibit both forward and backward token dependency. For example, our framework is 8.6% faster than PARSEC's implementation in x.264 video encoding applications.

2.3 Background

In this chapter, we will focus on token dependency and the state-of-the-art Pipeflow programming framework [31], rather than providing a comprehensive discussion of pipeline frameworks. For a broader discussion of other pipeline frameworks, readers are referred to [31] (or Chapter 1).

Token Dependency

Token dependency constrains the order in which tokens should execute in the pipeline and is defined by the applications, e.g., the x.264 application [12, 13]. A dependency exists between token t_1 and t_2 in which t_1 must complete before t_2 can begin. We categorize token dependency into two types: forward token dependency (FTD), which refers to dependency connecting from preceding to succeeding token, and backward token dependency (BTD), which refers to dependency in the opposite direction. Figure 2.2(a) shows a diagram in which all dependencies are FTDs, which are implicitly assumed in existing task-parallel pipeline framework. Since all dependencies are FTDs, there is no need to reorder the tokens to get the correct execution order. Figure 2.2(b) shows a diagram combining both FTDs and BTDs. As BTDs exist, we need to reorder the tokens to get

the correct execution order. For example, token 16 pointing to 7 and 12 are BTDs, we need to reorder 16 before 7 and 12.

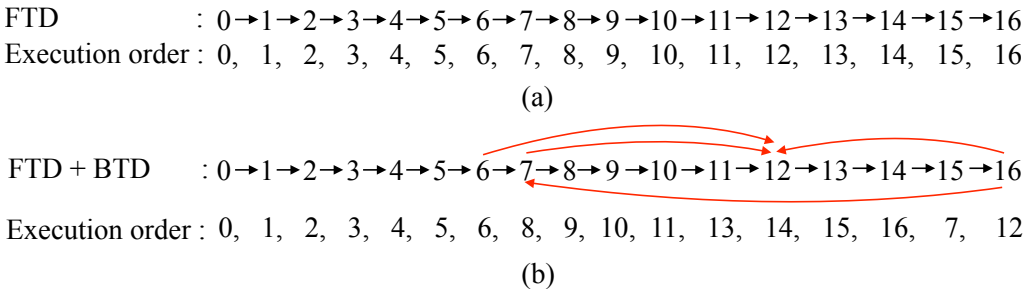


Figure 2.2: (a) The diagram of all FTDs and the corresponding execution order of tokens. (b) The diagram of both FTDs and BTDs and its corresponding execution order of tokens. Red edges pointing from token 6 and 7 to 12 are FTDs, and those from 16 to 7 and 12 are BTDs. Black edges are implicit dependencies and red ones are explicit dependencies. Execution order denotes the order in which the tokens should be executed.

To get the correct execution order of tokens when BTDs exist, existing frameworks adopt the *condition variable* primitive, such as PARSEC's pthread implementation, to first determine the execution order and then flow the reordered tokens through the pipeline. Figure 2.3 illustrates PARSEC's implementation using C++. Every token has its own condition variable `cv` and a mutex `mutex` and is handled by one thread. A token will wait on the `cv` of its dependent token until that dependent token finishes. For example, token 7 has a dependent token 16, meaning token 7 must wait until 16 finishes. 7 will acquire 16's `mutex` and wait on 16's `cv` until 16 finishes. Once token 16 finishes, token 7 can start the execution and then notify all the other waiting tokens that token 7 has finished.

The implementation of using *condition variable* is able to reorder the tokens whenever BTDs exist with higher data locality because a thread continues processing the same token before and after sleeping. However, we notice four challenges: (1) Manipulating *condition variable* requires a deep understanding of this low-level synchronization primitive from users.

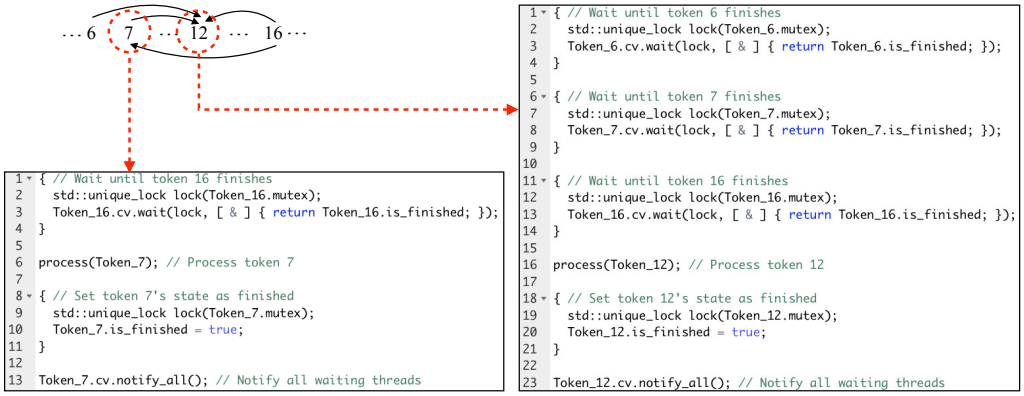


Figure 2.3: PARSEC's implementation of Figure 2.2(b) to reorder tokens using the *condition variable* primitive.

This can be particularly challenging for users, especially when dealing with complex token dependency, such as those involving tokens with multiple forward and backward dependencies (e.g., token 12 in Figure 2.3). (2) This solution is not an end-to-end implementation as users require manual token reordering outside the core pipeline execution logic. This introduces additional complexity and increases the risk of errors in the overall system. (3) The reliance on low-level synchronization primitives can increase the risk of deadlocks, especially in resource-constrained environments (insufficient threads) with complex dependency graphs. For instance, consider a scenario where token 6 has backward dependencies on tokens 1 through 5, and only 5 threads are available. These 5 threads are all waiting for token 6 to complete, a deadlock occurs, as no thread is available to process token 6. (4) It is not a lock free implementation. Every token has its own condition variable and a mutex lock. A thread has to acquire the mutex lock before accessing a token. In addition to deadlock, using locks can suffer from contention between threads, which increases overhead from the locking mechanism itself. As a result, we need a lock-free solution that is able to avoid deadlock when application's token dependency is complex and insufficient threads are spawned.

Pipeflow: Task-parallel Pipeline Framework

Pipeflow [31] represents a state-of-the-art task-parallel pipeline framework, offering significant advancements over prior work such as oneTBB [3]. By decoupling task scheduling from data abstraction, Pipeflow introduces an efficient scheduling algorithm that optimizes pipeline execution. Furthermore, Pipeflow provides an expressive programming model, simplifying the process of defining complex pipeline applications for developers.

Pipeflow models pipeline applications as circular task graphs, as depicted in Figure 2.4. This representation facilitates the scheduling of tasks across multiple parallel execution units. For instance, in a scenario with three parallel execution lines, Pipeflow can effectively schedule token 0 on parallel line 0, token 1 on parallel line 1, and token 2 on parallel line 2 and so on, adhering to the execution order obtained by the dependency illustrated in Figure 2.2(a).

Figure 2.4 illustrates the concept of parallel execution within the Pipeflow framework. In this example, the first and second pipeline stages (or pipes) are serial pipes, meaning tokens within these stages must execute sequentially. For instance, token q in pipe 0 and parallel line 1 cannot commence execution until token p in pipe 0 and parallel line 0 has completed. In contrast, the third pipe is a parallel pipe. This allows for concurrent processing of tokens within this stage. For example, token p in pipe 2 and parallel line 0 can execute concurrently with token q in pipe 2 and parallel line 1.

2.4 Our Framework

We introduce a new task-parallel pipeline framework with token dependency enabled shown in Figure 2.5. At a high level, we first determine the execution order of tokens and then flow the ordered tokens through the circular task graph. We provide an expressive programming model for

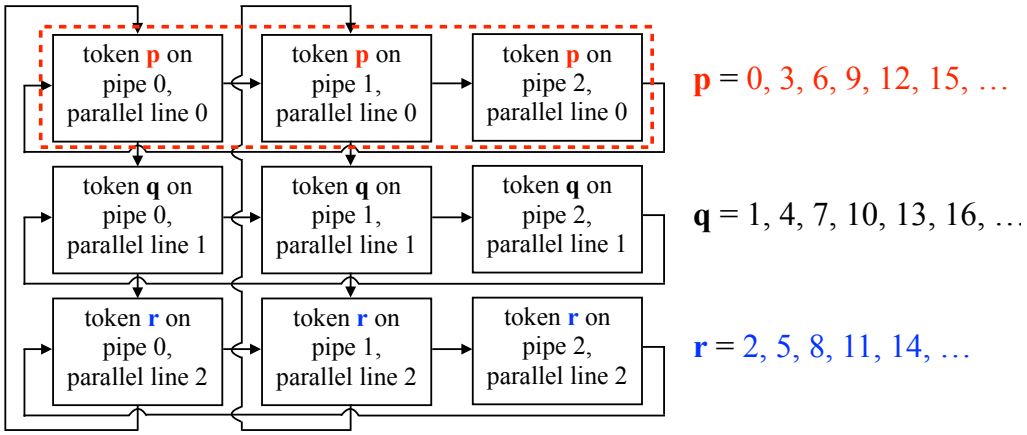


Figure 2.4: Pipeflow's circular task graph of an application in which every token is processed by a chain of 3 pipes (in the red dashed rectangle, referred to a `parallel line`) and up to 3 tokens can be processed concurrently. Edges denote dependencies.

users to explicitly express generalized bi-directional token dependency and provide a pipeline scheduling framework to support our programming model.

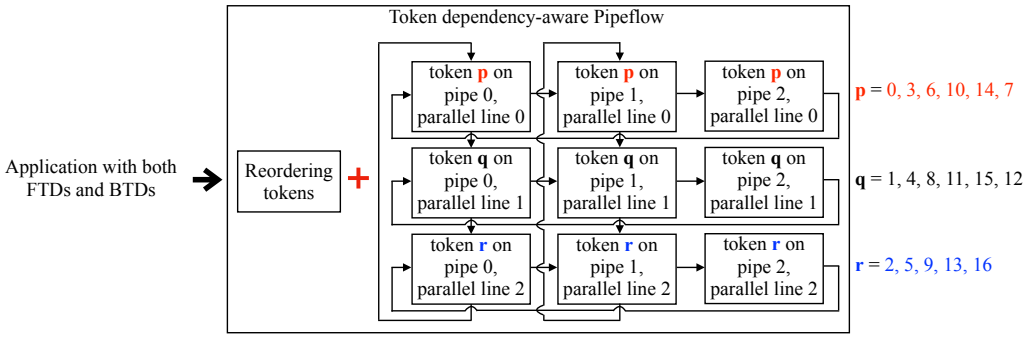


Figure 2.5: Overview of token dependency-aware Pipeflow running an application with both FTDs and BTDs, as illustrated in Figure 2.2(b). After determining the execution order of tokens, our framework schedules token 0, 3, 6, 10, 14, and 7 on parallel line 0 and so on.

Programming Model

We extend the Pipeflow framework [31] by introducing a novel programming model that explicitly supports bi-directional token dependency. Our new programming model leverages a two-tiered approach. Firstly, it retains Pipeflow's core structure for defining the pipeline structure. Secondly, we introduce a mechanism for explicitly defining token dependency within this framework. Listing 2.1 exemplifies an application of a serial-serial-parallel pipeline structure with token dependency in Figure 2.2(b). To define the pipeline structure, we follow Pipeflow's programming model. We use the API `Pipeline` to instantiate an object `pl` and define the pipeline structure. In the API, we specify the number of parallel lines and the abstract function of every pipe. For every pipe, we define the pipe type and a pipe callable using `Pipe`. We use `PipeType::SERIAL` to specify the type for the first pipe and the second pipe, and `PipeType::PARALLEL` for the third pipe. The pipe callable takes an argument `pf` which is used to query the status of a token that is executing the pipe callable. In this example, the first pipe stores a `float` in the buffer `buffer`, the second pipe stores a `string` in `buffer`, and the third pipe prints the value.

The second part of our new programming model is to specify the token dependency. To achieve an end-to-end implementation, we integrate this step into the pipeline by explicitly specifying the token dependency at the first pipe before the tokens flows to the pipes. To specify token dependency, we first use the number returned by `Pipeflow::num_deferrals` to start defining the token dependency. Initially, all tokens have zero `num_deferrals`. Then we specify token 12's three dependencies and token 7's dependency using `Pipeflow::defer`, respectively. For tokens that do not have dependencies or tokens whose execution orders are determined can resume execution, we define the corresponding function. Finally, we call `run` to submit the object `pl` to a runtime and execute it.

```
std::variant<float, std::string> data_type;
```

```
std::array<data_type, number_lines> buffer;

Pipeline pl(3,
  // Define the first pipe
  Pipe{PipeType::SERIAL,
    [&](Pipeflow& pf) {
      // Stop when 100 tokens are done
      if ( pf.token() == 100) {
        pf.stop();
      }
      else {
        if (pf.num_deferrals() == 0) {
          // Specify token 12's dependencies
          if (pf.token() == 12) {
            pf.defer(6);
            pf.defer(7);
            pf.defer(16);
          }
          // Specify token 7's dependency
          else if (pf.token() == 7) {
            pf.defer(16);
          }
          else {
            // Save a float in buffer
            buffer[pf.line()] = 0.0f;
          }
        }
        else {
          // Save a float in buffer
          buffer[pf.line()] = 0.0f;
        }
      }
    }
  },
  // Define the second pipe
  Pipe{PipeType::SERIAL,
    [&](Pipeflow& pf) {
```

```
      // Save a string in buffer
      buffer[pf.line()] =
        make_string(std::get<0>(buffer[pf.line()]));
    }
  },
  // Define the third pipe
  Pipe{PipeType::PARALLEL,
    [&](Pipeflow& pf) {
      // Print the string stored in buffer
      std::cout << std::get<1>(buffer[pf.line()]);
    }
  }
);
// Execute the pipeline
pl.run();
```

Listing 2.1: The implementation of a pipeline application consisting of 3 parallel lines and 3 pipes with a serial-serial-parallel type. The token dependency for the application is shown in Figure 2.2(b). Assume the first pipe stores a float in `buffer`, the second pipe stores a string in `buffer`, and the third pipe prints the value.

In contrast to existing approaches, such as PARSEC [12, 13], our framework eliminates the need for low-level synchronization primitives like *condition variables*, significantly simplifying dependency management. This not only reduces development complexity but also improves developer productivity by minimizing the risk of errors. Our framework introduces a concise and intuitive API for defining dependency, `pf.defer`. For example, specifying the dependencies for token 12 requires only three lines of code in Listing 2.1, compared to the 23 lines of code required in the PARSEC implementation (see Figure 2.3) for handling same dependencies. This significant reduction in code complexity simplifies debugging and maintenance, particularly in scenarios involving complex dependency graphs.

Scheduling Algorithm

To support our programming model, we design a new scheduling algorithm which includes two components: (1) Determining the correct execution order of tokens. (2) Scheduling the reordered tokens in the pipeline.

(1) Determining the Correct Execution Order of Tokens. The first part of our scheduling algorithm is to determine the correct execution order of tokens. The idea is to defer the execution of a token with unresolved token dependency and save the token until its dependency is resolved. Once that token becomes ready, we run it as soon as possible in order to resolve possible dependency for other tokens. To realize the idea, we use three data structures,

- `deferred_tokens` (`DT`): An associative container that stores deferred tokens and their respective dependencies. For example, in Figure 2.2(b), token 7 has one dependent token 16, meaning 16 must reorder before 7. We consider token 7 as a deferred token and represent the relationship as the entry {key:7, value:16} in `DT`.

- `token_dependencies` (`TD`): Another associative container that stores the reverse mapping of dependencies. For example, for the dependency between token 7 and 16 in Figure 2.2(b) `TD` stores {key:16, value:7}, allowing for efficient identification of tokens that depend on a given token. This enables rapid updates to `DT` when a dependency is resolved. Specifically, once token 16's order is determined later, we can quickly use the value obtained at `TD[16]`, which is token 7, to locate and remove the entry at `DT[7]`.

- `ready_tokens` (`RT`): A queue that stores tokens whose dependencies have been resolved and are ready for execution.

Figure 2.6 visualizes how we use the three data structures to determine the correct execution order of tokens in Figure 2.2(b). In (a), token 0 to 6 do not have BTDs and we put them in the `EST` list in order. In (b), token 7 has a dependent token 16 because of `7.defer(16)`. We first insert {key:7, value: 16} in `DT`, meaning 7 needs to be reordered after 16. Then we insert {key:16, value:7} in `TD` in order to locate 7 in `DT` quickly. `EST` does not have 7 as its' execution order is not yet decided. In (c), token 8 to 11 do not have BTDs and appear in `EST`. Token 12 has three dependencies, token 6, 7, and 16. As token 6's order has been determined in (a), we only insert {key:12, value:{7, 16}} in `DT` and then update `TD` to reflect the new dependency. In (d), token 13 to 15 do not have BTDs and appear in `EST`. In (e), token 16 does not have BTDs and appears in `EST`. As `TD[16]` exists, we use the value of that entry, token 7 and 12, to locate the corresponding entry in `DT`. Then we resolve 16's related dependency by deleting 16 from `DT[7]` and `DT[12]`. As a result, `DT[7]` is empty, meaning 7's order can be determined and we insert it in `RT`. In (f), `RT` is not empty and we append token 7 in `EST`. Next, `TD[7]` has token 12, and we directly use 12 to locate `DT[12]` and delete 7 from that entry. As a result, `DT[12]` is empty, meaning 12's order can be determined and be inserted in `RT`. In (g), we find 12 in `RT` and then append it in `EST`. In the end, we obtain the correct execution order of tokens as shown in Figure 2.2(b) using only three simple data structures, `DT`, `TD`, and `RT`.

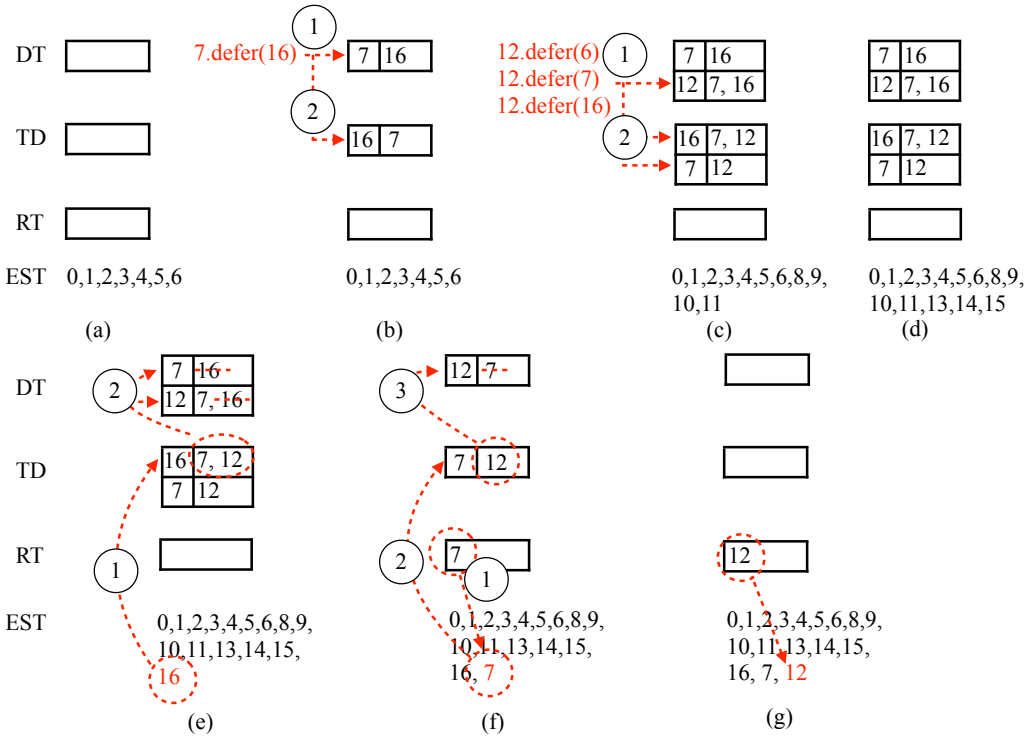


Figure 2.6: Visualization of how `DT`, `TD`, and `RT` determine the correct execution order of tokens with the token dependency in Figure 2.2(b). `EST` denotes the execution order of tokens and is used for illustration. We simplify `pf.defer(16)` in line 15 in Listing 2.1 to `7.defer(16)` for explanation purposes. The encircled numbers denote the operation sequence in each sub-figure.

(2) Scheduling Reordered Tokens in the Pipeline. After determining the correct execution order of tokens, we schedule the reordered tokens over the circular task graph as Pipeflow does [31]. We place one token per parallel line and schedule all tokens in a circular way across all parallel lines. As a result, we schedule the reordered tokens as shown in Figure 2.5. For example, we schedule token $0, 3, 6, 10, 14, 7$ over parallel line 0 and so on.

Pseudocode

To implement the algorithm discussed in Section 2.4, we formulate each parallel line as a task, which defines a function object to run by a thread in the thread pool. Each task (1) determines the execution order of tokens at the first pipe, (2) deals with one scheduling token per parallel line, and (3) decides which adjacent task to run on its next parallel line and pipe. Algorithm 3 implements such a task. When a task is to be scheduled, we must know which pipe and which parallel line for the token to run at. Each task owns an object `pf` of a specific line `l` (line 1). Once a token is done at a pipe, there are two cases for its corresponding task to proceed: (1) for a parallel pipe, the task moves to the next pipe at the same parallel line; (2) for a serial pipe, the task additionally checks if it can move to the next parallel line. For example, in Figure 2.5 when a thread finishes token p at pipe 1 and parallel line 0, we check whether the thread goes to the pipe 2 at the same parallel line 0 or pipe 1 at the adjacent parallel line 1. To carry out such a task dependency constraint, each pipe keeps a join counter of an *atomic* integer to represent its dependency value. The values of a serial pipe and a parallel pipe can be up to 2 and 1, respectively. We create a 2D array `join_counters` to store the join counter of each pipe at each parallel line. Line 2 initializes these join counters to either 2 (serial) or 1 (parallel) based on the corresponding pipe types. At the first pipe (line 3), a task either takes a ready token (i.e., a token whose order has been determined) (line 4:7) or a new token (line 8:11), and then invokes the pipe callable on that token (line 12). Then, we increment the number of processed tokens `num_tokens` by one if the task processes a new token (line 13:15).

When a task finds the token has dependency (line 16:18), we call `with_dependents` (line 17, defined in Algorithm 4) to keep valid dependents and remove invalid ones. After a token finishes at the first pipe, we call `resolve_token_dependencies` (line 20, defined in Algorithm 7)

to resolve its associated token dependency up to `longest_deferral` (line 19:21). `longest_deferral` keeps track of the biggest deferred token ID and is used to avoid redundant invocations of `resolve_token_dependencies`. For example, there is no need to invoke `resolve_token_dependencies` for applications that do not exhibit BTDs, such as Figure 2.2(a). At the first pipe (line 3:22), we perform the above operations. For other pipes, we simply invoke the pipe callables (line 23:25). After the pipe callable returns, we call `schedule_tasks(pf)` (line 26, defined in Algorithm 8) to determine the next possible tasks to run.

Algorithm 3: build_tasks(l)

```
   global: tasks: an array of tasks
   global: ready_tokens: a queue of ready tokens
   global: join_counters: a two-dimensional array of join counters
   global: longest_deferral: an integer of longest deferral
   global: num_tokens: the number of processed tokens
   Input: l: a parallel line id
 1 pf ← tasks[l];
 2 AtomicStore(join_counters[pf.line][pf.pipe], join_counter_of_pf.type);

 3 if pf.pipe == 0 then
 4     if ready_tokens.empty() == false then
 5         pf.token ← ready_tokens.front();
 6         ready_tokens.pop();
 7     end
 8     else
 9         pf.token ← num_tokens;
10         pf.num_deferrals ← 0;
11     end
12     invoke_pipe_callable(pf);
13     if pf.token == num_tokens then
14         Increment(num_tokens);
15     end
16     if pf.dependents.empty() == false then
17         with_dependents(pf);
18     end
19     if pf.token ⩽ longest_deferral then
20         resolve_token_dependencies(pf);
21     end
22 end
23 else
24     invoke_pipe_callable(pf);
25 end
26 schedule_tasks(pf);
```

Algorithm 4 shows the implementation of `with_dependents` in line 17 in Algorithm 3 to check if a token has valid dependents and construct deferred tokens. When a task finds the token has dependency, we call `check_dependents` (line 1, defined in Algorithm 5) to keep valid dependents and remove invalid ones. Invalid dependents refer to tokens whose order has been determined. If the token still has dependents after the first check (line 2:5), we construct it as a deferred token (line 3, defined in Algorithm 6) and reiterate the task with another token (line 4). If the token has no dependent, we reiterate the task with the same token (line 6:8).

Algorithm 4: with_dependents(pf)

Input: pf: a pipeflow object

```
1 check_dependents(pf);
2 if pf.dependents.empty() == false then
3 |   construct_deferred_tokens(pf);
4 |   goto Line 2 in Algorithm 3;
5 end
6 else
7 |   goto Line 12 in Algorithm 3;
8 end
```

Algorithm 5 shows the implementation of `check_dependents` in line 1 in Algorithm 4 to check if a token has valid dependents. We increment the number of deferrals of that token to track how many times this token has been deferred (line 1). We iterate the token's dependents to check the validity for two cases (line 2:15). Firstly, the dependent whose ID is bigger than the number of processed tokens (`num_tokens`) is a future token and thus is valid. We insert `pf.token` in `token_dependencies[dep]` (line 4) and update `longest_deferral` (line 5). Secondly, the dependent that is a deferred token is valid, and we insert the corresponding entry in

`token_dependencies` (line 8:10). The remaining dependents are invalid and are removed from the token's dependents (line 11:13).

Algorithm 5: check_dependents(pf)

```
   global: token_dependencies: a hashmap of a token and the
           deferred tokens
   global: longest_deferral: an integer of longest deferral
   global: num_tokens: the number of processed tokens
   Input: pf: a pipeflow object
 1 Increment(pf.num_deferrals);
 2 for dep ∈ pf.dependents do
 3     if num_tokens ⩽ dep then
 4         token_dependencies[dep].push(pf.token);
 5         longest_deferral ← max(longest_deferral, dep);
 6     end
 7     else
 8         if dep ∈ deferred_tokens then
 9             token_dependencies[dep].push(pf.token);
10         end
11         else
12             pf.dependents.erase(dep);
13         end
14     end
15 end
```

Algorithm 6 shows the construction of a deferred token in line 3 in Algorithm 4. For a deferred token, we insert the key-value pair in `deferred_tokens`, where the key is the token ID, and the value includes the token's ID, `num_deferrals`, and its `dependents`.

Algorithm 6: construct_deferred_tokens(pf)

global: deferred_tokens: a hashmap of a token and its deferred object

Input: pf: a pipeflow object

```
1 deferred_tokens[pf.token].token ← pf.token;
2 deferred_tokens[pf.token].num_deferrals ←
   pf.num_deferrals;
3 deferred_tokens[pf.token].dependents ← pf.dependents;
```

Algorithm 7 implements `resolve_token_dependencies` in line 20 in Algorithm 3. When a token whose order has been determined at the first pipe and has an entry in `token_dependencies`, we need to resolve the associated token dependency (line 1:9). We iterate over every element (`deferred_token`) of the entry in `token_dependencies[pf.token]` and remove the token from `deferred_token`'s dependents (line 3). If `deferred_token` does not have any dependent left, it is no longer a deferred token and becomes ready. We insert `deferred_token` in `ready_tokens` and remove it from `deferred_tokens`.

Algorithm 7: resolve_token_dependencies(pf)

```
   global: token_depencencies: a hashmap of a token and its
           related deferred tokens
   global: deferred_tokens: a hashmap of a token and its deferred
           object
   global: ready_tokens: a queue of ready tokens
   Input: pf: a pipeflow object
1  if pf.token ∈ token_dependencies then
2      for deferred_token ∈ token_dependencies[pf.token] do
3          deferred_token.dependents.erase(pf.token);
4          if deferred_token.dependents.empty() == true then
5              ready_tokens.push(deferred_token);
6              deferred_tokens.erase(deferred_token);
7          end
8      end
9  end
```

Algorithm 8 shows how we schedule tasks when a token finishes at a pipe in line 26 in Algorithm 3. We update variables (line 1:4) and define an array to track next tasks (line 5). We update the join counters based on the pipe type and determine the next possible tasks to run (line 6:11). When the join counter of a pipe reaches zero, we bookmark this pipe as a task to run next (line 7 and line 10). If two next tasks exist (line 12), the current task informs the scheduler to call a worker thread to run the task at the next parallel line (line 13) and reiterates itself on the next pipe (line 14). The idea here is to facilitate data locality as applications tend to deal with the next pipe as soon as possible. If only one task exists, the current task directly runs the next task with the updated `pf` object (line 16:20).

Algorithm 8: schedule_tasks(pf)

```
   global: num_pipes: the number of pipes
   global: num_lines: the number of parallel lines
   global: join_counters: a two-dimensional array of join counters
   Input: pf: a pipeflow object
 1 curr_pipe ← pf.pipe;
 2 next_pipe ← (pf.pipe + 1)%num_pipes;
 3 next_line ← (pf.line + 1)%num_lines;
 4 pf.pipe ← next_pipe;
 5 next_tasks = {};
 6 if curr_pipe is SERIAL and
     AtomicDecrement(join_counters[next_line][curr_pipe]) == 0
     then
 7 |   next_tasks.insert(1);
 8 end
 9 if AtomicDecrement(join_counters[pf.line][next_pipe]) == 0
     then
10 |   next_tasks.insert(0);
11 end
12 if next_tasks.size() == 2 then
13 |   call_scheduler(tasks[next_line]);
14 |   goto Line 2 in Algorithm 3;
15 end
16 if next_tasks.size() == 1 then
17 |   if next_tasks[0] == 1 then
18 |   |   pf ← tasks[next_line];
19 |   end
20 |   goto Line 2 in Algorithm 3;
21 end
```

2.5 Experimental Results

We implemented our framework using C++20 and evaluated the runtime performance on a x.264 application. We compiled all programs using g++11.4 with -std=c++20 and -O3 enabled. We ran all the experiments on a Ubuntu Linux 22.04 machine with 20 Intel i5-13500 CPU cores at 4.8 GHz and 128 GB RAM. All data is an average of ten runs.

Real-world x.264 Application

We evaluated our framework with x.264 applications and demonstrated the advantages of our approach to effectively handle generalized token dependency. The goal of x.264 is to generate H.264-compatible video streams [12, 13]. In H.264 standard, there are three types of video frames, *I*, *P*, and *B* frames. To encode frames, different frame types would reference either preceding or succeeding frames. *I* frames do not reference other frames, *P* frames reference one preceding *I* frame, and *B* frames reference one preceding and one succeeding *I* or *P* frame. Figure 2.1 illustrates a sample dependency diagram between these three frames. The backward dependency of *B* frames cannot be easily handled using existing pipeline frameworks without using `condition variable` to first reorder the tokens as PARSEC [12, 13] does. We modified the x.264 benchmark from [12, 13] by duplicating the frames to a bigger benchmark to evaluate the performance under different frame sizes and thread sizes. In addition, we added more `B` frames in the modified benchmark to further mimic the H.264 standard.

We considered PARSEC's pthread implementation [12, 13] as the baseline because PARSEC is the popular pipeline benchmark for many applications, such as ferret and x.264. To apply pipeline parallelism to x.264 applications, PARSEC assigns a thread to each frame. Every frame has a *condition variable* associated with its dependency. If a frame has unre-

solved dependency, its condition variable will wait until its dependency is resolved. When the frame becomes ready, its condition variable will broadcast the frame's readiness to any other frames that are waiting for the frame to finish. We implemented PARSEC's condition variable solution using C++. Figure 2.3 illustrates the implementation in which we use token 12 to simulate a *B* frame. This fine-grained control requires a high familiarity with low-level synchronization primitives from developers and is error-prone. With our framework, applications can directly specify the frame dependencies at the first pipe. Besides, PARSEC's solution could lead to deadlock when insufficient threads are spawned as discussed in Section 2.3. Our framework can avoid deadlock regardless of the thread counts.

Figure 2.7 compares the frame reordering time between our framework and PARSEC with up to 2 million frames and using up to 20 threads. We find out that the gap between our solution and PARSEC increases as the frame sizes grow. For example, when using 8 threads, the gap increases from 0.7 second at 1 million frames (2^{20}) to 1.4 seconds at 2 million frames (2^{21}). For the largest 2 million frame sizes, our framework is consistently faster than the baseline. For example, ours is 8.2%, 8.6%, 5.4%, 6.8%, 7.8%, and 5% faster than PARSEC when using 8, 10, 12, 14, 16, and 20 threads, respectively. We also notice that all of the plots show one trend that our framework outperforms PARSEC regardless of the frame sizes and thread sizes. For example, when running 1 million frames we are 0.7 second faster with both 8 and 16 threads; when using 16 threads, we are 8%, 8%, and 7.8% faster running 0.5, 1, and 2 millions frames, respectively. We attribute the observations to the reasons: (1) PARSEC uses fine-grained low-level *condition variable* to address the bi-directional frame dependencies. When a frame has unresolved dependency, the thread that processes the frame has to wait until the dependency is resolved. When a frame finishes, the thread needs to broadcast the completeness of the frame to other waiting

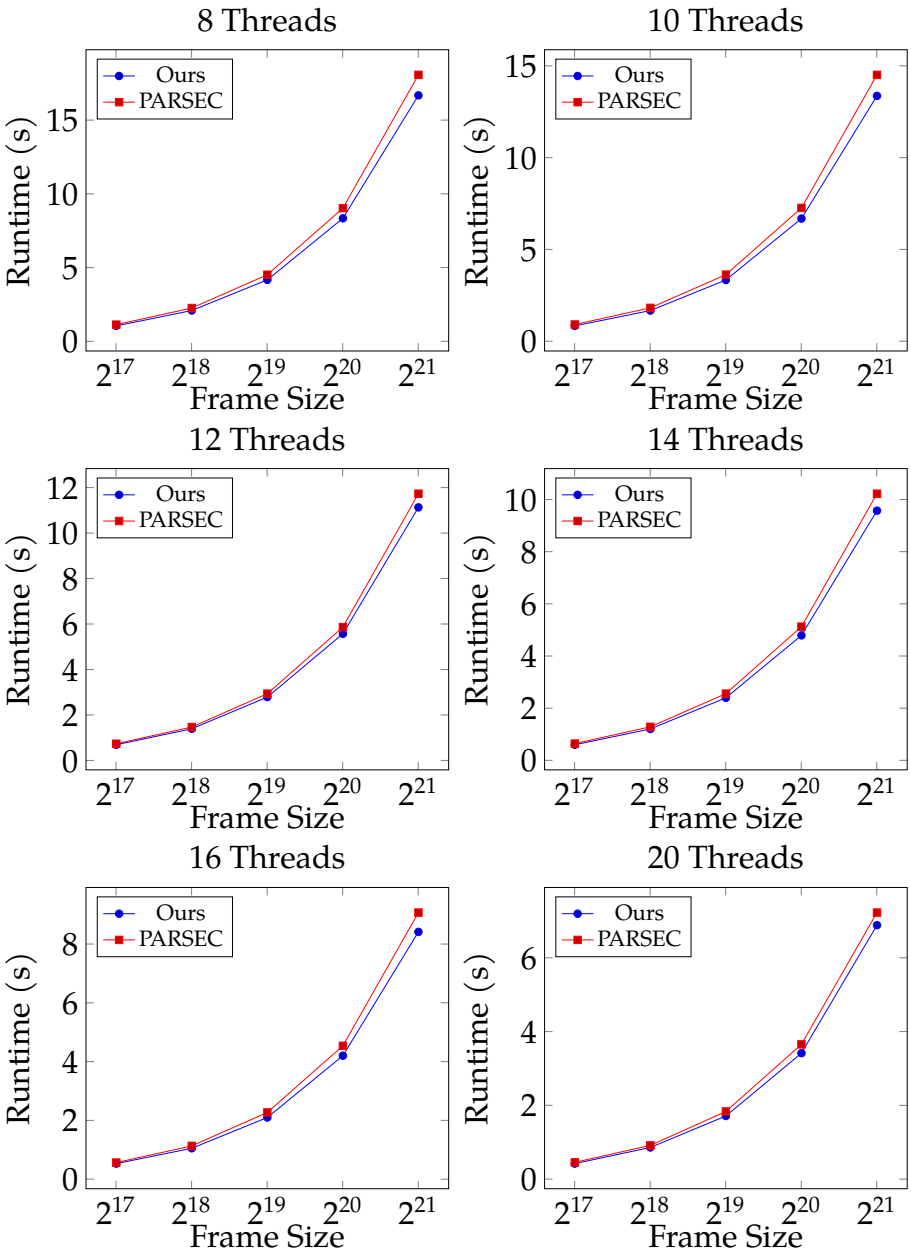


Figure 2.7: Runtime comparison between our framework and PARSEC at different frame and thread sizes.

frames. This mechanism that puts waiting threads to sleep and wakes up threads to resume the operations causes overheads. (2) Our solution does not wait for the frame but stores a deferred frame in `deferred_token` and continues to schedule other frames without waiting. This design could avoid the overheads that *condition variable* brings. As a result, our framework demonstrates the runtime advantages over the baseline in all cases regardless of the frame sizes and thread sizes.

2.6 Conclusion

In this chapter, we have introduced a new task-parallel pipeline programming framework on top of the state-of-the-art Pipeflow to explore pipeline parallelism in applications with token dependency. We have introduced an expressive programming model for applications to explicitly specify generalized token dependency. We have introduced a simple yet efficient scheduling algorithm to reorder tokens and schedule reordered tokens in the pipeline. We have evaluated the performance of our framework on an x.264 video encoding application. For example, our framework is 8.6% faster than PARSEC's implementation. We have integrated Pipeflow into the open-source task-parallel programming system, Taskflow [72] to benefit the HPC community. Our future plans are to (1) apply the framework to more different types of applications and bring interdisciplinary ideas to the parallel computing community, (2) extend token dependency-aware Pipeflow to task-parallel GPU computing platforms [21, 28, 29, 89, 106, 115, 116, 117, 153] and distributed environment [66, 69, 70], (3) leverage machine learning techniques to further improve the scheduling performance [27, 30, 132], and (4) improve the data locality for a thread while reordering tokens.

3 PROGRAMMING DYNAMIC TASK PARALLELISM FOR HETEROGENEOUS EDA ALGORITHMS

3.1 Abstract

Many EDA applications are extremely sparse, irregular, and control-flow intensive. Parallelizing this type of application can benefit from the ability to express dynamic task parallelism across arbitrary decision-making points at runtime. Unlike the traditional construct-and-run models, dynamic task parallelism offers programmers great flexibility to parallelize EDA algorithms that incorporate complex execution logic under dynamic control flow, such as branch-and-bound techniques, on-the-fly pruning, and recursive decomposition strategies. In this paper, we introduce a new programming model that supports the dynamic building of a computational task graph. We will cover scheduling details and best practices for exploring task parallelism under dynamic control flow. We will present a real use case of our model that has successfully parallelized a static timing analysis workload.

3.2 Introduction

Task graph programming (TGP) has inspired many new parallel and heterogeneous electronic design automation (EDA) algorithms [25, 26, 34, 43, 45, 47, 48, 49, 50, 61, 62, 63, 64, 73, 79, 114, 119, 120, 122, 123, 152] and large-scale machine learning problems [28, 84, 86, 87, 89, 116, 117]. Different from traditional loop-based models that explore parallelism across parallel loops, TGP formulates a workload as a *task graph* that models a function call as a *task* and a functional dependency as an *edge*. Figure 3.1(a) gives a task graph example of four tasks and four dependencies. By leveraging TGP, applications can enable top-down optimization to implement irregular

parallel decomposition strategies that consist of many tasks and dependencies. Then, a TGP runtime can scale these dependent tasks across a large number of processors with dynamic load balancing [113]. As a result, the parallel computing community has yielded many successful TGP in various application domains, such as OpenMP [5], Kokkos-DAG [35], PaRSEC [17, 58], OpenCilk [16, 141], HPX [93], Taskflow [59, 67, 72, 75, 111], and Fastflow [10].

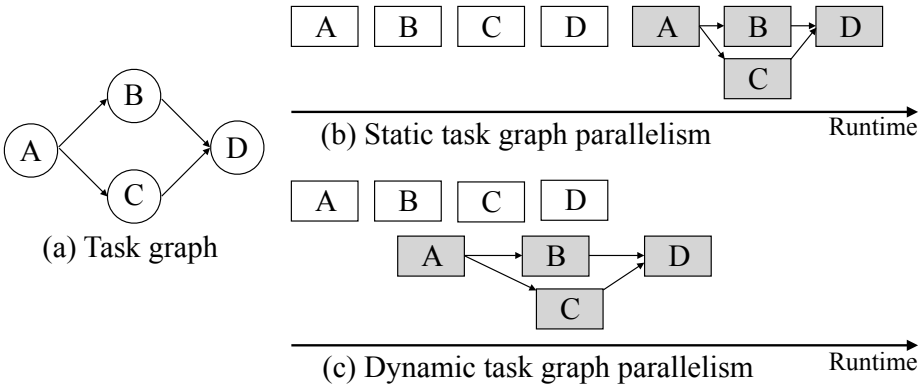


Figure 3.1: An illustration of the execution diagram of a task graph. White blocks denote the task creation and gray rectangles denote the task execution. Edges refer to the dependencies. (a) A task graph. (b) The execution diagram of STGP. (c) The execution diagram of DTGP.

Typically, TGP is categorized to two types: *static task graph programming* (STGP) and *dynamic task graph programming* (DTGP). In STGP, applications define the task graph first and submit it to an STGP runtime for execution, as shown in Figure 3.1(b). Since the graph structure is known in advance, the runtime can perform whole-graph optimization. On the other hand, DTGP defines the task graph structure dynamically. Tasks and dependencies are created on the fly depending on runtime variables and control-flow results, allowing the task creation time to overlap with the task execution time (see Figure 3.1(c)). Thus, DTGP is often more flexible than STGP when dealing with many EDA algorithms that fre-

quently incorporate dynamic control flow to implement irregular parallel decomposition strategies.

In this chapter, we introduce a new DTGP library called *AsyncTask* to assist EDA applications in quickly leveraging the power of DTGP. Compared to existing DTGP libraries, such as OpenMP [5] and PaRSEC [17, 58], AsyncTask is more expressive and transparent. For example, Listing 3.1 demonstrates the AsyncTask code for Figure 3.1(a). The code *explains itself* through a clean graph description language. The program creates a task graph of four tasks, `A`, `B`, `C`, and `D`. The dependency constraints state that task `A` runs before task `B` and task `C`, and task `D` runs after task `B` and task `C`. We summarize our contributions as follows.

- **Programming Model**. We present a simple and efficient dynamic task graph programming model. Our programming model provides a clear graph description language for applications to easily and quickly describe dynamic task graph parallelism. The expressiveness of our model improves programmer's productivity when coding large and complex task graphs for EDA applications.
- **Task Scheduling Algorithm**. We present an efficient task scheduling algorithm to support our programming model. Unlike existing solutions that mostly count on heavy mutexes to schedule dependent tasks [5, 17, 58], we only use lightweight *atomic counters* to resolve dependencies between tasks, enabling a more efficient task scheduling algorithm.

We have evaluated AsyncTask on a real-world static timing analysis application. Compared with the widely-used OpenMP [5] library, AsyncTask achieves a significant speed-up of 3.41× on a large design of 420K tasks and 530K dependencies.

```
int main(){
  Executor executor;
```

```
  // Create task A
  auto[A,fu_A]=executor.dependent_async([](){
    printf("Task A\n");
  });

  // Create task B, which has A as the dependent task
  auto[B,fu_B]=executor.dependent_async([](){
    printf("Task B\n");
  }, A);

  // Create task C, which has A as the dependent task
  auto[C,fu_C]=executor.dependent_async([](){
    printf("Task C\n");
  }, A);

  // Create task D, which has B and C as the dependent tasks
  auto[D,fu_D]=executor.dependent_async([](){
    printf("Task D\n");
  }, B, C);

  // Wait until D finishes
  fu_D.get();
}
```

Listing 3.1: AsyncTask implementation of Figure 3.1(a).

3.3 Background

Mainstream DTGP libraries used by EDA applications include OpenMP [5], PaRSEC [17, 58], and OpenCilk [16, 141]. In this section, we discuss their implementations of Figure 3.1(a) and compare them with AsyncTask (Listing 3.1). Then, we discuss their scheduling algorithms and highlight their limitations.

OpenMP

OpenMP [5] is a popular library that simplifies the development of parallel applications by adding parallelism to existing serial code through the use of compiler directives, pragmas, and runtime library routines. To implement Figure 3.1(a), OpenMP uses `#pragma omp task` construct to define a task and `depend` clause to specify that task's dependencies. Since OpenMP relies on a task's input and output data to describe a task's dependencies, applications need a data storage to store the data of every task. Listing 3.2 demonstrates the OpenMP implementation. Applications define a dynamic array `dependency` to store the execution results of the four tasks in the `A-B-C-D` order. Then applications use the entries in `dependency` as the inputs and outputs for a task. For example, applications use `#pragma omp task` to create task `D` and specify `D`'s inputs to be `dependency[1]` (i.e., `B`'s output) and `dependency[2]` (i.e., `C`'s output), and output to be `dependency[3]` in the `dependend` clause with `in` or `out` flags. To schedule tasks, OpenMP implements a lock-based hash table, in which the key of each entry is the address of a task's input or output data, and the value of that entry is a list of tasks accessing that address. As scheduling tasks require frequent accessing and updating the hash table, the overhead of using mutexes is heavy and can impact the overall runtime performance when running large and complex task graphs with multiple threads.

```
int main(){
  int *dependency = new int [4];
  #pragma omp parallel
  {
    #pragma omp single
    {
      // Create task A
      #pragma omp task depend(out:dependency[0])
      { printf("Task A\n"); }
```

```
      // Create task B, which has a dependency from A
      #pragma omp task depend(in: dependency[0])
                       depend(out:dependency[1])
      { printf("Task B\n"); }

      // Creae task C, which as a dependency from A
      #pragma omp task depend(in: dependency[0])
                       depend(out:dependency[2])
      { printf("Task C\n"); }

      // Create task D, which has depdnencies from B and C
      #pragma omp task depend(in: dependency[1],
                                  dependency[2])
                       depend(out:dependency[3])
      { printf("Task D\n"); }
    }
  }
  delete [] dependency;
}
```

Listing 3.2: OpenMP implementation of Figure 3.1(a).

PaRSEC

PaRSEC [17, 58] is a task-based runtime for distributed system. It leverages Domain Specific Languages (DSL) in its dataflow model to implement applications. To program a PaRSEC implementation, applications need the following steps: (1) Initialize a Message Passing Interface (MPI) engine as PaRSEC is a runtime for distributed system. (2) Define an application data structure using PaRSEC memory allocator to correctly build up the dependencies between tasks. (3) Initialize a PaRSEC taskpool to execute the tasks. (4) Define PaRSEC tasks and their function definitions. We simplify some implementation details to save more space

and do our best to keep the implementation as real as possible. For instance, the original API of creating task `D` in Figure 3.1(a) needs 15 arguments and we simplify those to 8 arguments. Listing 3.3 implements the PaRSEC code for Figure 3.1(a). Applications first define a MPI engine, an application data structure `dependency`, and a PaRSEC taskpool. Since PaRSEC specifies dependencies through a task's input and output data as OpenMP does, the data structure `dependency` is used to store the computation results of the four tasks in the `A-B-C-D` order. Next, applications create tasks using the `parsec_dtd_insert_task` API. The arguments of the API include the task's function label, the task's input and output data, and an end-of-argument `PARSEC_DTD_ARG_END` flag. The task's inputs (flagged with `INPUT`) and outputs (flagged with `OUTPUT`) are referenced using `tile_of_key` together with the corresponding index in `dependency`. For the individual task's function definition, applications unpack the arguments in the exact order as they are specified in `parsec_dtd_insert_task`. For example, applications specify task `D`'s two inputs (indexed 1 and 2 at `dependency`) in `parsec_dtd_insert_task` first and then one output (indexed 3 at `dependency`). Applications must obey the order to unpack them using `unpack_args`. To mark the end of a function definition, applications need to use the `PARSEC_HOOK_RETURN_DONE` flag. The task scheduling algorithm of PaRSEC is similar to OpenMP's design. Both of them rely on a lock-based hash table to manage the dependencies between tasks. The main difference is that PaRSEC additionally considers where to execute tasks that are created at a remote machine.

```
// Define task A
int A(parsec_task_t* this_task) {
  int *out;
  unpack_args(this_task, &out);
  printf("Task A\n");
  return PARSEC_HOOK_RETURN_DONE; }
```

```
// Define task B
int B(parsec_task_t* this_task) {
  int *in, *out;
  unpack_args(this_task, &in, &out);
  printf("Task B\n");
  return PARSEC_HOOK_RETURN_DONE; }

// Define task C
int C(parsec_task_t* this_task) {
  int *in, *out;
  unpack_args(this_task, &in, &out);
  printf("Task C\n");
  return PARSEC_HOOK_RETURN_DONE; }

// Define task D
int D(parsec_task_t* this_task) {
  int *in1, *in2, *out;
  unpack_args(
    this_task, &in1, &in2, &out);
  printf("Task D\n");
  return PARSEC_HOOK_RETURN_DONE; }

int main() {
  // 1. Initialize MPI
  // 2. Initialize application data, dependency
  // 3. Initialize PaRSEC taskpool, dtd_tp
  parsec_dtd_insert_task(A,
    tile_of_key(dependency,0),INPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_dtd_insert_task(B,
    tile_of_key(dependency,0),INPUT,
    tile_of_key(dependency,1),OUTPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_dtd_insert_task(C,
    tile_of_key(dependency,0),INPUT,
```

```
    tile_of_key(dependency,2),OUTPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_dtd_insert_task(D,
    tile_of_key(dependency,1),INPUT,
    tile_of_key(dependency,2),INPUT,
    tile_of_key(dependency,3),OUTPUT,
    PARSEC_DTD_ARG_END
  );
  parsec_taskpool_wait();
}
```

Listing 3.3: PaRSEC implementation of Figure 3.1(a).

OpenCilk

OpenCilk [16, 141] is a software infrastructure for task-parallel programming. A typical OpenCilk code is to spawn threads for tasks' operations and explicitly join threads for synchronization. Listing 3.4 demonstrates OpenCilk implementation of Figure 3.1(a). Applications spawn a thread to run `A` using the `cilk_spawn` directive, and explicitly join the thread to finish `A`'s execution using `cilk_sync`, ensuring the completion of `A`. The same pattern applies to task `B`, `C`, and `D` as well. As OpenCilk uses explicit synchronization directives to manage dependencies between tasks, the task scheduling algorithm is to wake up a thread from its thread pool to do a task's operation, and release that thread to the thread pool. When reaching the synchronization directives, the program execution halts until all threads finish and return to the thread pool.

```
// Define task A
void A(){ printf("Task A\n"); }

// Define task B
void B(){ printf("Task B\n"); }
```

```
// Define task C
void C(){ printf("Task C\n"); }

// Define task D
void D(){ printf("Task D\n"); }
int main() {
  cilk_spawn A();
  cilk_sync;
  cilk_spawn B();
  cilk_spawn C();
  cilk_sync;
  cilk_spawn D();
  cilk_sync;
}
```

Listing 3.4: OpenCilk implementation of Figure 3.1(a).

Limitations of Existing DTGP Libraries

Although OpenMP, PaRSEC, and OpenCilk have been used in many applications, we find several limitations of using them for DTGP: (1) Describing a task's dependencies through that task's input and output data is not expressive. Applications need to figure out the dataflow between two tasks to represent the task dependency, which is an indirect description and could reduce the code readability. (2) Programming a large task graph is very verbose. Applications have to explicitly indicate a task's input and output data, such as using `in` and `out` in OpenMP or `INPUT` and `OUTPUT` in PaRSEC. For PaRSEC users, they have to additionally write `PARSEC_DTD_ARG_END` to denote the end of input arguments and `PARSEC_HOOK_RETURN_DONE` to indicate the end of function definition. (3) Relying on a lock-based hash table to schedule tasks is not efficient. Their runtimes have to acquire a mutex when accessing the hash table, which introduces non-negligible lock overheads especially when running with multiple threads to schedule

a complex task graph.

Because of the above limitations, we have arrived at a conclusion that we need a new DTGP library that provides an expressive programming model to simplify the building of dynamic task graph parallelism. Additionally, we need an efficient task scheduling algorithm to support the programming model without much synchronization overhead.

3.4 AsyncTask

At a high level, AsyncTask enables an efficient implementation of irregular parallel decomposition strategies through a top-down dynamic task graph. We provide an expressive programming model for applications to describe the task graphs easily. We also introduce a task scheduling algorithm to support our programming model with only atomic counters to reduce the overhead of managing the dependencies between tasks.

Dynamic Task Graph Programming Model

To enable expressive DTGP, we directly specify a task's dependencies with its dependent tasks without describing the dependencies through the task's input and output data. Our programming model provides a simple interface `dependent_async` for applications to create tasks easily. Applications specify a lambda to encapsulate a task's operation followed by a list of dependent tasks in the input arguments. `dependent_async` will return a pair consisting of an instantiated task object and a future object which holds the execution result of that task.

Listing 3.1 exemplifies the code of Figure 3.1(a) using `dependent_async`. We define an executor which is a thread-safe object that manages a set of worker threads and executes our tasks. We create four tasks, `A`, `B`, `C`, and `D`. Every task defines its own lambda as the first argument followed by a list of dependent tasks. The dependent tasks must be created before the

current task. For instance, task `B` defines its operation to print a string in the lambda and specifies one dependent task `A` which is created before `B`. Upon returning from `dependent_async`, we obtain a pair consisting of an instantiated task object and a future object holding the execution result of that task. After constructing all tasks, we call `fu_D.get` to wait for task `D` to finish. As we construct tasks in the order `A`-`B`-`C`-`D`, the completion of `D` in turns means that all the other tasks have finished their executions.

This programming model is simple and expressive. Applications only need to write a callable (or lambda) and a list of dependent tasks to create a task. There is no extra flag needed, such as `in` and `out` in OpenMP or `INPUT`, `OUTPUT`, `PARSEC_DTD_ARG_END`, and `PARSEC_HOOK_RETURN_DONE` in PaRSEC. Without these flags, our AsyncTask implementation is easy to read and debug, enabling EDA developer's high productivity when programming large and complex task graphs.

Algorithm

To support our programming model, we design a new task scheduling algorithm, as illustrated in Figure 3.2. After applications call `dependent_async` to create a task, the executor checks if the new task has any dependent tasks. If yes, AsyncTask builds up the dependencies between the new task and each one of its dependents (denoted with ❶), and then the new task waits until all of its dependents finish executions (denoted with ❷). Otherwise, AsyncTask directly executes the new task (denoted with ❸). Next, we dive into the three parts in more details.

❶ **Building up the dependencies**. If a new task has any dependent tasks, we need to build up the dependencies between the new task and each one of its dependent tasks (see ❶ in Figure 3.2). In AsyncTask, we express every dependency with the new task presenting in the successor list of every dependent task. Assigning every task a successor list and representing dependencies using successor lists simplifies our design

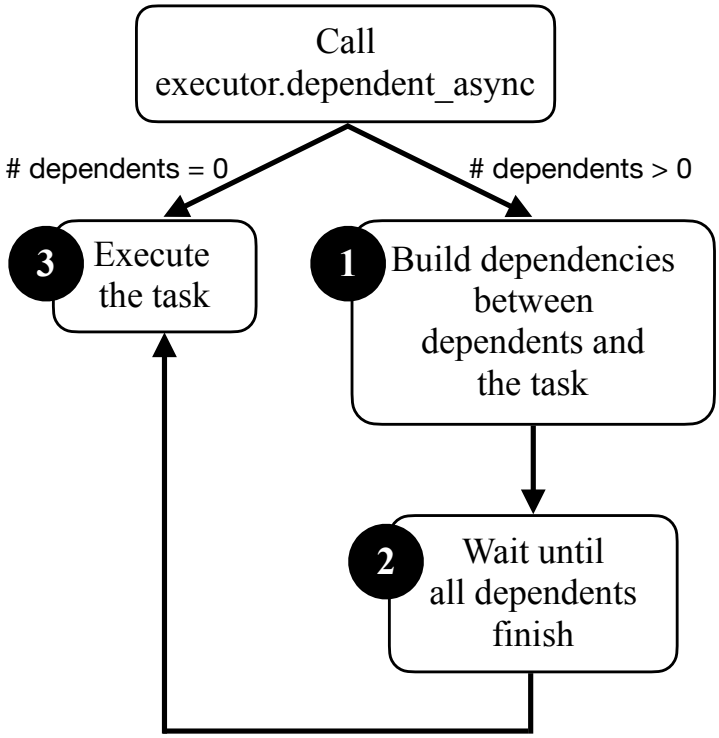


Figure 3.2: A flowchart of our task scheduling algorithm.

when it comes to resolving a dependency. For example, for the task graph in Figure 3.1(a), both task `B` and `C` have task `A` as the dependent task, and `A`'s successor list would have two successor tasks `B` and `C`. When `A` finishes, AsyncTask can quickly resolve the dependencies by directly checking `A`'s successor list rather than iterating every existing task to see what dependency to resolve. To safely and successfully add tasks into a dependent task's successor list, there are two concerns. First, when an executor adds tasks into a dependent task's successor list, we need to ensure that dependent task is alive. Since every task is an instantiated task object, it will be destroyed and returned to the operating system after it finishes the execution. To avoid adding tasks in an empty task object's successor list (see Figure 3.3(a)), we leverage the logic C++ smart

pointer *std::shared_ptr* to retain *shared* ownership of a task between the main thread and the executor, ensuring that task remains alive throughout the entire program (see Figure 3.3(b)). When a worker thread finishes a task's execution, it will remove the task from the executor, decrementing the number of shared owners by one. If that counter reaches zero, the task is then destroyed.

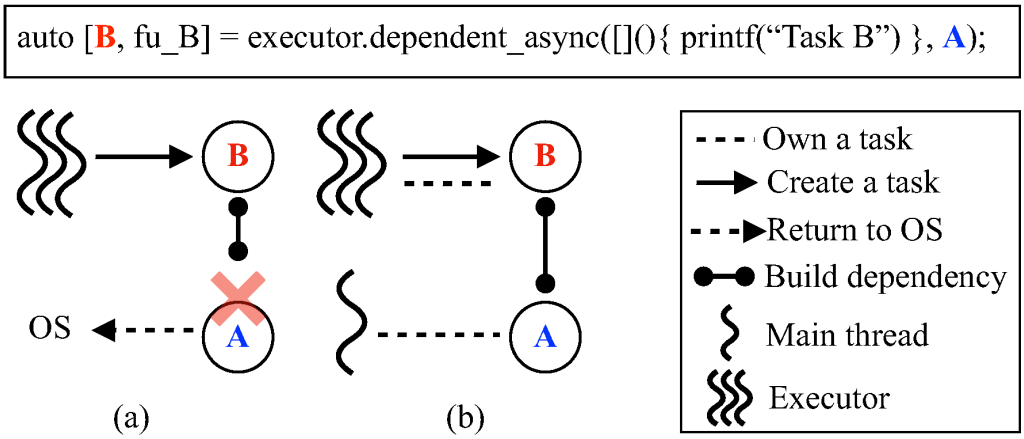


Figure 3.3: An illustration of shared ownership of task `A` and `B` in Figure 3.1(a). (a) `A` finishes and returns to operating system (OS). An executor relates task `B` to an empty task. (b) Main thread owns `A`. An executor successfully relates `B` and `A`.

Second, we need to protect every successor list from data race as multiple threads can add tasks in a successor list simultaneously. To avoid data race, we assign every task an atomic variable to protect its own successor list. Every atomic variable has three states `FINISHED`, `UNFINISHED`, and `LOCKED`. `FINISHED` denotes a task completion, `UNFINISHED` denotes an ongoing execution of a task, and `LOCKED` denotes that another task is adding itself to the successor list of the current task. Figure 3.4 visualizes how a three-state atomic variable can protect a successor list from data race. In (a), assume `A` has finished its execution and its state is set to `FINISHED`. There is no need to add `B` and `C` in `A`'s successor list. No data race on `A`'s

successor list. In (b), three tasks are performing compare-and-swap (CAS) operations on A's state at the same time. If A succeeds in this operation, then we are in the situation of (a). If B succeeds, then B changes A's to LOCKED and can add itself in A's successor list solely, as illustrated in (c). After B finishes the adding, B changes A's state back to UNFINISHED, and the whole process repeats for C.

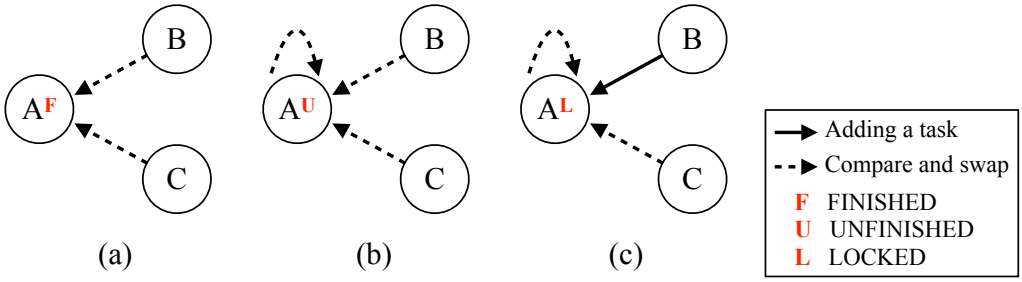


Figure 3.4: An illustration of using the atomic variable to change task A's state. A, B, and C refer to the tasks in Figure 3.1(a). A is trying to change the state to FINISHED. B and C are trying to add themselves to A's successor list. (a) A is at the FINISHED state. (b) A is at the UNFINISHED state. (c) A is at the LOCKED state.

2 **Waiting for dependent tasks to finish**. After AsyncTask successfully creates a new task and builds up its dependencies, the next step for the new task is to wait for its dependent tasks to finish (see 2 in Figure 3.2). To achieve this, we assign every task an atomic counter to keep track of the number of its unfinished dependent tasks. The initial value is the number of dependent tasks specified in `dependent_async` API. When one of its dependent tasks finishes the execution, the dependent task will decrease the atomic counter of that task by one. If that atomic counter becomes zero, meaning the task has no unfinished dependent task and is ready to execute. Figure 3.5 visualizes the process. In (a), task B's initial atomic counter is one and it is performing the CAS operation in order to add itself in A's successor list. In (b), task B successfully added itself in A's successor list. In (c), task A finishes the execution and decreases

the atomic counter of its successor B by one. Task B now has the atomic counter equal to zero and is ready to execute. We refer the atomic counter to join counter later and would use them interchangeably in the paper.

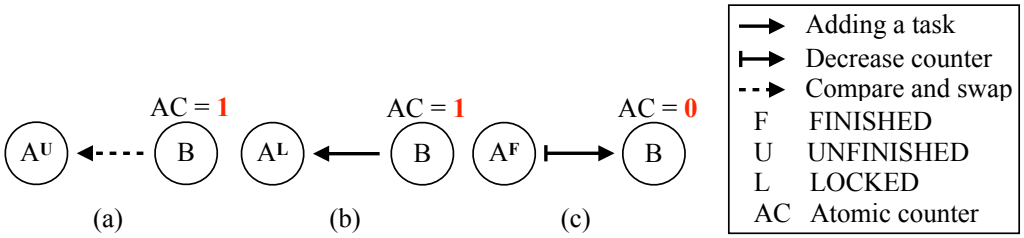


Figure 3.5: An illustration of using atomic counters to represent the number of dependent tasks. A and B refer to the tasks in Figure 3.1(a). (a) B is performing the CAS operation on A's state. (b) B is adding itself in A's successor list. (c) A finishes its execution and decreases B's atomic counter by one.

3 **Executing a task and resolve dependencies**. AsyncTask executes a task when a task has no dependent task or all of its dependent tasks have finished their executions(see 3 in Figure 3.2). When a task finishes the execution, we need to resolve the associated dependencies between it and all of its successor tasks. To achieve this, a task iterates its successor list and decrease the atomic counter of every successor by one. Figure 3.6 visualizes how a task resolves the associated dependencies after it finishes the execution. In (a), task A finishes the execution, iterates its successor list, and decrease the atomic counter of B and C by one. Now, B and C have zero join counter and are ready to execute. In (b), B and C have finished the execution and both decrement D's join counter by one. D has zero join counter and is ready to execute.

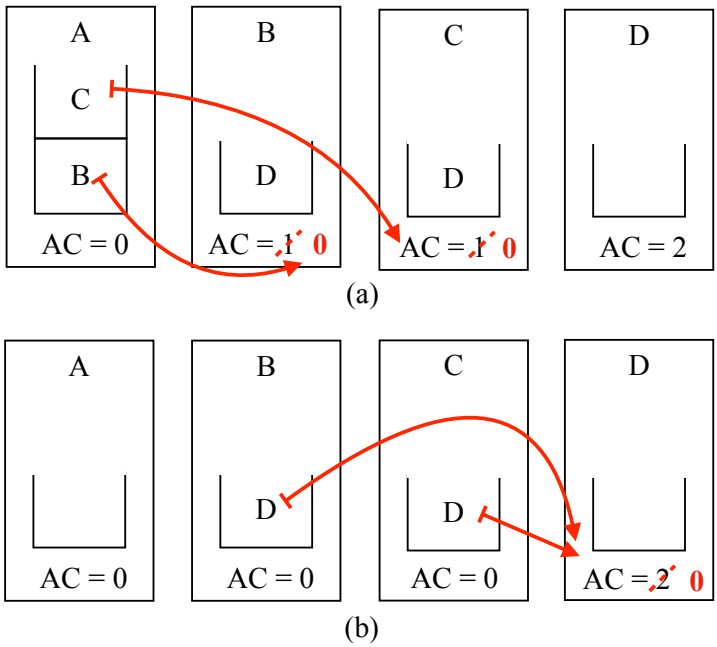


Figure 3.6: An illustration of resolving dependencies after a task finishes. `A`, `B`, `C`, and `D` refer to the tasks in Figure 3.1(a). (a) `A` finishes the execution and decreases the atomic counter (AC) of `B` and `C` by one. (b) `B` and `C` finish, and both decrease `D`'s AC by one.

Pseudocode

In this section, we implement the flowchart in Figure 3.2 based on the design overview presented in the previous section. Algorithm 9 implements the `dependent_async` API which takes a callable (or lambda) and a list of dependent tasks (`deps`) and returns a pair consisting of the created task and a future. We first define a future object `future` which will hold the calculation result of the task (line 1). Then, we calculate how many dependent tasks the new task has (line 2). We initialize the new task `task` (line 3). Next, we iterate every dependent task and build the dependencies (line 4:6). We build the dependencies between `task` and every dependent task in line 5 (details are given in Algorithm 10). Since

we are in a multi-threaded environment, some dependent tasks may have finished before we build the dependencies between them and `task`. If all dependents finished or the task has no dependent task specified (line 7:9), we can directly schedule `task` (line 8). In the end, we return a pair of the created task `task` and the future object `future` (line 10).

Algorithm 9: dependent_async(callable, deps)

Input: *callable*: a callable of the task
Input: *deps*: a list of dependent tasks
Output: (*task*, *future*): a pair of the created task and the corresponding future object

```
1  Create a future
2  num_deps ← sizeof(deps)
3  task ←initialize_task(callable, num_deps, future)
4  for dep ∈ deps do
5  |  process_dependent(task, dep, num_deps)
6  end
7  if num_deps == 0 then
8  |  schedule_async_task(task)
9  end
10 Return (task, future)
```

Algorithm 10 implements `process_dependent` in which we build the dependency between a task and one of its dependent tasks. This API takes the task `task`, a dependent task `dep`, and the number of dependent tasks `num_deps` in its inputs. We add `task` in `dep`'s successor list if `dep` has not finished. First, we store the state of `dep` in `dep_state` (line 1). We create a variable `target_state` to store the state `UNFINISHED` (line 2). Next, we perform the CAS atomic operation on `dep_state` (line 3). If `dep_state` is equal to `target_state` (i.e. `dep` has not yet finished), we swap `dep_state` to `LOCKED`, which means we now enter the critical region and have the exclusive access to `dep`'s successor list (line 4:5). We add `task` in the successor list (line 4) and resume `dep_state` back to `UNFINISHED` (line 5). If `dep_state` is not equal to `target_state`, `target_state` would

be set to dep_state atomically by the CAS operation. If target_state is set to FINISHED, that means dep has finished and we decrement task's join counter by one and update num_deps accordingly (line 7:8). If target_state is set to LOCKED, that means some other task enters the critical section in line 4:5 first, we reiterate to the beginning to try the whole process again (line 11).

Algorithm 10: process_dependent(task, dep, num_deps)

```
   Input: task: a created task
   Input: dep: a dependent task
   Input: num_deps: the number of dependent tasks
 1 dep_state ← dep.state
 2 target_state ← UNFINISHED
 3 if dep_state.CAS(target_state, LOCKED) then
 4 |   dep.successors.push(task)
 5 |   dep_state ← UNFINISHED
 6 end
 7 else if target_state == FINISHED then
 8 |   num_deps ←AtomicDecrement(task.join_counter)
 9 end
10 else
11 |   goto line 2
12 end
```

Algorithm 11 implements the schedule_async_task API in which we execute a task, change its state to FINISHED, and then resolve the dependencies for its successors. This API takes the task task in the input. Before executing task, we change its state to FINISHED to prevent any other tasks from adding themselves in the task's successor list. We define target_state to be UNFINISHED (line 1). Then we perform the CAS operation on task.state. If succeed, that means task.state is equal to target_state (i.e. UNFINISHED) and is then set to FINISHED (line 2). If failed, that means some other task enters the critical section in lines 4:5 in Algorithm 10 and is adding itself in task's successor list. In such case,

we reset `target_state` to `UNFINISHED` and perform the CAS operation again (line 3). When we successfully set `task.state` to `FINISHED`, we can execute `task` (line 5). Next, we iterate the successor list and decrement the join counter of each successor by one (line 6:10). If any successor whose join counter becomes zero after the decrementation, we schedule that successor directly (line 8). Now, we finish executing `task`, we can decrement the number of `task`'s shared owners (`ref_count`) by one (line 11). If `task` does not have any shared owner, we can delete `task` and return its allocated resource to the operating system (OS) (line 12).

Algorithm 11: schedule_async_task(task)

```
   Input: task: a created task
 1 target_state ← UNFINISHED
 2 while not task.state.CAS(target_state, FINISHED) do
 3 |   target_state ← UNFINISHED
 4 end
 5 Invoke(task.callable)
 6 for successor ∈ task.successors do
 7 |   if AtomicDec(successor.join_counter) == 0 then
 8 |   |   schedule_async_task(successor)
 9 |   end
10 end
11 if AtomicDecrement(task.ref_count) == 0 then
12 |   Delete task
13 end
```

3.5 Experimental Results

We implemented AsyncTask using C++20 and evaluated its performance on an industrial static timing analysis (STA) application [63, 79] that leverages task graph parallelism to parallelize graph-based analysis (GBA). We consider the state-of-the-art open-source STA engine, OpenTimer[6],

as our experimental environment. OpenTimer formulates the GBA algorithm into a task graph and 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., slew, delay, arrival time), while each edge represents a dependency between two tasks. Table 3.1 lists the statistics of the three circuits we use. $\|V\|$ denotes the number of the tasks in a circuit and $\|E\|$ denotes the number of the edges.

We compiled programs using Clang++17 with `-std=c++20` and `-O3` enabled. We ran all the experiments on a Centos Stream 8 machine with 8 Intel i7-9700K CPU at 3.60GHz and 32 GB RAM. All data is an average of ten runs. The implementation of AsyncTask is available in the Taskflow project [72].

Table 3.1: Task ($\|V\|$) and edge ($\|E\|$) counts of three circuits.

Circuits	$\|V\|$	$\|E\|$	$\|V\| + \|E\|$
wb_dma	13,125	16,593	29,718
tv80	17,038	23,087	40,125
ac97_ctrl	42,438	53,558	95,996

Baseline

Given the large number of parallel libraries, it is impractical to compare AsyncTask with all of them. We consider OpenMP[5] as the baseline because it is a mainstream DTGP library that has been widely employed by many EDA applications. Compared with other existing DTGP libraries, OpenMP allows applications to more flexibly define task dependencies using directive-based programming (e.g., range iterator clauses).

Performance Comparison

Figure 3.7 compares the memory and runtime between AsyncTask and OpenMP with up to 16 threads for completing the analysis of the three circuits. In terms of memory usage, we see that OpenMP consistently consumes more memory than AsyncTask in all cases. This is because OpenMP implements a global lock-based hash table to track tasks and their dependencies. The key of each entry in the table is the memory address of a task's input or output data and the value is the tasks that access the corresponding memory address. The number of entries in the table grows in proportion to the edge counts. On the other hand, AsyncTask does not need a global data structure but assigns each task a successor list, which can largely reduce the overhead of lock access.

Regarding runtime performance, AsyncTask outperforms OpenMP in all cases. For example, AsyncTask is 3.19×, 3.19×, and 3.41× faster than OpenMP at 16 threads for wb_dma, tv80, and ac97_ctrl, respectively. The reason for the runtime difference comes from the design that OpenMP needs mutexes to access its global hash table in order to resolve dependencies between tasks. However, AsyncTask only uses lightweight atomic counters to resolve task dependencies.

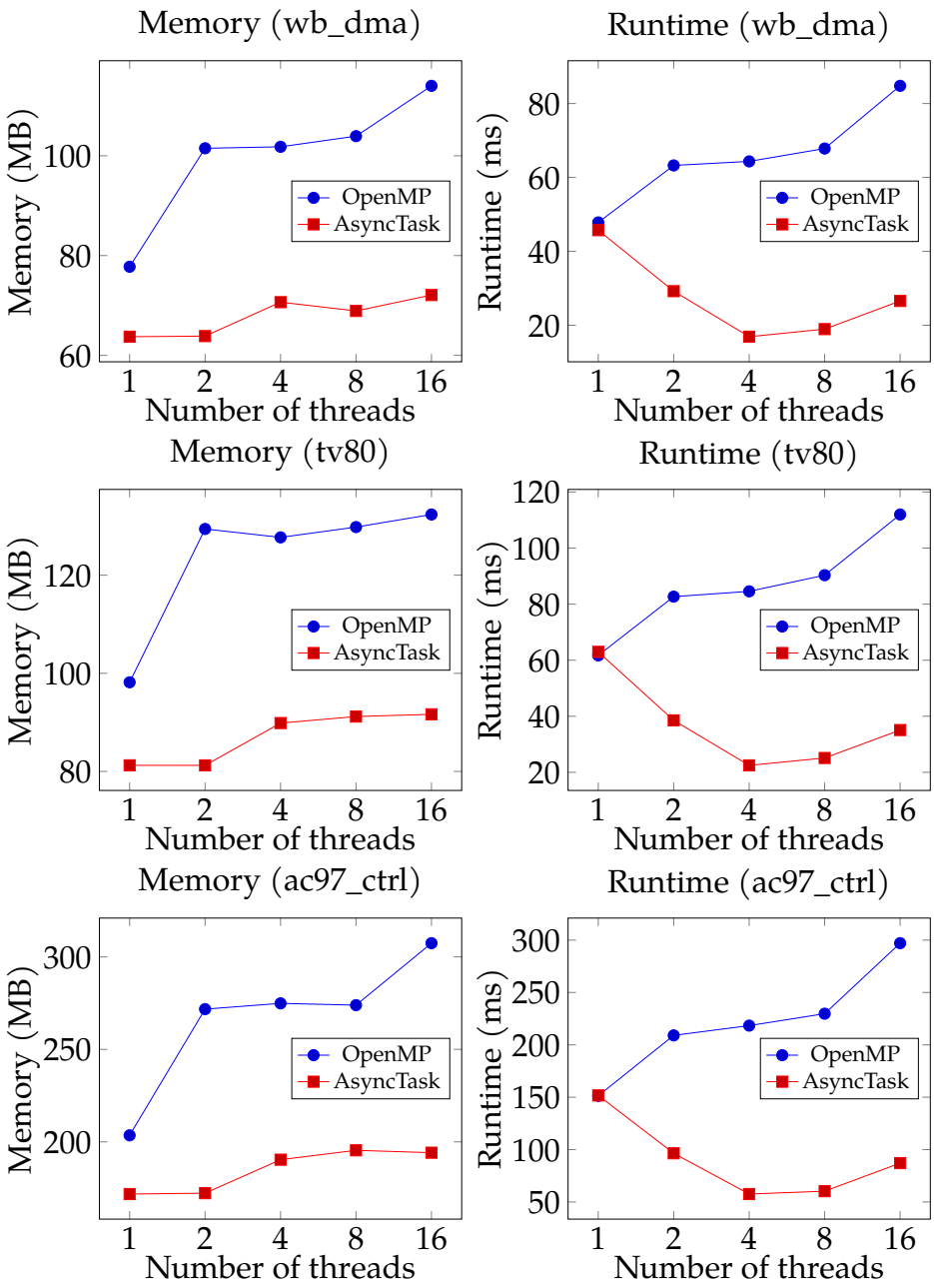


Figure 3.7: Memory and runtime comparison of the STA workload on three circuits (wb_dma, tv80, ac97_ctrl) between AsyncTask and OpenMP.

3.6 Conclusion

In this chapter, we have introduced a new DTGP library called AsyncTask to support the programming of dynamic task graph parallelism. AsyncTask has introduced a new expressive programming model supported by an efficient scheduling algorithm. We have also presented a real use case in static timing analysis and demonstrated the promising performance of AsyncTask over a mainstream DTGP library, OpenMP. AsyncTask has been integrated into the open-source Taskflow project. Future work will focus on applying AsyncTask to other EDA applications, such as distributed computing [66, 69, 70, 78], macro modeling [102], and path-based analysis [77, 79, 80, 81, 82].

4 A RESOURCE-EFFICIENT TASK SCHEDULING SYSTEM USING REINFORCEMENT LEARNING

4.1 Abstract

Computer-aided design (CAD) tools typically incorporate thousands or millions of functional tasks and dependencies to implement various synthesis and analysis algorithms. Efficiently scheduling these tasks in a computing environment that comprises manycore CPUs and GPUs is critically important because it governs the macro-scale performance. However, existing scheduling methods are typically hardcoded within an application that are not adaptive to the change of computing environment. To overcome this challenge, this paper will introduce a novel reinforcement learning-based scheduling algorithm that can learn to adapt the performance optimization to a given runtime (task execution environment) situation. We will present a case study on VLSI timing analysis to demonstrate the effectiveness of our learning-based scheduling algorithm. For instance, our algorithm can achieve the same performance of the baseline while using only 20% of CPU resources.

4.2 Introduction

Computer-aided design (CAD) tools typically incorporate thousands or millions of functional tasks and dependencies to implement various synthesis and analysis algorithms [25, 26, 34, 43, 45, 47, 48, 49, 50, 61, 62, 63, 64, 73, 79, 114, 119, 120, 122, 123, 152]. For instance, [63] describes timing analysis algorithms in a top-down *task graph* where each task represents a function and each edge represents a functional dependency. Efficiently scheduling these tasks in a computing environment that comprises manycore central processing units (CPUs) and graphics processing

units (GPUs) is critically important because it governs the macro-scale performance [43, 45, 47, 48, 49, 50, 51, 62, 68, 119, 120]. However, existing scheduling solutions either resort to general-purpose heuristics (e.g., work stealing [73, 75, 111, 113]) or a custom scheduling method (e.g., hard-coded [152]). These solutions are typically not adaptive to the change in the computing environment and often consume large scheduling resources due to the randomness involved in dynamic load balancing.

Recent advances in machine learning have inspired us to design a new scheduling framework that learns to interact with a computing environment [36]. Despite exciting progress in learning-based scheduling solutions, most of them target *independent* job batches in a high-performance computing (HPC) cluster. These solutions are not suitable for CAD problems where the goal is to find a *resource-efficient* scheduling plan for running dependent tasks using minimal CPU resources. This type of scheduling plan is very important because many CAD task graphs are much larger and more complex than conventional HPC workloads. For instance, a timing propagation task graph can compose up to 500 million tasks and 700 million dependencies to complete a full-timing analysis of a large design [63, 72]. Due to the sparsity, we may just use a few CPU cores to optimally complete the task graph, which in turn improves the resource utilization and the overall system throughput performance.

To this end, we introduce in this paper a resource-efficient task-scheduling system by harnessing the power of reinforcement learning. We summarize our technical contributions below:

- **Scheduling Algorithm.** We have introduced a reinforcement learning-based task scheduling algorithm to adapt the performance optimization to the computing environment. With our scheduling algorithm, applications are able to schedule tasks with few execution contexts while achieving a comparable runtime performance to existing solutions.

- **Generalizability.** We apply our RL-based scheduling algorithm to schedule a wide range of task graphs and show its superior performance. Surprisingly, our experimental results show that the RL-based scheduling policy learned from limited classes of task graphs, can generalize well to a wide range of diverse task graphs. Therefore, our RL-based scheduling algorithm provides a universal scheduling solution to multi-core systems.
- **Extensible State Representations.** We have introduced an easy-to-extend state representation to accommodate new computing environment statistics, such as power consumption. With this state representation, applications are able to quickly switch to a different scheduling algorithm based on their specific needs.

We have evaluated our framework on a real static timing analysis (STA) workload that executes a task graph to complete full timing analysis. Compared to the popular heuristic-based scheduler that assigns tasks to all 40 workers uniformly at random, our RL-based scheduler achieved a slightly lower runtime on multiple task graphs using only 7-8 workers.

4.3 Background

System Overview

Our system targets at static timing analysis (STA) application, one of the most important steps in the entire EDA flow, and describes a STA workload as a task graph[63]. The goal is to efficiently schedule the tasks of the task graph. The task graph consists of multiple nodes and edges, which represent the tasks and the dependencies among the tasks, respectively. In particular, the task dependencies not only constrain the execution order of the tasks, but also determine the data flow among them. Take the task

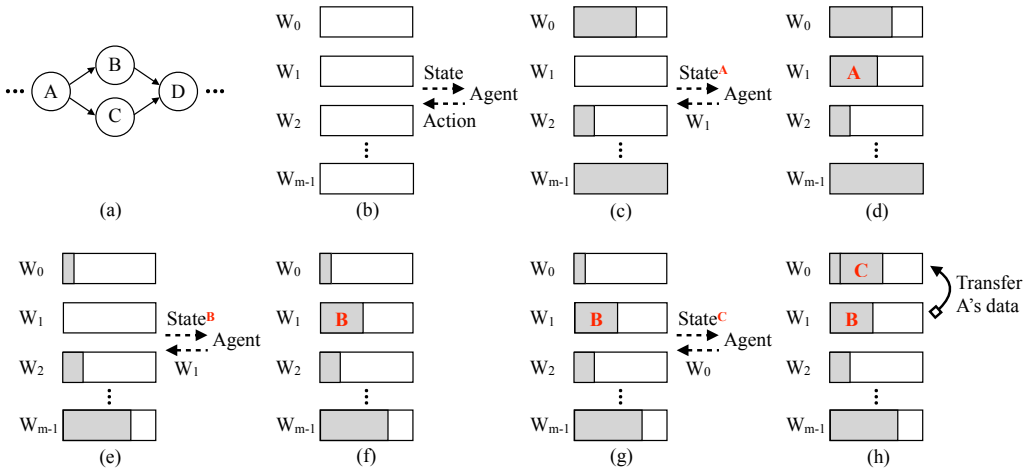


Figure 4.1: Illustration of our task scheduling system. Gray rectangles denote the workloads of workers. (a) A task graph. (b) The task scheduler. (b) The scheduler asks `Agent` for task `A`'s action. `Agent` suggests W_1. (d) W_1 has `A` in its queue. (e) The scheduler asks `Agent` for task `B`'s action. `Agent` suggests W_1. (f) W_1 has `B` in its queue. (g) The scheduler asks `Agent` for task `C`'s action. `Agent` suggests W_0. (h) W_0 has `C` in its queue. `A`'s data is transferred to W_0.

graph shown in Figure 4.1(a) as an example. The task dependencies require that `A` executes before `B` and `C`, and `D` executes after `B` and `C`. Moreover, the execution of `B` and `C` needs `A`'s data, and the execution of `D` needs both `B` and `C`'s data. To schedule tasks across the execution contexts (e.g., CPUs) in a non-stationary computing environment, we introduce a reinforcement learning (RL)-based task scheduler as illustrated in Figure 4.1(b). Next, we provide a high-level overview of this RL-based scheduler, and leave all the technical details to Section 4.4.

We consider a multi-core system and denote the i-th worker as W_i. Each worker has its own task queue to store the tasks assigned to it. We denote the general computing environment as `State`, which includes the total workloads assigned to the workers and some information about the task to be scheduled (see Section 4.4 for the details). Then, based on the

current State, the reinforcement learning Agent determines an Action that assigns the task to a certain worker. Figure 4.1(c)-(h) illustrate how our task scheduler schedules the three tasks A, B, and C from the task graph in Figure 4.1(a). To explain, in (c), the Agent first assigns A to worker W_1 based on the current $State^A$. In (d), A is inserted into W_1's queue. After W_1 completes A, Agent assigns B and C to W_1 and W_0 according to $State^B$ and $State^C$, respectively, as shown in Figure4.1(e) and (g). As B is assigned to the same worker (W_1) as A, there is no data transfer cost for B shown in Figure 4.1(f). In contrast, C requires an extra data transfer cost shown in Figure 4.1(h).

4.4 Reinforcement Learning-Based Scheduling

In this section, we reformulate the task scheduling problem as a reinforcement learning (RL) problem, and apply the Deep Q-Learning algorithm [131] to train a good RL policy for autonomous task scheduling.

Reinforcement learning and Markov Decision Process

Reinforcement learning (RL) is a powerful machine learning framework for learning optimal decision-making in a so-called *Markov Decision Process* (MDP) [11, 39, 98, 99, 125, 144, 150]. In RL, an agent interacts with a complex environment through an MDP, and the interaction data are used to further improve the agent's decision-making. Specifically, MDP is an abstract sequential decision-making process that consists of the following key elements.

- State s_t. At any time t, the agent observes the global state s_t of the environment, which contains all information that the agent needs to take an action.

- Policy π and action a_t. Based on the current state s_t, the agent follows its policy $\pi(\cdot|s_t)$ to take an action a_t. Here, policy π is regarded as a probability distribution over all possible actions conditioned on the state s_t.

- State transition kernel P. After action a_t is taken, the global state s_t transfers to a new state s_{t+1} following the environment's transition kernel $P(\cdot|s_t, a_t)$, which is a distribution over all possible states conditioned on s_t, a_t.

- Reward r_t. The agent receives a reward signal r_t after the state transition at time t. Here, the reward r_t generally depends on s_t, a_t and s_{t+1}.

Equation (4.1) below illustrates the evolution of MDP. In RL, the goal of the agent is to learn the optimal policy π^*, following which yields the highest accumulated reward when interacting with the environment through the MDP.

$$\text{(MDP):}\ s_0 \xrightarrow{\pi(\cdot|s_0)} a_0 \xrightarrow{P(\cdot|s_0,a_0)} (s_1, r_0) \xrightarrow{\pi(\cdot|s_1)} a_1 \cdots \tag{4.1}$$

Formulate Task scheduling as an MDP

We view task scheduling as an MDP. Specifically, consider a multi-core system with m workers. Denote T_k as the k-th task to be assigned to the workers, and denote $P(T_k)$ the set of parent tasks of T_k. Then, the elements of this MDP can be specified as follows.

- State. Consider the time when the k-th task T_k is ready for scheduling, the state of the RL agent is a vector containing $2m + 1$ coordinates. The first m coordinates record the current total workload of the queues of the m workers (each coordinate records the workload of one queue). The next m coordinates record the distribution of the

total workload of the parent tasks $P(T_k)$ over the m workers, i.e., each coordinate records the amount of workload of $P(T_k)$ done by one worker. Finally, the last coordinate of the state vector records the total workload of the task T_k to be scheduled. We can see Figure 4.2 for the state vectors for task `A` and `B` from the task graph in Figure 4.1(a). These state information are queried from the system whenever a new task is ready for scheduling, and we take logarithm to reduce the scale of large numerical values in this state. In particular, these information is directly related to the balance of workload assigned to the workers and the data transfer cost, which are two critical factors that affect the overall system performance.

- Policy and action. There are m possible actions since the task T_t will be assigned to one of the m workers. The policy is specified based on a so-called state-action value table $Q(s, a)$, which evaluates the expected return of taking action a in state s. In the next subsection, we describe the deep Q-learning algorithm used to learn $Q(s, a)$ in a data-driven way.

- State transition. Once task T_k is assigned to a worker based on the action generated by the policy, the workload of that worker's queue will change. This will further lead to a new system state. We note that the new state depends only on the previous state and the action taken, which satisfies the Markov property required by MDP.

- Reward. After each state transition, the agent receives a reward signal, which is designed based on two system performance-related characteristics: the balance of total workload assigned to the workers and the data transfer cost induced by scheduling the task T_k.

  The balance of workload is defined as the gap between the maximum queue load and the minimum queue load among the workers, which quantifies the level of imbalance of the workers' queues.

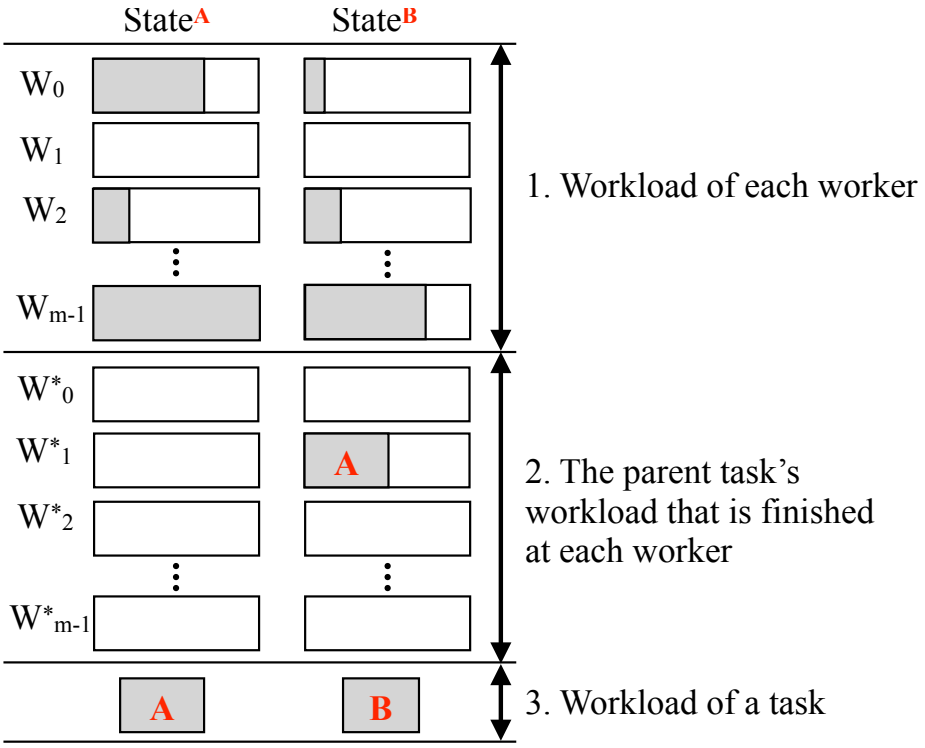


Figure 4.2: Illustration of the state representations when scheduling task `A` and `B` in the task graph of Figure 4.1(a). The first column refers to the `State` for task `A` and the second for `B`. Gray rectangles denote the workloads. Every state includes 1) the workload of each worker, 2) the parent task's workload that is finished at each worker, and 3) the workload of a task. The first m rows of $State^A$ and $State^B$ correspond to Figure 4.1(c) and 4.1(e), respectively. As task `A` is executed by W_1, W_1^* for $State^B$ is the workload of task `A`. The workload of other W^* is empty.

For the data transfer cost, suppose T_k is assigned to worker i, then the induced data transfer cost is the total data load of its parent tasks $P(T_k)$ that are not assigned to worker i (these parent tasks' data need to be transferred to worker i in order to execute task T_k).

Mathematically, the reward signal is defined as follows,

$$r_t := -\log_{10}(\text{workload balance}) - \alpha \log_{10}(\text{transfer cost}), \quad (4.2)$$

where $\alpha > 0$ is a hyper-parameter, and we take logarithm to reduce the scale of large numerical values. Intuitively, the above reward design penalizes taking actions that would cause imbalanced queue workloads and high data transfer cost.

Deep Q-Learning Algorithm

Deep Q-learning is a popular algorithm that aims to learn the optimal policy that maximizes the expected accumulated reward [131]. This problem is formulated as follows,

$$\max_{\pi} \; J(\pi) := \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \big| \pi\Big],$$

where $\gamma \in (0, 1)$ is a pre-selected discount factor. In particular, given state s_t at time t, the policy generates an action a_t based on a state-action value function Q according to

$$\pi(a_t|s_t) = \arg\max_{a} Q(s_t, a). \tag{4.3}$$

Intuitively, $Q(s, a)$ evaluates the expected return of taking action a in state s, and the policy π simply suggests the action that leads to the highest Q value in a given state. It is well-known that the Q-function satisfies the following Bellman equation [143],

$$Q(s_t, a_t) = r_t + \gamma \mathbb{E}\Big[\max_{a} Q(s_{t+1}, a)\Big]. \tag{4.4}$$

The main idea of deep Q-learning is to parameterize the Q-function using a deep neural network $Q_\theta(s, a)$, where θ denotes the network parameters. This network takes state as the input and outputs the Q values associated with all possible actions, as illustrated in Figure 4.3. Specifically, the deep Q-learning algorithm is summarized in Algorithm 12, and it

consists of the following key steps.

- Data collection. At each time step t, the agent takes an action a_t following an ϵ-greedy strategy, i.e., a_t is chosen uniformly at random with probability $\epsilon(t)$ (called exploration), otherwise, it is chosen based on the Q-network as $a_t = \arg\max_a Q_\theta(s_t, a)$ (called exploitation). Here, $\epsilon(t)$ is a pre-defined parameter that decays over t to encourage exploration at the beginning and exploitation later on. After taking the action a_t, we collect the transition data (s_t, a_t, r_t, s_{t+1}) and add it to the Experience Replay memory, which stores the latest N transition data and will be used in the training phase.

- Data sampling. In each iteration of the training phase, we query B random samples from the Experience Replay memory. Then, for each sampled transition data (assume collected at time T), we compute the following target based on the Bellman equation in Equation (4.4).

$$y_T = r_T + \gamma \max_a Q_{\theta'}(s_{T+1}, a), \quad \forall\, T \in B. \tag{4.5}$$

  Here, $Q_{\theta'}$ corresponds to the so-called target Q-network, whose parameters θ' are copied from the original Q-network Q_θ every K time steps. The purpose is to decouple the original Q-network from the target and allows to properly use automatic back propagation in the model update later.

- Model update. With the computed targets, we build the following loss function associated with the sampled data.

$$L = \frac{1}{B} \sum_{T \in B} \big(y_T - Q_\theta(s_T, a_T)\big)^2. \tag{4.6}$$

  Then, we update the Q-network's parameters θ using back-propagation through the computed loss L.

Algorithm 12: Deep Q-Learning Algorithm

Initialize: Q-network θ, copy to target network $\theta' \leftarrow \theta$
for *Iterations* $t = 0, 1, \ldots$ **do**
  ▶ Take action a_t following the $\epsilon(t)$-greedy policy.
  ▶ Get data (s_t, a_t, r_t, s_{t+1}) and add to replay buffer.
  ▶ Sample a batch of B samples from the replay buffer and compute the target

$$y_T = r_T + \gamma \max_a Q_{\theta'}(s_{T+1}, a), \quad \forall\, T \in B.$$

  ▶ Update θ via back-propagation via the following loss.

$$L = \frac{1}{B} \sum_{T \in B} \big(y_T - Q_\theta(s_T, a_T)\big)^2.$$

  ▶ if $t \bmod K = 0$, $\theta' \leftarrow \theta$.
end

4.5 Experimental Results

We evaluated the performance of our reinforcement learning-based task scheduling system on an industrial static timing analysis (STA) application [63, 79] that exploits task graph parallelism to parallelize graph-based analysis (GBA). STA is a critical step in the overall EDA flow because it verifies the expected timing behavior of a circuit design and reports the critical paths that violate the given timing constraints (e.g., set-up, hold). As our system schedules task graphs, we used the state-of-the-art open-source STA engine, OpenTimer [6], to generate a task graph for us. OpenTimer formulates the GBA algorithm into a task graph. The task graph represents the corresponding circuit graph 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., slew,

delay, arrival time), while each edge represents a dependency between two tasks. After OpenTimer generates a task graph, our scheduler directly performs the scheduling on the task graph. Table 4.1 lists the statistics of the nine circuits we used.

We compiled programs using gcc-12 with `-std=c++17` and `-O3` enabled. We ran all the experiments on a Ubuntu 19.10 (Eoan Ermine) machine with 80 Intel Xeon Gold 6138 CPU at 2.00GHz and 256 GB RAM.

Baseline and Deep Q-Learning

For comparison, we implemented a baseline method based on the random action (RA) engine, which assigns each task to one of the 40 workers uniformly at random. Such a baseline method is widely used to schedule STA workloads. It randomly assigns the tasks to the workers without adapting to the dynamic and non-stationary computing environment.

To learn our RL-based task scheduler, we implemented Algorithm 12 to train the RL policy using a mixed graph composed of the following three graphs: aes_core, tv80 and c6288. Specifically, we implemented Algorithm 12 with the following set of hyper-parameters: batch size $B = 64$, target network synchronization period $K = 10$, reward discount factor $\gamma = 0.95$, reward weight $\alpha = 0.01$, and experience replay memory size $N =$ 10k. In particular, for the $\epsilon(t)$-greedy policy, we adopt the initialization $\epsilon(0) = 1.0$ (at $t = 0$) and multiply $\epsilon(t)$ by a factor of $\epsilon\_decay = 0.99998$ after every iteration. Also, we used a four-layer fully-connected neural network to parameterize the Q-function [55] and the network architecture is illustrated in Figure 4.3. To update the model parameters θ via back-propagation, we use the standard Adam optimizer with learning rate $\eta = 1e - 4$ [96].

Figure 4.4 plots the training loss (left figure) and the accumulated reward (right figure) achieved by the RL policy during the training process. From the left figure, it can be seen that the training loss decays quickly, indi-

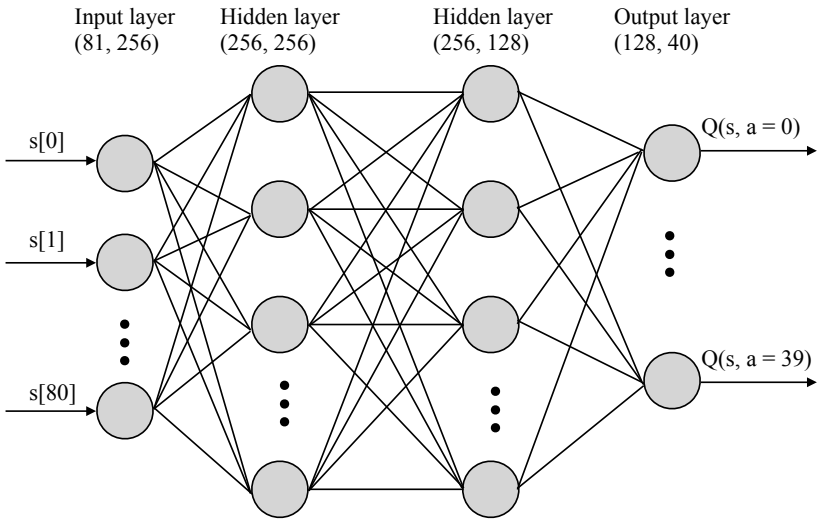


Figure 4.3: Illustration of the Q-network architecture. The network takes in the state vector as input (input dimension=81), then propagates it forward through 2 hidden layers and finally outputs the Q-values corresponding to each of the 40 possible actions (output dimension=40). The task at hand is then scheduled to the worker corresponding to the highest Q-value.

cating that the learned policy performs well on the training data. Moreover, the right figure shows that the RL policy eventually achieved an accumulated reward at around -6.0. Next, we further test the trained RL policy on some unseen test graphs and demonstrate its superior generalization performance.

Performance Comparison

We tested and compared the runtime performance of the baseline RA scheduler and our RL-based scheduler on various graphs with different configurations, and the results are summarized in Table 4.1. We note that the mixed graph used in the test phase is composed of the same three types of graphs (aes_core, tv80 and c6288) as those used in the training phase, but with different configurations. Hence, the mixed graph used in

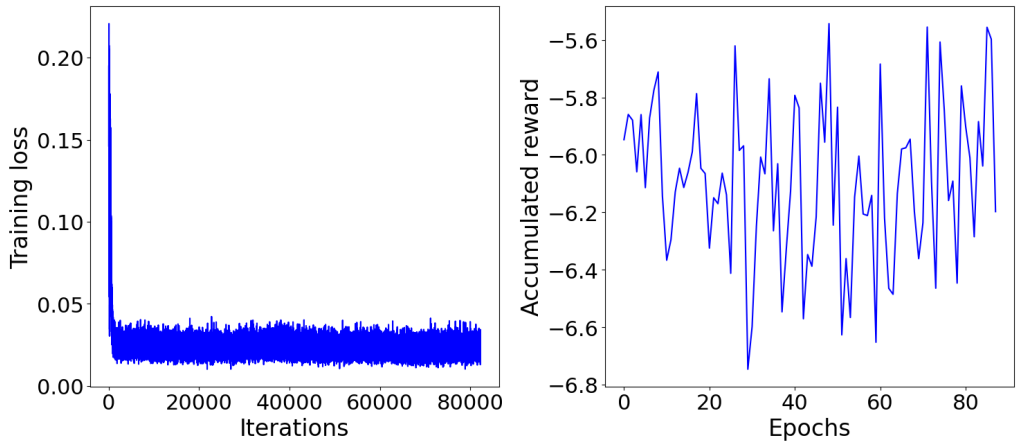


Figure 4.4: Left: Training loss v.s. iterations in the training. Right: Accumulated reward v.s. epochs in the training. Every epoch consists of 1K iterations, and the accumulated reward for each epoch is calculated by $R = \sum_{t=1}^{1000} \gamma^t r_t$.

the test phase is very different from the one used in the training phase.

Table 4.1: Runtime comparison between the random action (RA) scheduler and our reinforcement learning (RL) scheduler. $\|V\|$ and $\|E\|$ respectively denote the number of nodes and edges of a graph. Impr. denotes the performance of RL over RA.

Graph	$\|V\|$	$\|E\|$	Runtime (Seconds)			# Workers		
			RA	RL	Impr.	RA	RL	Impr.
mixed graph	88,626	115,777	38.44	**38.29**	0.39%	40	**7**	471%
aes_core	66,751	86,446	29.55	**28.89**	2.28%	40	**8**	400%
ac97_ctrl	42,438	53,558	18.92	**18.09**	4.59%	40	**8**	400%
tv_80	17,038	23,087	7.76	**7.04**	10.22%	40	**8**	400%
wb_dma	13,125	16,593	5.52	**5.33**	3.56%	40	**8**	400%
c6288	4,837	6,244	2.01	**1.98**	1.52%	40	**8**	400%
c7552_slack	3,802	4,791	1.75	**1.60**	9.38%	40	**7**	471%
usb_phy_ispd	2,447	2,999	1.11	**1.00**	11%	40	**7**	471%
s1494	2,292	2,925	1.04	**0.97**	7.22%	40	**7**	471%

From Table 4.1, we can see that the total runtime of our RL-based

scheduler is consistently slightly lower than that of the RA scheduler for all the test graphs. Surprisingly, these runtime results are achieved by our RL-based scheduler using only 7-8 workers, which are much more efficient compared to the RA scheduler that utilizes all of the 40 workers. Thus, the experimental results clearly demonstrate the advantage of our RL-based scheduler, indicating its superior memory efficiency and energy efficiency. This also implies that, through the reinforcement learning framework and our specialized reward design, the RL-based scheduler successfully learned a policy that enhances workload balance among the workers and reduces unnecessary data transfer cost.

In Figure 4.5 and 4.6, we further plot the distribution of tasks assigned to each worker under the RA scheduler and our RL-based scheduler for two task graphs, aes_core and mixed graphs, respectively. Specifically, we can observe from Figure 4.5 that, in order to schedule the tasks of the aes_core graph, the RA scheduler assigns tasks uniformly to all the 40 workers. As a comparison, our RL-based scheduler assigns tasks to only 8 workers, and moreover, most of the tasks are actually assigned to only 4 workers. We can make very similar observations in the Figure 4.6 for the mixed graph.

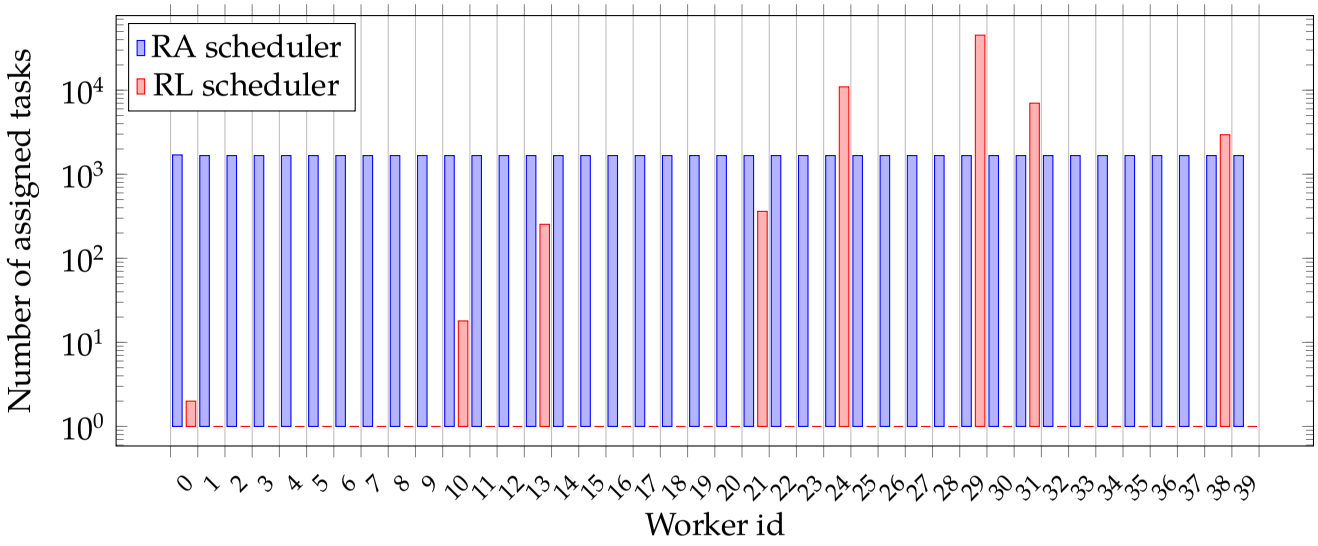


Figure 4.5: Histogram of assigned tasks per worker for the aes_core graph.

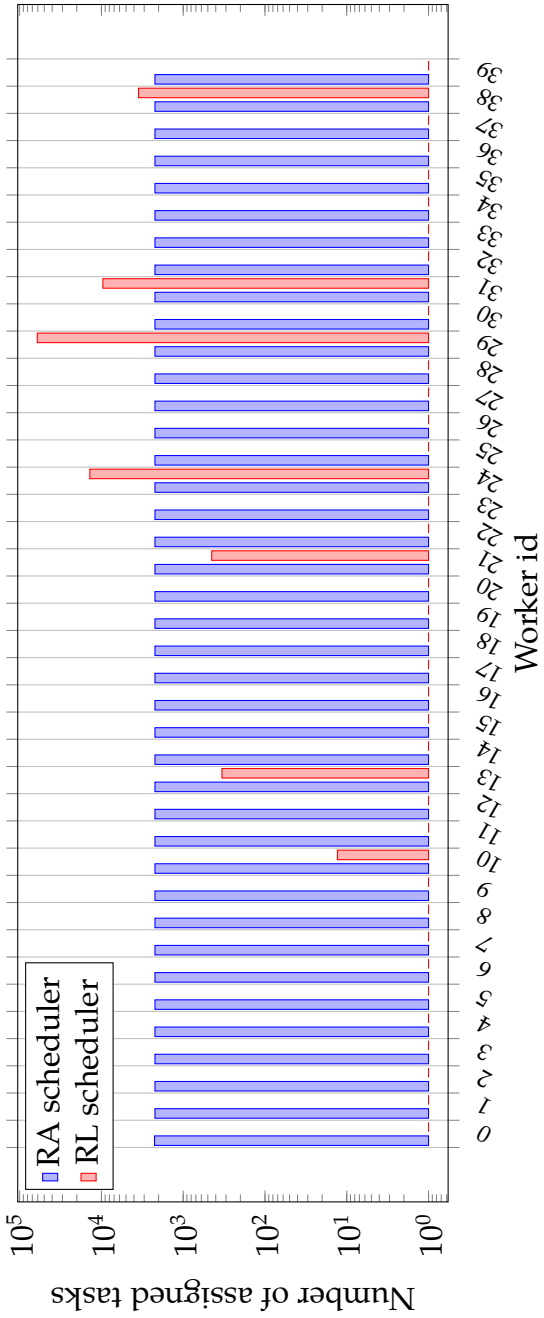


Figure 4.6: Histogram of assigned tasks per worker for the mixed graph.

4.6 Conclusion

In this chapter, we have introduced a resource-efficient reinforcement learning-based task scheduling system to adapt the performance optimization to the computing environment. We have evaluated our task scheduling system on an industrial static timing analysis workload. Compared to the popular heuristic-based scheduler, our RL-based scheduler achieved a lower runtime on all task graphs while using only 20% of workers. Our future work plans to extend our framework to a distributed environment [65, 66, 69, 70, 78, 114] and consider GPU task graphs [28, 29, 116] into our state model.

5 REINFORCEMENT LEARNING-GENERATED TOPOLOGICAL ORDER FOR DYNAMIC TASK GRAPH SCHEDULING

5.1 Abstract

Dynamic task graph scheduling (DTGS) has become a powerful tool for parallel and heterogeneous applications, such as static timing analysis and large-scale machine learning. DTGS allows applications to define the task graph structure on-the-fly, enabling concurrent task creations and task executions. However, to schedule tasks, DTGS relies on applications to define a topological order for the task graph. Existing algorithms for generating this order primarily rely on heuristics like level-by-level sorting, which lack adaptability to dynamic computing environments. This paper introduces a novel method that leverages reinforcement learning to generate topological orders for DTGS systems. We will delve into the details of our design and present a real-world use case. For instance, when scheduling a large task graph with 3.9 million tasks and 7.4 million dependencies in a large-scale static timing analysis workload, our method achieves a speedup of up to 1.52× compared to the baseline.

5.2 Introduction

Dynamic task graph scheduling (DTGS) has emerged as a powerful technique for processing parallel and heterogeneous applications, such as static timing analysis [19, 25, 26, 27, 29, 31, 34, 43, 45, 47, 48, 49, 50, 61, 62, 63, 64, 73, 79, 83, 114, 119, 120, 132, 152] and large-scale machine learning problems [28, 89, 116, 117]. Unlike traditional loop-based models that explores parallelism across loops, DTGS represents function calls as `tasks`

and dependencies between them as `edges` in a `task graph`. DTGS empowers applications to perform top-down optimization within complex parallel decomposition strategies involving numerous tasks and dependencies. A DTGS runtime then efficiently schedules these dependent tasks across a large pool of execution units with dynamic load balancing [113]. As a result, the parallel computing community has seen the rise of numerous successful DTGS libraries catering to various applications, such as OpenMP [5], Kokkos-DAG [35], PaRSEC [17, 58], HPX [93], Taskflow [72, 75], and AsyncTask [29].

To leverage the power of DTGS, applications define the task graph structure dynamically according to runtime variables and control-flow results. As tasks and dependencies are created on-the-fly, DTGS allows the task creation time to overlap with the task execution time, as shown in Figure 5.1. Thus, DTGS is flexible when dealing with many algorithms that frequently incorporate dynamic control flow in implementing irregular parallel decomposition strategies, such as electronic design automation (EDA) algorithms [63].

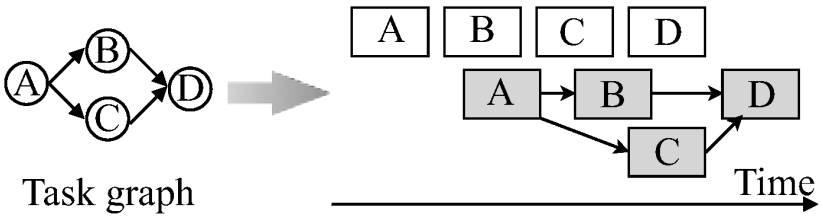


Figure 5.1: Scheduling a dynamic task graph with four tasks and four edges. White rectangles denote the task creations and gray the task executions. Tasks are created in the topological order `A-B-C-D`. Task creations overlap with task executions.

To schedule tasks under the task dependency constraints, DTGS runtime requires applications to create tasks in a topological order of the task graph. For example, in Figure 5.1, four tasks should be created either in the topological order `A-B-C-D` or `A-C-B-D`. To obtain the order, topological

sorting algorithms, such as Kahn's algorithm [4], are widely used. These heuristic-based algorithms generate orders primarily based on the graph structures, such as level-by-level sorting. However, such heuristic-based approaches have limitations. First, solely relying on graph structure lacks adaptability to dynamic changes in the computing environment. This can lead to suboptimal scheduling and can consume large scheduling resources due to the randomness involved in DTGS runtime's dynamic load balancing [113]. Second, heuristic algorithms generate deterministic orders. But, topological orders of a task graph are not unique and different topological orders for the same task graph can lead to substantial performance differences, as shown in Figure 5.2.

Figure 5.2: Runtime of DTGS system finishing one EDA application with three different topological orders.

To overcome the limitations of heuristic approaches, we leverage recent advancements in reinforcement learning (RL) [15, 36, 132] and develop a method to interact with the computing environment while generating topological orders. We summarize our technical contributions as follows:

- **Reinforcement Learning-based Framework**. We have designed a reinforcement learning-based method to generate topological orders for the dynamic task graph scheduling applications. With the

method, we are able to adapt to the computing environment throughout the whole decision-making process, allowing us to generate the topological order that achieves better runtime performance than the baseline.

- **Task Graph Encoding**. We have designed a new task graph encoding method to support our RL model. This technique allows us to encode the task graph dynamically according to the runtime status instead of encoding the whole graph.
- **Decision Space Categorization**. We have designed a new decision space categorization method to support our RL model. This method categorizes the timing-varying decision space into fixed number of categories, allowing us to reuse the trained RL model for various applications without the need for retraining.
- **Evaluations**. We have evaluated our method for generating the topological orders for a large-scale industrial static timing analysis (STA) application. With our method, we are able to generate topological orders that achieve up to 1.52× speedup over the baseline.

5.3 Background

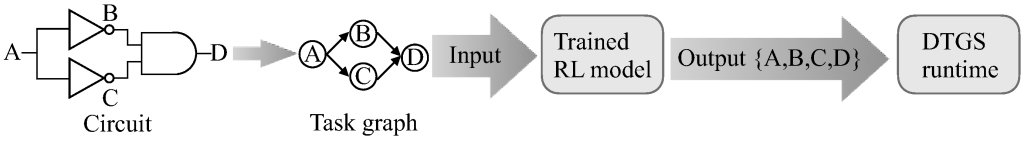


Figure 5.3: System overview. An example circuit is described as a task graph of four tasks and four edges. The trained RL model reads in the task graph and generates a topological order `A,B,C,D`. The DTGS runtime creates the four tasks in the order and executes them under the dependency constraint.

We target scheduling a large-scale static timing analysis (STA) application [63], one of the most important steps in the entire EDA flow, and describe a STA workload as a task graph. The task graph consists of multiple nodes and edges, which represent the tasks and the dependencies among the tasks, respectively. Every task has a known workload and the task dependencies constrain the execution order of the tasks. Take the task graph shown in Figure 5.3 as an example. We describe an example circuit as a task graph of four tasks and four edges (or dependencies). The task dependencies require that task `A` must execute before task `B` and task `C`, and task `D` must execute after task `B` and task `C`.

To efficiently schedule the task graph, we build a DTGS system using a recently released dynamic task graph library, AsyncTask [29]. AsyncTask provides a clear and concise graph description language for applications to easily explore dynamic task graph parallelism. The expressiveness of AsyncTask's programming model improves our productivity when coding large and complex task graphs for the STA workloads. Listing 5.1 shows AsyncTask implementation of the task graph in Figure 5.3. We create four tasks in the topological order `A-B-C-D` and use AsyncTask's `dependent_async` API to create each task. Every task defines its own lambda as the first argument which is followed by a list of dependent tasks. Upon returning from `dependent_async`, we obtain a pair consisting of an instantiated task object and a future object holding the execution result of that task. After constructing all tasks, we call `future_D.get` to wait for task `D` to finish. Since we construct tasks in the order `A-B-C-D`, the completion of task `D` in turn signifies the completion of all preceding tasks.

```
int main(){
  Executor executor;

  // Create task A
  auto[A,future_A]=executor.dependent_async(
    [](){ printf("Running task A\n"); });
```

```
  // Create task B, which has A as the dependent task
  auto[B,future_B]=executor.dependent_async(
    [](){ printf("Running task B\n"); }, A);

  // Create task C, which has A as the dependent task
  auto[C,future_C]=executor.dependent_async(
    [](){ printf("Running task C\n");}, A);

  // Create task D, which has B and C as the dependent tasks
  auto[D,future_D]=executor.dependent_async(
    [](){ printf("Running task D\n"); }, B, C);

  // Wait for the task graph to finish via future_D
  future_D.get();
}
```

Listing 5.1: AsyncTask implementation of the task graph in Figure 5.3.

In Listing 5.1, we must create tasks in a topological order because we can not create a task until certain tasks it depends on have existed. For example, we need to create task `A` before task `B`. That explains why AsyncTask requires applications to create tasks in a topological order. To obtain a topological order of a task graph, we design a method that incorporates a reinforcement learning (RL) model in DTGS. Figure 5.3 illustrates the system overview. After we describe a circuit as a task graph, the trained RL model reads in the task graph and outputs a topological order of the task graph. Then the DTGS creates all of the tasks based on the order and overlaps the task executions. We give the details of the trained RL model in Section 5.4.

5.4 Our Method

Runtime status governs the macro-scale performance in a task scheduling system [132]. To take the runtime status into account, we develop a reinforcement leaning model to interact with the computing environment while generating a topological order for the DTGS application. In this section, we first formulate the dynamic task graph scheduling problem as a reinforcement learning (RL) problem and then apply the Deep Q-Learning algorithm [131] to train a good RL policy. Figure 5.4 shows the training process in the DTGS application.

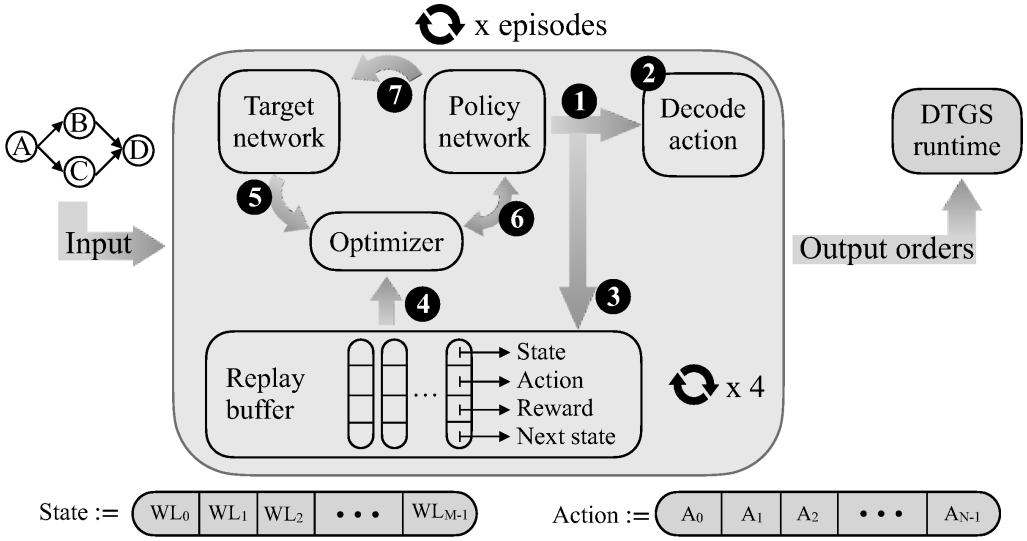


Figure 5.4: Overview of the training process. The input is a task graph of four tasks and four edges. The output is a topological order of the four tasks. The training process consists of seven steps, which iterates four times as there are four tasks in the input task graph. The whole training would iterate the task graph for several episodes.

Reinforcement Learning Formulation

To formulate the dynamic task graph scheduling problem as a reinforcement learning problem, we need to define four major components as follows:

- State. A state encodes the information that the RL agent needs to suggest an action. We encode the normalized workloads of ready tasks (tasks whose dependencies are resolved) and the graph structure in a vector of `M` coordinates as the state called `State`. `M` denotes the number of maximum fanout (or the successor tasks) of a task in the task graph. Each coordinate encodes the total workloads (`WL`) of ready tasks of the same number of fanout. For example, in Figure 5.4, task `A` is the ready task and has two fanouts – task `B` and task `C`. The current state would include task `A`'s workload in the third coordinate, `State[2]`, and the other coordinates are zero.

- Action. An action describes the operation that the RL agent suggests. In this paper, the RL agent needs to select the next task from all the ready tasks. We note that the number of the ready tasks may vary over time. However, the RL agent can only select a task from a *fixed* number of ready tasks. To accommodate the constraint, we categorize the ready tasks to `N` groups based on the workloads. For example, in Figure 5.4, when the RL agent needs to select the next task between task `B` (suppose being categorized to group 0) and task `C` (suppose being categorized to group 1), the RL agent selects the next task from a certain *group* (say group 0) rather than selecting the next task directly. This allows us to maintain a consistent action space size regardless of the number of ready tasks at any given time.

- State Transition. After an action is performed (i.e., the next task is created by the DTGS runtime), the current state will transfer to a

new state. For example, in Figure 5.4, after task `A` is created by the DTGS runtime, the new state will have the sum of task `B`'s and `C`'s workloads at the second coordinate (`State[1]`) because both task `B` and `C` are now ready tasks and have one fanout.

- Reward. After the RL agent takes an action, we receive a reward feedback from the computing environment. This reward signal guides the agent's learning process towards actions that optimize a specific resource utilization metric. Here, we focus on minimize free memory space (FMS) to keep all computing units as busy as possible. Therefore, we design the following reward to reflect this objective:

$$\text{reward} = -(\text{FMS}^{\text{after}} - \text{FMS}^{\text{before}}), \tag{5.1}$$

where $\text{FMS}^{\text{before}}$ denotes the normalized FMS before performing the action and $\text{FMS}^{\text{after}}$ denotes the normalized FMS after the action. Note that minimizing FMS is equivalent to maximizing the reward.

Next, we discuss how to train a good RL policy using the Deep Q-Learning algorithm to maximize the accumulated reward over time.

Deep Q-Learning

Deep Q-Learning is a popular algorithm that aims to learn the optimal policy that maximizes the expected accumulated reward over time [131]. We apply this algorithm to solve our dynamic task graph scheduling problem which is now formulated as a RL problem. Figure 5.4 illustrates the Deep Q-Learning algorithm for our scheduling system in seven steps.

❶ **Generate an action**. Based on the current state, the policy network generates an action. The policy network is a feed forward neural network, as shown in Figure 5.5. The network reads in the current state vector with dimension M as the input. It then processes the information through two

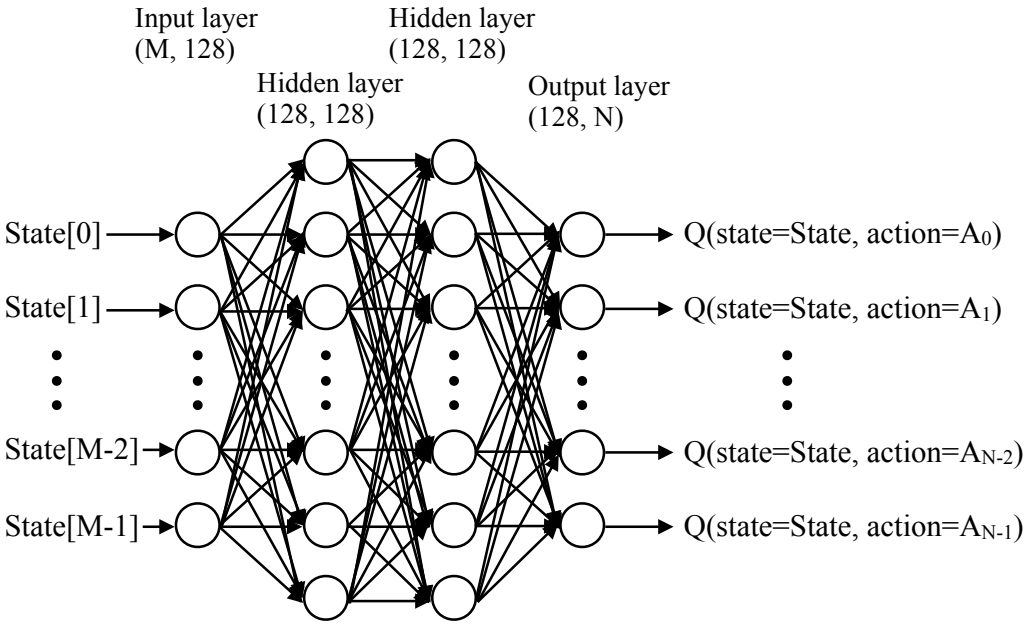


Figure 5.5: Illustration of the policy network architecture. There is one input layer of dimension $M * 128$, two hidden layers of dimension $128 * 128$, and one output layer of dimension $128 * N$. The activation function is ReLU [7].

hidden layers, each with a dimension of 128×128. Finally, the network outputs a set of Q-values corresponding to each of the N possible actions. A Q-value represents the expected reward associated with taking a specific action in a given state. In essence, it estimates the long-term benefit of choosing an action. To balance exploration and exploitation during learning, we employ the ϵ-greedy strategy. During exploration (with probability ϵ), we randomly select an action from the available options. This helps the RL agent discover potentially better actions outside of its current knowledge. In contrast, during exploitation (with probability $1 - \epsilon$), we select the action with the highest Q-value, favoring actions predicted to yield the best rewards in the current state.

The parameter ϵ gradually decays over time, following the equation

below,

$$\epsilon = \epsilon^{end} + (\epsilon^{start} - \epsilon^{end}) * e^{-1*steps/\epsilon^{decay}}, \tag{5.2}$$

where $steps$ records the number of processed steps so far, ϵ^{start} denotes the initial value, ϵ^{end} denotes the final value, and ϵ^{decay} denotes the decay rate. This decay encourages exploration in the initial stages to learn the environment and transitions to exploitation later for optimal performance.

As discussed in Section 5.4, an action represents a group of ready tasks with the same range of workloads. If the selected group is empty (meaning no ready tasks fall within that workload category), we implement the following strategies. We randomly select another action at the exploration phase, or select the group with the next highest Q-value at the exploitation phase.

2 **Decode the action to the next task**. Upon selecting a non-empty group (i.e., containing ready tasks), we randomly select a task from that group and forward the selected task to the DTGS runtime for task creation and execution.

3 **Store the transition information in the replay buffer**. To improve the agent's learning efficiency, we utilize a replay buffer. This buffer stores transitions as tuples, allowing the agent to revisit and learn from past experiences multiple times. Each tuple in the replay buffer contains four key elements:

- Current State (s): The state representation captures information relevant to the decision-making process at a specific point in time.
- Selected Action (a): The action is selected by the agent based on the current state.
- Reward (r): The reward signal is received from the environment after taking the selected action. This feedback guides the agent towards actions that optimize resource utilization (here, we minimize the free memory space).

- Next State (s'): The state representation after the selected action is executed, reflecting the updated environment.

By revisiting these transitions during training, the agent can learn from a broader set of experiences and improve its decision-making capabilities.

4 **Sample the replay buffer**. During training, we leverage batch sampling to learn from its past experiences stored in the replay buffer. This involves randomly selecting a mini-batch of data points (size denoted by B) from the buffer. By randomly selecting data points, we help to de-correlate the training samples. This is important because consecutive transitions in the replay buffer might be highly correlated, potentially hindering the learning process. De-correlated samples provide a more diverse set of experiences for the agent to learn from, improving the efficiency and effectiveness of training.

5 **Calculate the expected Q-value**. We incorporate a target network, which mirrors the architecture of the policy network (as illustrated in Figure 5.5). However, we do not update the target network's weights as frequently as the policy network's. This separation is crucial for reducing overestimation bias [103]. The target network addresses this issue by providing an unbiased estimate of the Q-value for the next state. We achieve this by using the target network to evaluate the expected Q-value of the action selected by the policy network, as shown in the following equation:

$$Q^{expected} = r + \gamma \max_{a} Q(s', a) \tag{5.3}$$

where γ represents the discount factor, which balances the importance of immediate rewards against the potential value of future rewards.

6 **Optimize the model**. During training, we calculate the difference (loss) between the estimated Q-value for the current state and action (denoted as $Q(s, a)$) and the target network's unbiased estimate of the expected Q-value for the next state (denoted as $Q^{expected}$). This loss

represents how well the policy network's predictions align with reality, and we aim to minimize it for effective learning. The equation for calculating loss δ is the following,

$$\delta = Q(s, a) - Q^{expected}. \tag{5.4}$$

To address the issue of outliers in noisy Q-value estimates, we employ the Huber loss function [2]. Unlike the standard quadratic loss, the Huber loss is less sensitive to extreme values, making it a robust choice for this scenario. The mathematical definition of the Huber loss is shown below,

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,r,s') \in B} \mathcal{L}(\delta), \tag{5.5}$$

where

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{if } |\delta| \leqslant 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

Once we calculate the loss $\mathcal{L}$, we leverage back-propagation to propagate the error signal through the policy network. This process guides the network in adjusting its internal parameters to minimize the loss in future predictions. We utilize the Adam optimizer [95] with a learning rate of LR.

7 **Update the target network**. To further enhance the stability and performance of the learning process, we employ *soft update* for the target network [110]. This approach balances the target network's weights between those of the policy network and its own past values. The equation to update the parameters of the target network is the following,

$$\theta' = \tau\theta + (1 - \tau)\theta', \tag{5.6}$$

where θ' denotes the weights of the target network, θ denotes the weights

of the policy network, and τ denotes the update rate of the target network.

In Figure 5.4, the task graph consists of four tasks. We iterate through the seven steps four times, constituting one episode for the agent. By running multiple episodes `episodes` (iterating through the entire process several times), the agent has the opportunity to learn more effectively from the environment and refine its policy for optimal task scheduling.

5.5 Experimental Results

We used the recently released dynamic task graph programming library *AsyncTask* [29] to implement the dynamic task graph scheduling system. We trained the RL model using Pytorch and compiled programs using g++11.4 with `-std=c++20` and `-O3` enabled to schedule the task graphs. We evaluated the runtime performance of scheduling tasks in the topological orders generated by the reinforcement learning model. 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 10 runs.

Baseline

We selected a heuristic approach as the baseline. This approach involves traversing a task graph and identifying a task whose dependencies have all been resolved. If multiple such tasks exist, we employ one of two predefined heuristics throughout the entire process. In heuristic 1, a task is selected randomly from the set of ready tasks. In heuristic 2, the task with the highest number of outgoing dependencies (fanout) is selected. In addition, we used Kahn's algorithm [4] as the third heuristic. It's important to note that we don't dynamically switch between these heuristics. Algorithm 13 details the implementation of the chosen approach.

Algorithm 13: baseline(task_graph)

```
   Input: task_graph: a task graph
   Output: order: a topological order
 1 order ← ∅;
 2 array ← ∅;
 3 /* push ready task to array                                    */
 4 for task ∈ task_graph do
 5 |   /* in_degree denotes the number of the fanin               */
 6 |   if task.in_degree == 0 then
 7 |   |   array.push(task);
 8 |   end
 9 end
10 while order.size < task_graph.tasks.size do
11 |   /* Used either Heuristic 1 or 2                            */
12 |   /* Heuristic 1                                             */
13 |   task ← pop one task randomly in array;
14 |   /* Heuristic 2                                             */
15 |   task ← pop one task with most fanout in array;
16 |   order.push(task);
17 |   /* Resolve dependencies for fanout tasks                   */
18 |   for ftask ∈ task.fanout do
19 |   |   ftask.in_degree ← ftask.in_degree − 1;
20 |   |   if ftask.in_degree == 0 then
21 |   |   |   array.push(ftask);
22 |   |   end
23 |   end
24 end
25 return order
```

Static Timing Analysis Workload

We used the industrial static timing analysis (STA) as the workload [63, 79], which exploits task graph parallelism to parallelize graph-based analysis. STA is representative of many analysis-driven EDA applications, and is a critical step in the overall EDA flow because it verifies the expected timing behavior of a circuit design and reports the critical paths that vi-

olate the given timing constraints (e.g., set-up time, hold time). As our system schedules task graphs, we used the state-of-the-art open-source STA engine, OpenTimer [6], to generate task graphs for us. OpenTimer formulates the graph-based analysis (GBA) algorithm into a task graph. The task graph represents the corresponding circuit graph 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., slew, delay, arrival time), while each edge represents a dependency between two tasks. Table 5.1 lists the statistics of the 12 task graphs we used. $\|V\|$ denotes the number of the tasks in a task graph and $\|E\|$ denotes the number of the edges.

Table 5.1: Task ($\|V\|$) and edge ($\|E\|$) counts of 12 task graphs.

Graphs	$\|V\|$	$\|E\|$	$\|V\| + \|E\|$
des_perf	371,587	464,810	836,397
vga_lcd	397,809	498,863	896,672
mgc_edit_dist	450,354	566,527	1,016,881
vga_lcd	679,258	823, 034	1,502,292
b19	782,914	1,048,609	1,831,523
b19_2	782,914	1,048,609	1,831,523
leon3mp	3,376,832	6,277,562	9,654,394
b19_3	3,914,570	5,243,045	9,157,615
netcard	3,999,174	7,404,006	11,403,180
leon2	4,328,255	7,984,262	12,312,517
leon3mp_2	6,753,664	8,297,576	15,051,240
netcard_2	7,998,348	9,806,794	17,805,142

Training and Hyper-parameters

We trained the RL policy with six task graphs, `tv80`, `ac97_ctrl`, `des_perf`, `usb_phy`, `c1355`, and `s1196`. It's important to note that the `des_perf` graph

used in training was a different instance from the one used in testing. The hyper-parameters we used for training are the followings:

- The update rate of the target network, τ, is 0.005.
- The value of discount factor, γ, is 0.99.
- The number of training iterations, $episodes$, is 50.
- The initial value of ϵ, ϵ^{start}, is 0.9.
- The final value of ϵ, ϵ^{end}, is 0.05.
- The decay rate of ϵ, ϵ^{decay}, is 1000.
- The number of input neurons, M, is 20.
- The number of output neurons, N, is 6.
- The number of batch size, B, is 128.
- The learning rate of Adam optimizer, LR, is $1e-4$.

Scheduling Task Graphs

After obtaining the topological order for a task graph using the trained RL policy and the baseline, we used *AsyncTask*'s *dependent_async* API to create the tasks in the generated topological order [29]. This API takes two arguments. The first argument is the callable function representing each task. This function could perform various operations like parasitic calculations, slew adjustments, delays, or arrival time calculations in STA. The second argument is a list of dependent tasks.

Runtime Performance Comparison

Figure 5.6(a) compares the runtime performance of our approach against the baseline. We only report the best runtime performance between two heuristic methods for the baseline. We can see that the baseline exhibits faster execution only for small-sized task graphs. For example, the baseline is 1.06× and 1.02× faster in `des_perf` and `mgc_edit_dist`, respectively, although these differences are minor. Conversely, our RL model outperforms the baseline in all other 10 task graphs. For example, ours is 1.24×, 1.52×, and 1.48× faster than the baseline in `b19`, `netcard`, and `leon2`, respectively. We believe this advantage stems from our RL model's ability to optimize for free system memory. Our approach adapts to changing computing environments and reduces scheduling resource consumption associated with AsyncTask's dynamic load balancing. In contrast, the baseline relies solely on the task graph structure, neglecting runtime information. The heuristic-based baseline can achieve good performance on small-sized graphs (i.e., `des_perf` and `mgc_edit_dist`). However, as graphs grow larger and more complex, with more concurrent tasks and scheduling resource consumption, the baseline's static strategy hinders adaptation, leading to performance degradation. Figure 5.6(b) visualizes the speedup achieved by our RL approach over the baseline. The speedup increases with graph size and complexity, further highlighting the superiority of our method particularly in modern industrial EDA flow that frequently requires analyzing the same task graphs multiple times.

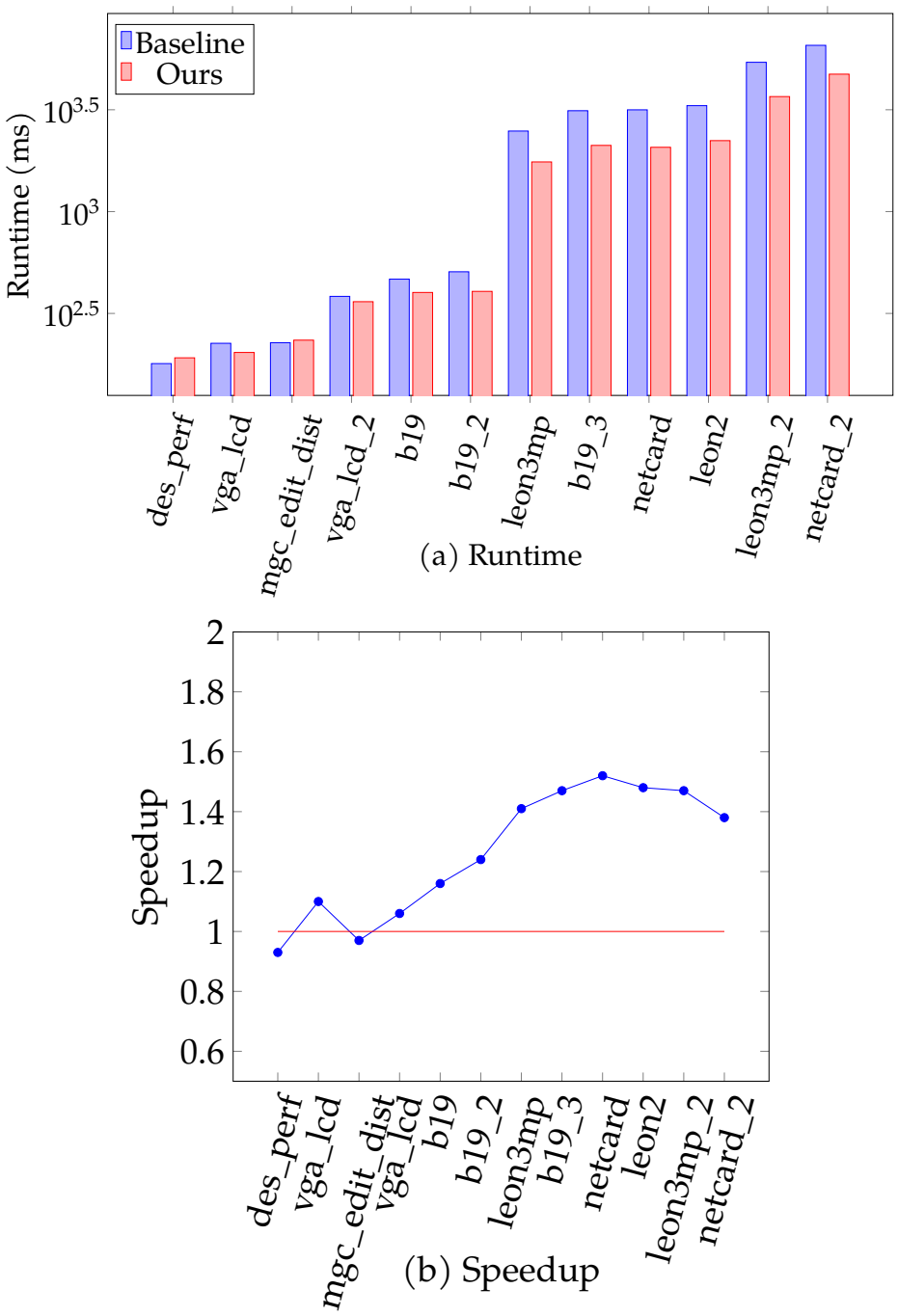


Figure 5.6: Performance comparison between the baseline and our RL approach on running 12 task graphs. (a) Runtime comparison. (b) Speedup of our approach over the baseline. The red horizontal line denotes the speedup of one.

5.6 Conclusion

In this chapter, we have introduced a reinforcement learning model to adapt to the dynamic runtime environment and generate a topological order for a task graph application with a better runtime performance running on a work-stealing runtime. In the future, we will apply our approach to other parallel graph algorithms [24, 91, 118, 145].

6 OPTIMIZING CUDA GRAPH SCHEDULING WITH REINFORCEMENT LEARNING: A CASE STUDY IN SSTA PROPAGATION

6.1 Abstract

CUDA Graph has shown potential in recent GPU-accelerated statistical static timing analysis (SSTA) propagation applications. By representing dependent SSTA tasks as a task graph and reusing the execution flow, CUDA Graph eliminates repetitive kernel launch overhead and improves task asynchrony. This enables more efficient scheduling of SSTA propagation tasks across logic gates. However, application-given CUDA graphs are often suboptimal, as they focus on capturing circuit structures while overlooking GPU resource availability and scheduling constraints. Unfortunately, the latter heavily relies on the CUDA Graph runtime, which is essentially a black box. To tackle this challenge, we introduce a Reinforcement Learning (RL)-based framework that optimizes CUDA graphs by learning to restructure SSTA graphs through interactions with the CUDA Graph runtime. Specifically, we formulate graph restructuring as a node-level adjustment problem and solve it by dynamically appending auxiliary edges to the graph during RL decision-making. To enable more informed decisions for our RL agent, we leverage Graph Neural Networks (GNNs) to encode both the graph structure and the application needs. Compared to the original application-given CUDA graph, our optimized CUDA graph can achieve up to a 12% runtime improvement.

6.2 Introduction

Statistical static timing analysis (SSTA) is a critical step in Electronic Design Automation (EDA) as it enables more accurate delay estimation than traditional static timing analysis (STA) by modeling on-chip process variation (OCV) as random variables [14, 37]. For example, [85] uses numerical integration to estimate circuit yield by exploring device parameter combinations, while [147] models gate delays as random variables and propagates rise and fall arrival time statistically through the timing graph. As design complexity continues to grow, manufacturing variations have introduced a broad range of OCVs that SSTA algorithms must evaluate during propagation. Despite the daunting computational cost, many computations are structurally independent across gates, transitions, and variation dimensions, revealing substantial opportunities for *data parallelism*. This parallelism makes SSTA propagation well-suited for GPU acceleration, which has emerged as a promising solution to meet its growing performance demands [23, 42, 45, 49, 121].

However, conventional GPU execution models (e.g., CUDA streams) face significant challenges in efficiently scheduling SSTA workloads. In practice, SSTA workloads involve repeated propagation over the circuit graph across different inputs and statistical values [14]. This leads to frequent kernel launches, which can accumulate to expensive synchronization costs when kernels are iteratively offloaded through one or more CUDA streams. To address this challenge, the recent state-of-the-art [23] leverages *CUDA Graph* to model SSTA propagation as a *GPU task graph*. Specifically, instead of launching kernels individually, CUDA Graph allows the execution flow to be constructed once and replayed multiple times with minimal CPU intervention, eliminating redundant kernel launches and reducing synchronization costs. This particularly benefits many SSTA propagation algorithms, where similar computational patterns are repeatedly executed across different timing scenarios [23].

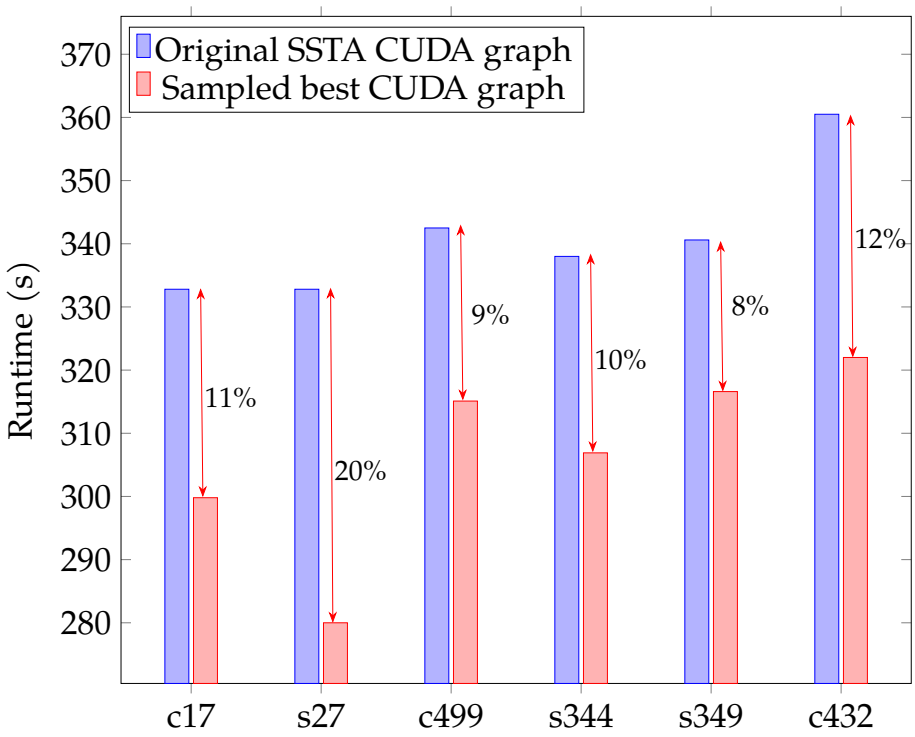


Figure 6.1: The original SSTA CUDA graphs leave at least 8% to 20% performance on the table, with the baseline derived from the minimum of 10,000 sampled graphs.

Despite the runtime improvement of CUDA Graph on SSTA workloads [23], application-given CUDA graphs are often suboptimal. For instance, in Figure 6.1, we evaluate six SSTA benchmarks by generating multiple variants of the application's original CUDA graph. Each sampled graph is created by randomly inserting a small number of auxiliary edges to restructure the application's original CUDA graph. We insert edges rather than delete them, as insertions satisfy the execution order of the application's original CUDA graph. In contrast, deletions break the physical interconnections between circuit gates and therefore violate the execution order. When comparing runtime performance, we observe that the original graph was 11% slower on *c17* and 20% slower on *s27*

than the best-performing sampled graph. This highlights the potential for graph restructuring to enhance execution efficiency. A key factor behind this performance gap is that the application's CUDA graphs prioritize capturing circuit structure but overlook GPU resource availability and scheduling constraints. This oversight often results in resource contention (e.g., multiple tasks competing for limited GPU resources) and reduced task parallelism, as tasks are forced to wait instead of executing concurrently. Unfortunately, the CUDA Graph runtime operates as a black box, with scheduling details hidden as proprietary information. This constraint makes it challenging to design a general-purpose heuristic that can optimize SSTA CUDA graphs across different GPU environments.

Despite the black-box challenge, this problem is particularly well-suited for Reinforcement Learning (RL) [30, 36, 131, 132], as RL can efficiently explore the complex graph search space and adapt to hidden scheduling behaviors through interactions with the CUDA Graph runtime. For instance, existing work [132] uses RL to adaptively optimize task scheduling for resource efficiency, while DRAS [36] leverages RL to automatically learn and converge to optimal scheduling policies in HPC clusters. Inspired by the success of RL-based schedulers in adaptively optimizing decisions under complex constraints and dynamic runtime conditions, we introduce an RL-based framework to optimize CUDA Graph scheduling for SSTA propagation workloads. We summarize our technical contributions as follows:

- **New Scheduling Description**. We formulate the CUDA Graph scheduling on SSTA propagation workloads as a node-level adjustment problem. With this problem formulation, we transform a complex scheduling challenge into a learnable problem that adjusts node levels in the application's CUDA graph.

- **Informed Decision Making**. We leverage Graph Neural Networks

(GNNs) to capture structural information from both the application workload and the CUDA graph, enabling informed decisions and enhancing the scheduling optimization process.

- **Reinforcement Learning-based Framework**. We introduce an RL-based framework to solve the node-level adjustment problem. By interacting with the CUDA Graph runtime, the RL agent adaptively learns to restructure the graph, ultimately generating an optimized CUDA graph for improved scheduling performance.

We have evaluated our framework on a set of industrial SSTA benchmarks [23]. For an application-given CUDA graph (i.e., original input CUDA graph), which mainly considers circuit graph structures, our framework generates an optimized CUDA graph by learning to restructure the original input CUDA graph through interactions with the CUDA Graph runtime, achieving up to 12% runtime improvement over the application-given CUDA graphs. Notably, our framework requires no changes to application-level algorithms, but instead restructures the given CUDA graph to guide the CUDA runtime toward better scheduling performance.

6.3 Scheduling SSTA Propagation Graph on GPU

In this paper, we consider the SSTA propagation workload in [23] as our problem formulation: Given an SSTA propagation graph, as illustrated in Figure 6.2(a), timing variations for gates and interconnections are modeled with random variables and stored in arrays, including voltage fluctuations (ΔV), temperature shifts (ΔT), channel length deviations (ΔL), and so on. In order to deal with various corner cases, each gate has up to 65536 data

points where each point represents one statistical phase.[1] During timing propagation through the circuit, each gate contributes the gate delay to the arrival times at its fan-in edges using statistical min/max operations. Delay computations are repeatedly performed for early and late modes, as well as for rise and fall transitions, across multiple corner cases within the SSTA propagation graph. Due to their structural independence across gates, rise/fall transitions, and variation dimensions, these computations exhibit a high degree of data parallelism and are well-suited for GPU acceleration.

[1]This problem formulation originates from a real-world challenge faced by our industry partners at a leading EDA company. While proprietary details cannot be disclosed, we abstract the core scheduling difficulties and practical constraints into a high-level formulation.

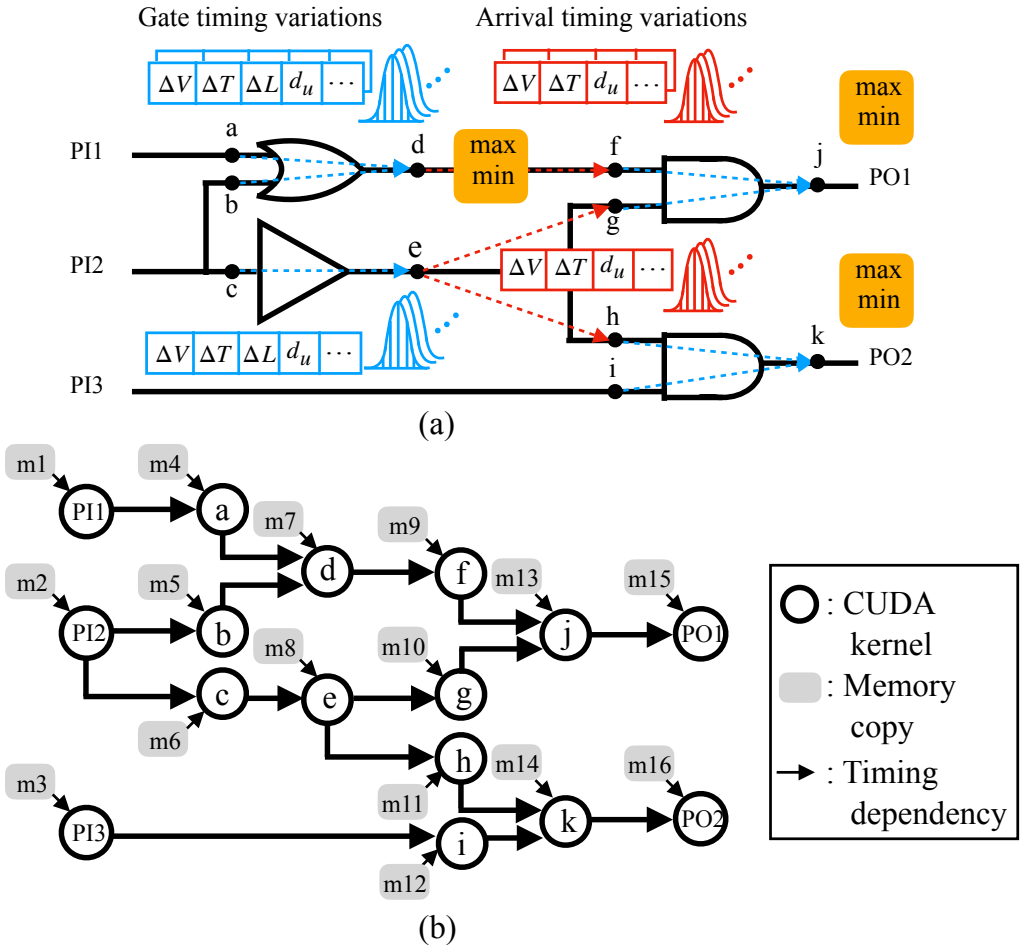


Figure 6.2: (a) An SSTA propagation graph. Gate timing (blue) and arrival timing (red) are modeled as random variables in arrays and propagated through the circuit graph using statistical max and min operators. (b) The corresponding CUDA graph of (a). Circles are kernel operations and gray rectangles are memory copy operations.

Figure 6.2(b) illustrates how [23] leverages CUDA Graph to execute an SSTA propagation graph from Figure 6.2(a). The algorithm (1) models circuit pins as CUDA kernels and timing dependencies as edges to capture the graph structure, and (2) inserts a unique memory copy operation before each kernel to transfer the corresponding statistical data points

from an array. After the construction, the algorithm outputs a CUDA graph that represents the SSTA propagation graph. Our goal here is to restructure the CUDA graph provided by the application. Instead of modifying application-level algorithms or kernels, we insert a small set of auxiliary edges to guide the CUDA Graph runtime toward better scheduling performance.

6.4 Our Framework

We introduce an RL-based framework to generate an optimized CUDA graph of SSTA workloads through interactions with CUDA Graph runtime. We formulate the generation of an optimized CUDA graph as a node-level adjustment problem, which simplifies the scheduling to an appending of edges in the graph and is easy for the RL agent to learn in the decision-making process. Specifically, we move nodes to new levels through the insertion of auxiliary edges, which results in a new CUDA graph without violating the topology order of the original graph description. Figure 6.3 shows this formulation.

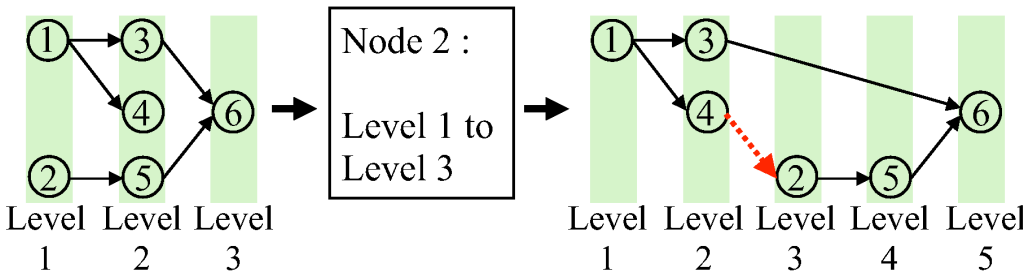


Figure 6.3: Illustration of the node-level adjustment formulation. The red edge moves node 2 from level 1 to level 3, resulting in a new graph that potentially alleviates the contention among nodes 3, 4, and 5.

With the node-level adjustment formulation, our RL-based framework comprises two modules. The first module generates a latent *node embed-*

ding, encapsulating both node attributes and graph topology. The second module uses the *node embedding* to adjust node levels and generate a new CUDA graph for the CUDA Graph runtime to execute. The framework learns from the feedback returned by the CUDA Graph runtime to iteratively improve our CUDA graph. Figure 6.4 illustrates an overview of our framework.

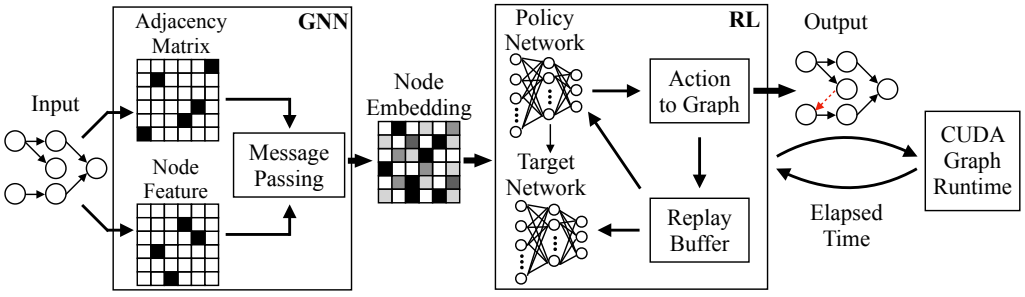


Figure 6.4: Overview of the framework. The framework consists of two modules: The first module is GNN and is used to encode the node attributes and graph topology and generate a latent representation *node embedding*. The second module is an RL model that uses the *node embedding* as a state and generates a new CUDA graph for the CUDA Graph runtime to run.

Graph Neural Network Module

To generate a better CUDA graph, we need structural information in making the decisions. The information we consider includes both node attributes, which capture the inherent characteristics of each kernel, and the graph topology, which reflects the dependencies and connectivity between kernels. To encode the information, we employ GNN [151] as the first module, as shown in Figure 6.4. The GNN module comprises three essential components: *adjacency matrix*, which encodes the connectivity between nodes; *node feature*, which represents the initial attributes of each node; and *message passing* mechanism, which enables information propagation

across the graph. Using these components, the GNN module effectively captures complex relationships and dependencies within the CUDA graph, providing a rich representation for subsequent decision-making.

Adjacency Matrix

The *adjacency matrix* is the first component and is used to define the connectivity structure of the graph and guide the process in the third component *message passing*. The matrix represents the relationships between nodes. A non-zero entry at position (i, j) indicates the presence of an edge pointing from node i to node j, while a zero entry signifies the absence of such a connection. In the third component *message passing*, the *adjacency matrix* acts as a filter, determining which nodes exchange information with each other.

Node Feature

The *node feature* is responsible for incorporating both node attributes and graph topology into a unified representation. Each node in the *input graph* represents a kernel operation (preceded by an implicit memory copy) and is characterized by six distinct elements. Three of these elements represent the node's resource requirements. `Thread counts` indicate the number of threads needed for a kernel execution, `block counts` indicate the number of blocks required, and `memory copy size` represents the data size copied from the CPU to the GPU for that node. The remaining three elements encode the graph topology, capturing both local and global properties of the graph. `Node level` indicates the node's level within the graph, `fanin count` represents the number of incoming edges, and `fanout count` represents the number of outgoing edges. Figure 6.5 illustrates the node feature for all nodes. To avoid larger-magnitude features from overshadowing others and ensure that all features contribute equally to the learning process, we normalize every element between 0 and 1.

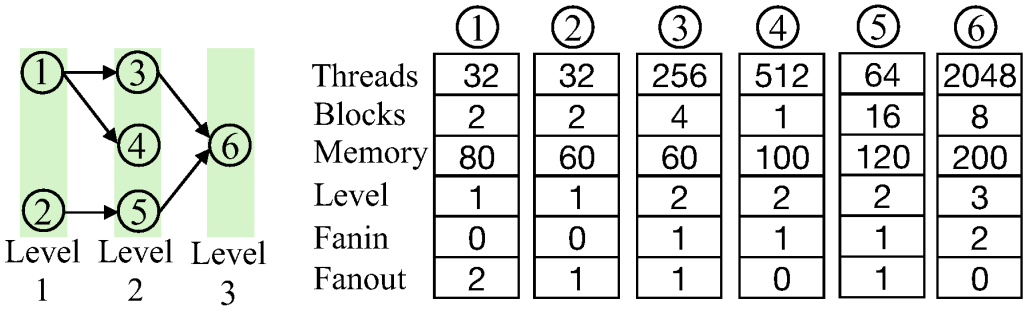


Figure 6.5: Example of the node feature for all nodes on the left graph.

Message Passing

The *message passing*, the third component of our GNN, serves as the fundamental mechanism for encoding *node feature* and generating the latent representation *node embedding*. This process involves iteratively propagating information between neighboring nodes, enabling the network to learn complex relationships within the graph. During each iteration, each node aggregates information from its neighbors' previous representations and combines it with the node's own features. For example, for the graph in Figure 6.5, node 6 aggregates information from both nodes 3 and 5. This aggregation process allows information to flow between nodes across the graph. By repeating this message passing procedure multiple times, we can capture increasingly long-range dependencies, allowing the network to learn sophisticated representations that reflect the global structure of the graph. In this framework, we employ a `two-hop` message passing approach, meaning every node propagates its information to nodes two hops away (e.g., node 2 propagates information to node 6 in Figure 6.5). We did not consider approaches with higher hops because nodes around the adjacent levels are more likely to compete GPU resources together (e.g., node 1 to node 5 compete with node 6). We utilize a popular graph convolutional network (GCN) [97] as the underlying network architecture because we do not observe significant runtime difference in our evaluations using

other architectures, such as GraphSAGE [54].

Reinforcement Learning Module

We leverage an RL agent as the second module to learn to generate an optimized CUDA graph through interactions with the CUDA Graph runtime. The RL agent receives *node embedding*, which encapsulates structural information, from the GNN module. Based on these embeddings, the agent suggests actions, which correspond to node-level adjustments, ultimately resulting in a new CUDA graph. After the CUDA Graph runtime executes the new graph, the agent receives the execution time as feedback. We model this learning process using RL's four fundamental components:

- State (s): A representation of the current situation of the environment that the agent perceives, which is the *node embedding* in the work.
- Action (a): A choice made by the agent that influences the environment, which is the level adjustment of nodes in this work. As each node resides at a specific level, the agent adjusts a target node from its current level to a new level. The action space is limited to three elements: $\{0, 1, 2\}$. Action 0 denotes that the target node remains at its current level. Action 1 denotes a transition to the next level (current level plus one) and Action 2 denotes a transition to the level two steps away (current level plus two). We intentionally limit the action space to avoid higher-level adjustments, as they tend to serialize CUDA Graph execution and degrade performance.
- State transition: The change in the environment's state that occurs as a result of the agent taking an action, including the changes of level, fanin and fanout.

- Reward (r): A signal that CUDA Graph runtime provides to the RL agent after the agent takes an action in a particular state. In this work, we focus on minimizing the execution time (ET) of a CUDA graph. Therefore, we design the reward function to reflect this objective:

$$\text{reward} = -(\texttt{ET}^{\text{after}} - \texttt{ET}^{\text{before}}), \tag{6.1}$$

  where $\texttt{ET}^{\text{before}}$ and $\texttt{ET}^{\text{after}}$ denote the normalized execution times before and after the action, respectively. Note that minimizing ET is equivalent to maximizing the reward.

To solve the RL problem, we leverage the Deep Q-learning (DQN) algorithm [131]. We do not choose traditional methods [143], such as value iterations, because they are computationally intractable in high-dimensional environments. DQN solves the problem by employing deep neural networks to approximate the `Q-function`, which represents the expected cumulative reward for taking an action (a) in a given state (s), denoted as $Q(s, a)$. The Q-function effectively maps state-action pairs to their corresponding Q-values, enabling the agent to make informed decisions. Next, we discuss how to adapt the DQN algorithm to suggest an action and obtain a CUDA graph with the components, *policy network*, *target network*, *replay buffer*, and *action to graph*, as shown in Figure 6.4.

Policy and Target Network

The *policy network*, a fully connected neural network, maps a state (represented by *node embedding*) to an action, which corresponds to node-level adjustments. The *policy network* has a layered architecture comprising 3072 neurons in the input layer, 256 neurons in the second layer, 64 neurons in the third layer, and 3 neurons in the output layer. The network receives the *node embedding*, forwarded by the GNN module, as its input state. Then the network processes this state information through the two

hidden layers. Finally, the network outputs a set of three Q-values. Each Q-value corresponds to the expected reward associated with a specific action that adjusts the level of a target node in the graph. The first Q-value corresponds to action 0, which does not adjust the target node's level. The second Q-value corresponds to action 1, which increments the current level by 1, and the third Q-value corresponds to action 2, which increments the current level by 2. To balance exploration and exploitation during the learning process, we employ the ϵ-greedy strategy [143] in determining the action. This strategy allows the agent to explore new actions with probability ϵ and exploit the learned policy with probability $1 - \epsilon$.

In addition to the *policy network*, we incorporate a separate network, the *target network*, which plays a crucial role in stabilizing the training process. The *target network* maintains an identical architecture to the *policy network*, but its weights are updated less frequently. This separation is essential for mitigating the overestimation bias [103], a common issue in DQN that can lead to unstable training. We periodically copy the weights from the *policy network* to the *target network* using a soft update mechanism [110], expressed in the equation:

$$\theta' = \tau\theta + (1 - \tau)\theta', \tag{6.2}$$

where θ' denotes the parameters of the *target network*, θ denotes the parameters of the *policy network*, and τ denotes the update rate of the *target network*. This soft update approach allows for a gradual and controlled transfer of learned information from the *policy network* to the *target network*. We use the *target network* to evaluate the expected Q-value of the action suggested by the *policy network*. This evaluation provides an unbiased estimate of the Q-value for the next state, crucial for accurate learning. The expected Q-value is calculated using the equation:

$$Q^{expected} = r + \gamma \max_{a} Q(s', a), \tag{6.3}$$

where γ represents the discount factor, reflecting the importance of future rewards, and s' denotes the state after the action is executed, reflecting the updated environment.

During the training phase, we employ the Huber loss function [2] to address the challenge of outliers in noisy Q-value estimates, which are common in RL. Our objective is to minimize the Huber loss. The mathematical definition of the Huber loss is shown below:

$$\mathcal{L} = \begin{cases} \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (\frac{1}{2}\delta^2) & \text{if } |\delta| \leqslant 1, \\ \frac{1}{|B|} \sum_{(s,a,r,s') \in B} (|\delta| - \frac{1}{2}) & \text{otherwise.} \end{cases}$$

where

$$\delta = Q(s, a) - Q^{expected}. \tag{6.4}$$

Once we calculate the loss, we leverage backpropagation to propagate the error signal through the entire model. To improve convergence, we stop propagating the error signal to the GNN model after 10 epochs in our evaluation. Finally, we utilize the Adam optimizer [95].

Action to Graph

The *action-to-graph* component serves as the link between the *policy network*'s output and the physical modification of the CUDA graph. It performs a transformation that converts the action proposed by the *policy network* into a concrete change within the graph, that is, the appending of a new auxiliary edge. Each action, taking values of 0, 1, or 2, corresponds to a level adjustment for a given target node. These adjustments are : (1) maintaining the current level (action 0), (2) incrementing the level by one (action 1), or (3) incrementing the level by two (action 2). When action 0 is suggested by the *policy network*, it implies that no change is needed for the target node's level. However, when actions 1 or 2 are suggested, they necessitate the addition of an auxiliary edge to facilitate the target node's

transition to the new level.

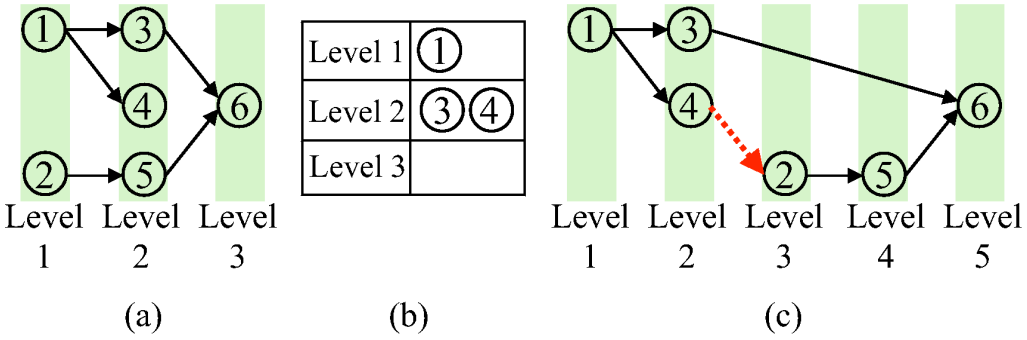


Figure 6.6: Illustration of using bucket list to convert an action to an edge. (a) A graph with 5 edges and 6 nodes within 3 layers. (b) A bucket list for node 2. (c) A new graph after moving node 2 from level 1 to level 3. The red dash edge is the auxiliary edge.

To add a new auxiliary edge for a target node, a straightforward approach is to connect the target node at its new level to a randomly selected node at the immediately preceding level. For example, as illustrated in Figure 6.6(a), an edge from node 1 to node 2 facilitates node 2's transition from level 1 to level 2. However, this design can introduce cycles within the graph, particularly when a successor node is involved in the random selection process. Consider Figure 6.6(a), if node 5, which is a successor of node 2 and randomly selected from level 2, is connected to node 2 to transition node 2 to a new level (level 3), a cycle is immediately introduced. Therefore, we require a mechanism to explicitly exclude successors of a target node from the random selection process, ensuring that the graph remains acyclic.

To prevent cycles while introducing an auxiliary edge for a target node, we construct a bucket list for each node. Each bucket list is an associative container that stores key-value pairs. The key represents a level, and the value is a set of nodes located at that level. Crucially, this bucket list excludes all the successor nodes of the target node. For instance, as

illustrated in Figure 6.6(b), the bucket list for node 2 in Figure 6.6(a) contains two entries: {level 1: node 1} and {level 2: {nodes 3, 4}}. Notably, this bucket list intentionally omits node 2's successors, nodes 5 and 6. This design enables a straightforward selection of a non-successor node, such as node 4 at level 2, to add a new edge to node 2. This edge addition transitions node 2 from level 1 to level 3, as shown in Figure 6.6(c). With this design, we can efficiently convert an action suggested by the *policy network* into a new edge without introducing any cycles into the resulting CUDA graph.

Replay Buffer

Replay buffer serves as a memory mechanism in the RL module for a stable and effective learning. We store a collection of transitions which consists of the current state, the suggested action, the received reward, and the next state. By storing these transitions, the agent is able to learn from past interactions. During training, the agent randomly samples batches from the replay buffer, which breaks the correlation between consecutive transitions and enables more efficient learning.

6.5 Experimental Results

We implemented our framework using C++17 and CUDA 12.2 and compiled the program with the nvcc compiler on a host compiler of g++11.4 with -std=c++17 and -O3 enabled. We used Pytorch to train and test the model. We ran all the experiments on a Ubuntu Linux 22.04 machine with 20 Intel i5-13500 CPU cores at 4.8 GHz and 128 GB RAM, and an Nvidia RTX A4000 GPU.

Benchmarks and Baseline

We evaluated the runtime performance on 12 circuit graphs derived from the [23]. Each circuit graph represents an SSTA propagation graph, as detailed in Section 6.3. The statistical timing quantities of varying batch size B for each pin were sampled from a normal distribution. Table 6.1 presents the statistics of the 12 circuit graphs used in the evaluation, with the default batch size B set to 64, meaning that we calculated a pin's 64 data points concurrently per iteration. To finish 65536 data points, each benchmark requires 1024 iterations. We used the application's original CUDA graph as the baseline, which also represents the GPU-accelerated solution provided by [23, 116]. We did not include other baselines, as to the best of our knowledge, this work is the first to address CUDA Graph scheduling optimization for SSTA propagation.

Training

We trained our model on a synthetic circuit derived from [23]. To ensure generalization, the training and testing phases used different circuit instances. We used the following hyper-parameters: target network update rate of 0.005, discount factor of 0.99, training iterations of 100, initial ϵ of 1.0, final ϵ of 0.05, ϵ decay rate of 0.997, learning mini batch size of 16, and Adam optimizer learning rate of 0.0001. Figure 6.7 shows the training error and the rewards achieved by the RL policy. The left plot demonstrates a rapid decay in training loss, indicating effective policy learning. The right plot shows the RL policy converging to rewards between 5 and 7.

Table 6.1: Runtime comparison and circuit statistics of the benchmarks. The batch size in this table is 64. T^B and T^O denote the runtime of the baseline and ours, respectively. Δt denotes the runtime difference between the baseline and ours. *Impr.* denotes the runtime improvement of our CUDA graph over the baseline. $\|E\|'$ denotes the number of edges in the new CUDA graph generated by our framework.

Benchmark	$\|V\|$	$\|E\|$	T^B (s)	T^O (s)	Δt (s)	Impr.	$\|E\|'$
c17	75	78	332.81	306.19	26.61	8%	89
s27	81	87	332.84	292.89	39.95	12%	95
c17_2	150	156	333.47	314.8	18.67	5.6%	162
s27_2	243	261	335.8	308.6	27.2	8.1%	271
c432	483	619	360.51	330.23	30.28	8.4%	633
c499	604	742	342.54	320.62	21.92	6.4%	760
s344	526	625	338	317.72	20.28	6%	640
s349	550	649	340.63	325.64	14.99	4.4%	664
c2670	1365	1665	437.76	422	15.76	3.6%	1685
s1196	1854	2344	494.74	474.95	19.79	4%	2366
s1494	2292	2925	539.96	524.84	15.12	2.8%	2950
usb_phy	2447	2999	540.8	524.58	16.22	3%	3036
Average					22.23 s	6%	

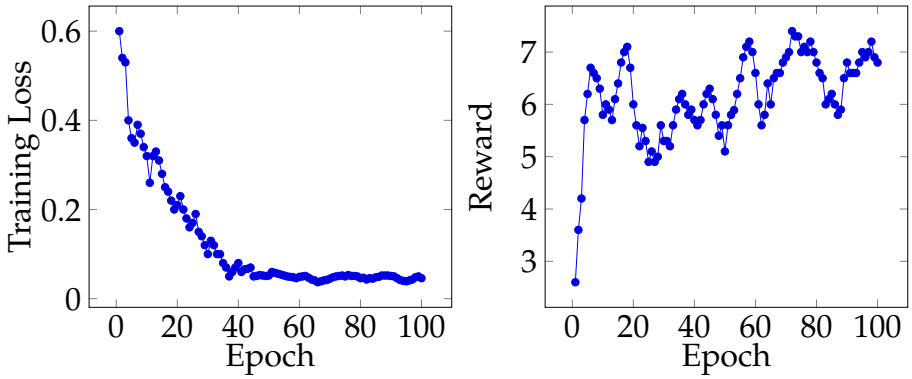


Figure 6.7: Training loss and rewards achieved by the RL policy.

Overall Performance Comparison

Table 6.1 presents the runtime performance of the baseline and our solution. Our solution consistently outperforms the baseline across all benchmarks with batch size 64. For instance, our solution achieves a 12% runtime improvement over the baseline for *s27*. Similarly, for *c2670*, we observe a 3.6%. The reason is the following. The baseline exhibits excessive task parallelism, which, while seemingly beneficial, can introduce resource contention that ultimately degrades runtime performance. Our framework appends edges within the CUDA graph to slightly reduce task parallelism in favor of improved resource utilization. This controlled parallelism aims to achieve a balance between task execution and scheduling management, leading to overall improved performance compared to the baseline. Note that the runtime improvement may appear small for some benchmarks (e.g., 3% on *usb_phy*), but in practice, the gain can accumulate to hours as SSTA applications must run many iterations across different input values and configurations [23]. More importantly, this runtime gain comes at little cost to developers, as our framework requires no changes to application-level algorithms but simply restructures the CUDA graph to guide the runtime toward better scheduling performance.

These results highlight that batch size is a critical factor in maximizing the benefits of our optimized CUDA graph. We observe that the runtime improvement becomes smaller at a larger batch size. For example, in Figure 6.8, when running *s27_2*, the improvement decreases from 8.1% with batch size 64 to 2.6% with batch size 4096. We attribute this result to the increased computational load associated with larger batch sizes. Apparently, as kernel computation begins to dominate overall runtime, the relative benefit of improved scheduling diminishes. However, we should also notice that larger batch sizes incur higher GPU memory usage, which in turn limits the maximum circuit size that can be processed on a single GPU.

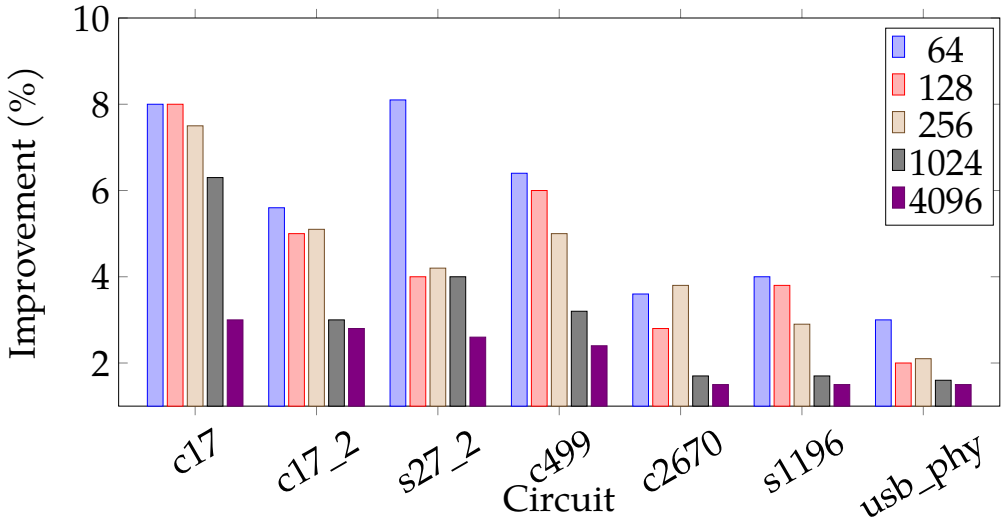
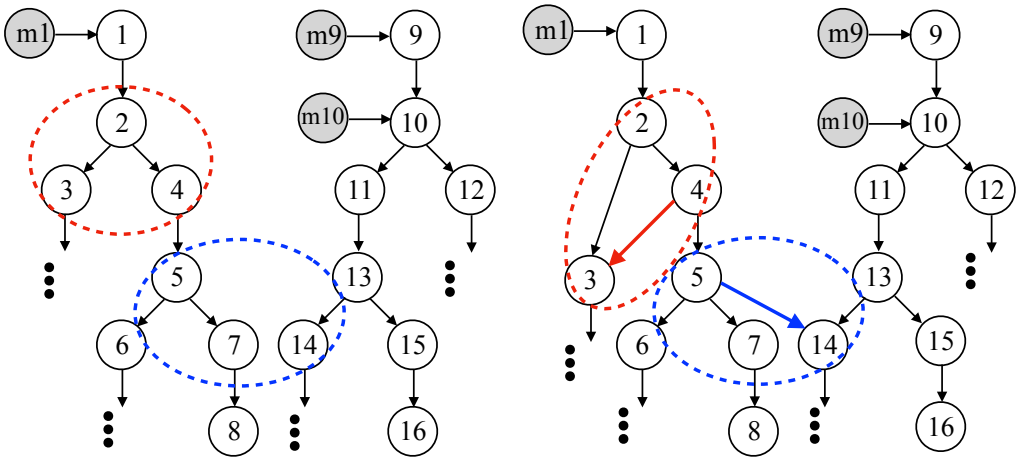


Figure 6.8: Plot of runtime improvement of our framework over the baseline on seven benchmarks with five different batch sizes.



(a) Original CUDA graph (b) RL-optimized CUDA graph

Figure 6.9: Partial *c17* CUDA graph visualization: (a) application's original, (b) our optimized CUDA graph. Blue/red dashed cycles indicate changes due to blue/red edge additions. White circles denote kernels and gray denote memory copies.

Our RL algorithm always brings positive benefits, as it does not modify any application-level algorithms, but restructures the given CUDA graph to guide the CUDA runtime toward better scheduling performance. These edges facilitate improved scheduling by adapting the application's needs (i.e., application-level information) to the changing environment on GPU. However, excessive edge additions can potentially serialize CUDA Graph execution, leading to performance degradation. Our framework strikes a balance by minimizing the number of auxiliary edges added. In Table 6.1, our framework consistently introduces a small number of auxiliary edges. For instance, in the *usb_phy* circuit with batch size 64, our CUDA graph includes only 37 additional edges. Figure 6.9 visualizes the application's original CUDA graph and the optimized CUDA graph from our RL algorithm. Our framework added two edges to reduce scheduling overhead caused by excessive task parallelism in the application's CUDA graphs.

Quality of Result

In addition to the runtime comparison, we evaluate the *Quality of Result* (QoR) by judging how close each CUDA graph is to the potentially best result. Specifically, we generated 10,000 distinct CUDA graphs per benchmark by randomly appending auxiliary edges to the application's original CUDA graph. Then, from these 10,000 sampled graphs, we extracted the minimum and maximum runtimes and normalized them to $[0, 1]$ to establish the potentially best and worst performance bounds. We can express QoR as follows,

$$\text{QoR} = 1 - \left(\frac{\text{T} - \text{T}^{min}}{\text{T}^{max} - \text{T}^{min}}\right), \tag{6.5}$$

where T^{min} and T^{max} denote the sampled minimum and maximum runtime, respectively. In Figure 6.10, our optimized graph consistently exhibits a higher QoR compared to the baseline. For example, on the *c2670* benchmark with batch size 128, our solution achieves a normalized QoR of 0.96,

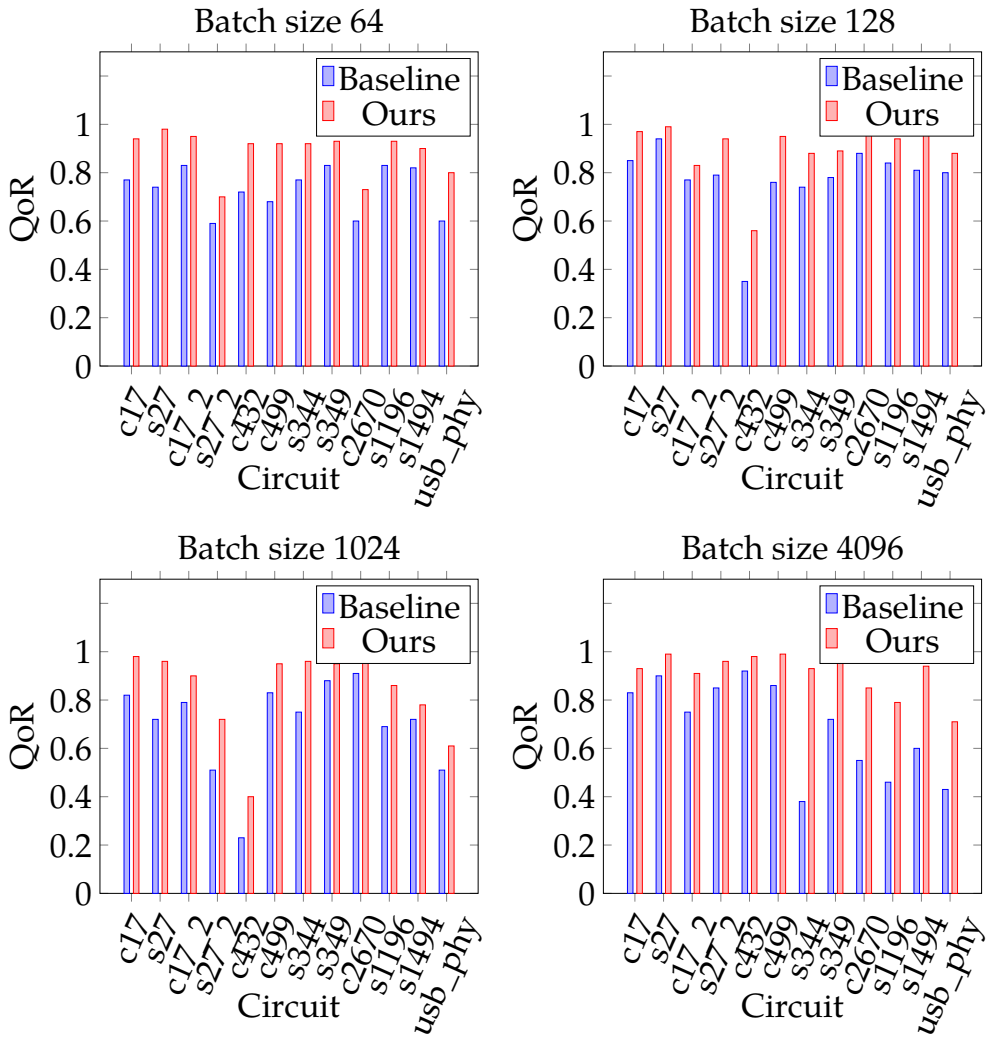


Figure 6.10: QoR between the application-given and our optimized CUDA graphs. A value closer to 1 indicates better QoR.

compared to 0.9 for the baseline. Furthermore, the majority of our optimized CUDA graphs have QoR over 0.8, indicating a close proximity to the best sample, whereas the baseline typically falls between 0.6 and 0.8. This result highlights the effectiveness of our approach in consistently generating higher-quality CUDA graphs.

Comparison with Random Edge Insertion

To further validate the effectiveness of our framework, we implemented a method that randomly inserts the same quantity of auxiliary edges as ours into the original CUDA graph. The goal is to show that blindly inserting edges without a learning-guided process yields limited or even negative performance improvement. As shown in Figure 6.11, we randomly inserted 11 and 37 edges into the application's *c17* and *usb_phy* CUDA graphs, ran this experiment 2000 times, and recorded the runtime for each run. We notice that only a small fraction of CUDA graphs can achieve performance comparable to our RL-based solution. For example, on the *usb_phy* benchmark, fewer than 5% of CUDA graphs match the performance of our RL-optimized CUDA graph. We attribute this finding to the fact that our framework learns to identify beneficial edge insertions by interacting with the CUDA Graph runtime and adapting to its hidden scheduling mechanisms. In contrast, the non-learning approach inserts edges randomly without leveraging system feedback. While random insertion may occasionally yield good performance after many attempts, most resulting CUDA graphs fail to deliver decent improvements.

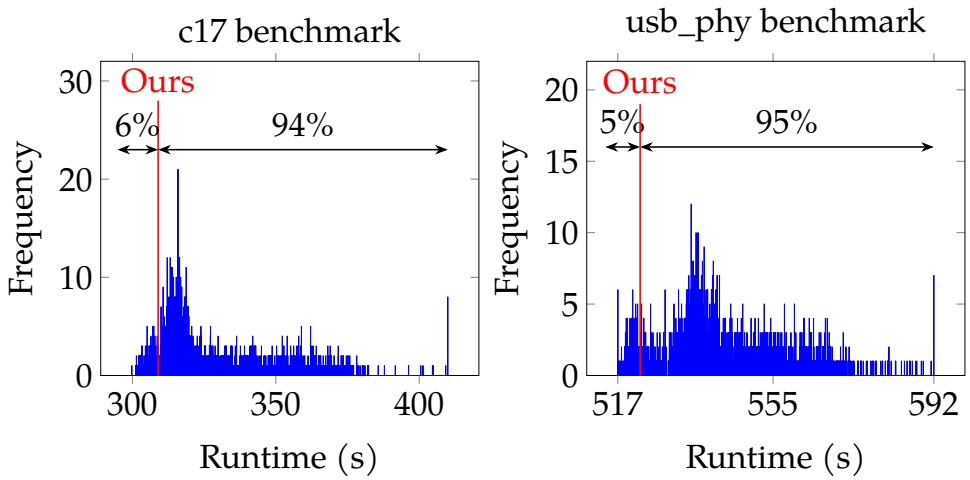


Figure 6.11: Histogram of the random edge insertion approach on *c17* and *usb_phy*. 2000 different CUDA graphs were generated by randomly appending the same amounts of auxiliary edges as our solution to the application's original CUDA graph. Only a small portion ($\sim$ 5%) of the 2000 CUDA graphs perform better runtime performance as our optimized CUDA graph (indicated by the vertical red line).

6.6 Conclusion

We have introduced a reinforcement learning-based framework to optimize CUDA Graph scheduling on SSTA propagation workloads. We have formulated the CUDA Graph scheduling as a node-level adjustment problem. To solve the problem formulation, we have leveraged a graph neural network to encode structural information and employed a deep Q learning algorithm to interact with CUDA Graph runtime and generate an optimized CUDA graph. Compared to the baseline, we achieved up to 12% runtime improvement. Notably, this performance improvement is almost free, as our framework requires no changes to application-level algorithms, but simply restructures the application's given CUDA graph to guide the CUDA Graph runtime toward better scheduling performance.

In the future, we plan to extend our algorithms to other applications powered by task graph parallelism [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 38, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 78, 83, 88, 89, 90, 91, 100, 101, 102, 105, 106, 107, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 132, 133, 145, 146, 152, 153, 154, 155].

7 CONCLUSION

In conclusion, this thesis highlights several significant advancements in the discipline of task-parallel programming and task graph scheduling with efficient algorithms and frameworks to enhance runtime performance and efficiency. The main contributions are summarized as follows:

1. **An Efficient Task-parallel Pipeline Programming Framework**. We have introduce a new task-parallel pipeline programming framework called Pipeflow to separate data abstractions and task scheduling, enabling a more efficient implementation of task- parallel pipeline algorithms than existing frameworks. We have demonstrated Pipeflow outperforms existing solutions by up to 111.33% faster to avoid redundant data abstractions in task-parallel applications.

2. **A Task-parallel Pipeline Programming Model with Token Dependency**. We have extended our Pipeflow framework to support both forward and backward token dependencies. We have demonstrated our token-aware Pipeflow is 8.6% faster than existing solutions and is more efficient in programming video encoding applications.

3. **Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms**. We have introduced a new programming model, call AsyncTask, that supports the dynamic building of a computational task graph. We have demonstrated AsyncTask outperforms a mainstream library by up to 3.19$\times$ faster on real-wold applications through a straightforward programming model and an efficient dynamic task graph scheduling.

4. **Resource-efficient Task Scheduling**. We have introduced a novel reinforcement learning-based scheduling algorithm that learns to adapt the performance optimization to a given runtime situation. We

have demonstrated our scheduling algorithm achieves a promising runtime performance on all evaluations while using only 20% of computing units. This highlights the effectiveness of our reinforcement learning model in scheduling tasks across resource-conscious subset of computing units.

5. **Topological Order for Dynamic Task Graph Scheduling**. We have introduced a novel method that leverages reinforcement learning to generate topological orders for dynamic task graph scheduling systems. We have demonstrated our reinforcement learning-generated order achieves a speedup of up to 1.52× over the baseline on real-world applications, showing the advantage of our method in efficiently adapting the task scheduling to a work-stealing runtime.

6. **CUDA Graph Scheduling Optimization**. We have introduced a reinforcement learning-based framework to optimize CUDA Graph scheduling via graph restructure. We have demonstrated our optimized CUDA graph can achieve up to a 12% runtime improvement on statistical static timing analysis (SSTA) propagation workloads. This runtime gain comes at little cost to developers, as our framework requires no changes to application-level algorithms but simply restructures the CUDA graph to guide the runtime toward better scheduling performance.

Together, these contributions advance the state of the art in task-parallel programming and intelligent task graph scheduling, offering practical frameworks and algorithms applicable to a wide range of high-performance computing scenarios. The research centers on efficient task graph scheduling and algorithmic design to enhance runtime performance.

Future works should cover the following directions: (1) selecting the pipeline architecture for the Pipeflow library using machine learning techniques; (2) improving the data locality for the Pipeflow library; (3) apply-

ing AsyncTask to more applications, such as distributed computing, macro modeling, and path-based analysis; (4) extending our resource-efficient reinforcement learning-based framework to a distributed environment and considering GPU task graphs into our model; (5) applying our reinforcement learning-based framework to more parallel parallel graph algorithms, such as quantum circuit simulator, with better topological orders and thus better runtime performance; (6) exploring the scalability of our CUDA Graph optimization framework for larger and more complex graphs, and extending the framework to more applications powered by task graph parallelism. By following these directions, the frameworks and algorithms introduced in this thesis can further contribute to ongoing advancements in task-parallel programming and scheduling and heterogeneous programming.

BIBLIOGRAPHY

[1] C++ Condition Variable. https://en.cppreference.com/w/cpp/thread/condition_variable

[2] Huber Loss. https://en.wikipedia.org/wiki/Huber_loss

[3] Intel oneTBB. https://github.com/oneapi-src/oneTBB

[4] Kahn's Algorithm for Topological Sorting. https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/

[5] OpenMP. https://www.openmp.org/

[6] Opentimer. https://github.com/OpenTimer/OpenTimer

[7] Rectifier (Neural Networks). https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

[8] Taskflow. https://taskflow.github.io

[9] TAU 2018 Contest. https://sites.google.com/view/taucontest2018/home

[10] Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: High-Level and Efficient Streaming on Multicore. In: Programming Multicore and Manycore Computing Systems. pp. 261–280 (2017)

[11] Basaklar, T., Goksoy, A.A., Krishnakumar, A., Gumussoy, S., Ogras, U.Y.: DTRL: Decision Tree-based Multi-Objective Reinforcement Learning for Runtime Task Scheduling in Domain-Specific System-on-Chips. In: ACM Transactions on Embedded Computing Systems. pp. 1–22 (2023)

[12] Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implication. In: International conference on Parallel Architectures and Compilation Technique (PACT). pp. 72–81 (2008)

[13] Bienia, C., Li, K.: Scaling of the PARSEC Benchmark Inputs. In: International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 561–562 (2010)

[14] Blaauw, D., Chopra, K., Srivastava, A., Scheffer, L.: Statistical Timing Analysis: From Basic Principles to State of the Art. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 589–607 (2008)

[15] Blanco, F.G., Russo, E., Palesi, M., Patti, D., Ascia, G., Catania, V.: Deep Reinforcement Learning based Online Scheduling Policy for Deep Neural Network Multi-Tenant Multi-Accelerator Systems. In: Design Automation Conference (DAC). pp. 1–6 (2024)

[16] Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. In: ACM SIGPLAN Notices. pp. 207–216 (1995)

[17] 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)

[18] Bosilca, G., Harrison, R., Herault, T., Javanmard, M., Nookala, P., Valeev, E.: The Template Task Graph (TTG) - An Emerging Practical Dataflow Programming Paradigm for Scientific Simulation at Extreme Scale. In: ACM ESPM2. pp. 1–7 (2020)

[19] Chang, C., Chiu, C.H., Zhang, B., Huang, T.W.: Incremental Critical Path Generation for Dynamic Graphs. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 771–774 (2024)

[20] Chang, C., Huang, T.W., Lin, D.L., Guo, G., Lin, S.: Ink: Efficient Incremental k-Critical Path Generation. In: ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2024)

[21] Chang, C., Zhang, B., Chiu, C.H., Lin, D.L., Chung, Y.H., Lee, W.L., Guo, Z., Lin, Y., Huang, T.W.: PathGen: An Efficient Parallel Critical Path Generation Algorithm. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 416–424 (2025)

[22] Chang, C.C., Huang, T.W.: uSAP: An Ultra-Fast Stochastic Graph Partitioner. In: IEEE High-performance and Extreme Computing Conference (HPEC). pp. 1–7 (2023)

[23] Chang, C.C., Huang, T.W.: Statistical Timing Graph Scheduling Algorithm for GPU Computation. In: ACM/IEEE Design Automation Conference (DAC) (2025)

[24] Chang, C.C., Zhang, B., Huang, T.W.: GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In: ACM International Conference on Parallel Processing (ICPP). pp. 565–575 (2024)

[25] Chiu, C.H., Huang, T.W.: Composing Pipeline Parallelism using Control Taskflow Graph. In: ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC). pp. 283–284 (2022)

[26] Chiu, C.H., Huang, T.W.: Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In: ACM/IEEE Design Automation Conference (DAC). pp. 1388–1389 (2022)

[27] Chiu, C.H., Huang, T.W.: An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 766–770 (2024)

[28] Chiu, C.H., Lin, D.L., Huang, T.W.: 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). pp. 468–479 (2021)

[29] Chiu, C.H., Lin, D.L., Huang, T.W.: Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 1–8 (2023)

[30] Chiu, C.H., Morchdi, C., Zhou, Y., Zhang, B., Chang, C., Huang, T.W.: Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In: IEEE High-performance and Extreme Computing Conference (HPEC) (2024)

[31] Chiu, C.H., Xiong, Z., Guo, Z., Huang, T.W., Lin, Y.: An Efficient Task-Parallel Pipeline Programming Framework. In: International Conference on High Performance Computing in Asia-Pacific Region (HPC-Asia). pp. 95–106 (2024)

[32] Chung, Y.H., Jiang, S., Lee, W.L., Zhang, Y., Ren, H., Ho, T.Y., Huang, T.W.: SimPart: A Simple Yet Effective Replication-aided Partitioning Algorithm for Logic Simulation on GPU. In: International European Conference on Parallel and Distributed Computing (Euro-Par) (2025)

[33] Ding, X., Wang, K., Gibbons, P., Zhang, X.: BWS: Balanced Work Stealing for Time-sharing Multicores. In: ACM European Conference on Computer Systems (EuroSys). pp. 365–378 (2012)

[34] Dzaka, E., Lin, D.L., Huang, T.W.: Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In: IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw) (2023)

[35] Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. In: Journal of Parallel and Distributed Computing. pp. 3202–3216 (2014)

[36] Fan, Y., Lan, Z., Childers, T., Rich, P., Allcock, W., Papka, M.E.: Deep Reinforcement Agent for Scheduling in HPC. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 807–816 (2021)

[37] Forzan, C., Pandini, D.: Statistical Static Timing Analysis: A survey. In: Integration. pp. 409–435 (2009)

[38] Gener, S., Hassan, S., Chang, L., Chakrabarti, C., Huang, T.W., Ogras, U., , Akoglu, A.: A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In: ACM International Workshop on Extreme Heterogeneity Solutions (ExHET). pp. 1–9 (2025)

[39] Goksoy, A.A., Kanani, A., Chatterjee, S., Ogras, U.: Runtime Monitoring of ML-based Scheduling Algorithms Toward Robust Domain-specific SoCs. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 4202–4213 (2024)

[40] Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.: SPar: A DSL for High-Level and Productive Stream Parallelism. In: Parallel Processing Letters (2017)

[41] Griebler, D., Hoffmann, R., Danelutto, M., Fernandes, L.: High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. In: The Journal of Parallel Programming. pp. 253–271 (2019)

[42] Gulati, K., Khatri, S.: Accelerating Statistical Static Timing Analysis using Graphics Processing Units. In: Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 260–265 (2009)

[43] Guo, G., Huang, T.W., Lin, C.X., Wong, M.: An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In: ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2020)

[44] Guo, G., Huang, T.W., Lin, Y., Guo, Z., Yellapragada, S., Wong, M.: A GPU-Accelerated Framework for Path-Based Timing Analysis. In: IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD). pp. 4219–4232 (2023)

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

[46] Guo, G., Huang, T.W., Lin, Y., Wong, M.: GPU-accelerated Path-based Timing Analysis. In: IEEE/ACM Design Automation Conference (DAC). pp. 721–726 (2021)

[47] Guo, G., Huang, T.W., Wong, M.D.F.: Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In: IEEE/ACM Design, Automation and Test in Europe Conference (DATE) (2023)

[48] Guo, Z., Huang, T.W., Lin, Y.: 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). pp. 3466–3478 (2020)

[49] Guo, Z., Huang, T.W., Lin, Y.: GPU-accelerated Static Timing Analysis. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2020)

[50] Guo, Z., Huang, T.W., Lin, Y.: HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2021)

[51] Guo, Z., Huang, T.W., Lin, Y.: Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. In: IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD). pp. 4973–4984 (2023)

[52] 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)

[53] Guo, Z., Zhang, Z., Li, W., Huang, T.W., Shi, X., Du, Y., Lin, Y., Wang, R., Huang, R.: HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 1–9 (2024)

[54] Hamilton, W.L., Ying, R., Leskovec, J.: Inductive Representation Leanring on Large Graphs. In: Conference on Neural Information Processing Systems (NIPS) (2017)

[55] Haykin, S.: Neural Networks: A Comprehensive Foundation. In: Prentice Hall PTR (1994)

[56] Hoffman, R., Loff, J., Griebler, D., Fernandes, L.: OpenMP as Runtime for Providing High-Level Stream Parallelism on Multi-Cores. In: The Journal of Supercomputing. pp. 7655–7676 (2022)

[57] Hoffmann, R., Korch, M., Rauber, T.: Performance Evaluation of Task Pools Based on Hardware Synchronization. In: ACM Supercomputing (2004)

[58] Hoque, R., Herault, T., Bosilca, G., Dongarra, J.J.: Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In: Workshop on Latest Advances in Scalable Algorithm for Large-Scale Systems (ScalA). pp. 1–8 (2017)

[59] Huang, T.W.: A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 1–2 (2020)

[60] Huang, T.W.: TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In: IEEE International Workshop on Programming and Performance Visualization Tools (ProTools). pp. 1–6 (2021)

[61] Huang, T.W.: Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In: IEEE High-Performance Extreme Computing Conference (HPEC) (2022)

[62] Huang, T.W.: qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 746–756 (2023)

[63] Huang, T.W., Guo, G., Lin, C.X., Wong, M.D.F.: OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) (2021)

[64] Huang, T.W., Hwang, L.: Task-parallel Programming with Constrained Parallelism. In: IEEE High-Performance Extreme Computing Conference (HPEC) (2022)

[65] Huang, T.W., Lin, C.X., , Wong, M.: Distributed Timing Analysis at Scale. In: ACM/IEEE Design Automation Conference (DAC). pp. 1–2 (2019)

[66] Huang, T.W., Lin, C.X., Guo, G., Wong, M.: A General-purpose Distributed Programming System using Data-parallel Streams. In: ACM Multimedia Conference (MM). pp. 1360–1363 (2018)

[67] Huang, T.W., Lin, C.X., Guo, G., Wong, M.: Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2019)

[68] Huang, T.W., Lin, C.X., Guo, G., Wong, M.: Essential Building Blocks for Creating an Open-source EDA Project. In: ACM/IEEE Design Automation Conference (DAC). pp. 1–4 (2019)

[69] Huang, T.W., Lin, C.X., Wong, M.: DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 757–764 (2017)

[70] Huang, T.W., Lin, C.X., Wong, M.: DtCraft: A High-performance Distributed Execution Engine at Scale. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 1070–1083 (2019)

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

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

[73] Huang, T.W., Lin, D.L., Lin, Y., Lin, C.X.: Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) (2022)

[74] Huang, T.W., Lin, Y.: Concurrent CPU-GPU Task Programming using Modern C++. In: IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS). pp. 588–597 (2022)

[75] 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. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) (2021)

[76] Huang, T.W., Wong, M.: OpenTimer: A High-Performance Timing Analysis Tool. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 895–902 (2015)

[77] Huang, T.W., Wong, M.: UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 1862–1875 (2016)

[78] Huang, T.W., Wong, M., Sinha, D., Kalafala, K., Venkateswaran, N.: A Distributed Timing Analysis Framework for Large Designs. In: IEEE/ACM Design Automation Conference (DAC). pp. 1–6 (2016)

[79] Huang, T.W., Wong, M.D.: Accelerated Path-Based Timing Analysis with MapReduce. In: International Symposium on Physical Design (ISPD). pp. 103–110 (2015)

[80] Huang, T.W., Wong, M.D.: On Fast Timing Closure: Speeding up Incremental Path-based Timing Analysis with Mapreduce. In: International Workshop on System Level Interconnect Prediction (SLIP) (2015)

[81] Huang, T.W., Wu, P.C., Wong, M.: Fast Path-Based Timing Analysis for CPPR. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 596–599 (2014)

[82] Huang, T.W., Wu, P.C., Wong, M.D.F.: UI-Timer: An Ultra-fast Clock Network Pessimism Removal Algorithm. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 758–765 (2014)

[83] 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). pp. 51–59 (2024)

[84] 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)

[85] Jess, J., Kalafala, K., Naidu, S., Otten, R., Visweswariah, C.: Statistical Timing for Parametric Yield Prediction of Digital Integrated Circuits. In: ACM/IEEE Design Automation Conference (DAC). pp. 932–937 (2003)

[86] Jia, Z., Lin, S., Qi, C.R., Aiken, A.: Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks. In: International Conference on Machine Learning. pp. 2274–2283 (2018)

[87] 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)

[88] Jiang, S., Huang, T.W., Ho, T.Y.: GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In: IEEE High-performance and Extreme Computing Conference (HPEC) (2023)

[89] Jiang, S., Huang, T.W., Ho, T.Y.: SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In: ACM International Conference on Parallel Processing (ICPP). pp. 51–61 (2023)

[90] Jiang, S., Chung, Y.H., Chang, C.C., Ho, T.Y., Huang, T.W.: BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 79–94 (2025)

[91] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei: FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In: ACM International Conference on Parallel Processing (ICPP). pp. 388–399 (2024)

[92] Kahng, A.: Reducing Time and Effort in IC Implementation: A Roadmap of Challenges and Solutions. In: ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2018)

[93] Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: A Task Based Programming Model in a Global Address Space. In: International Conference on Partitioned Global Address Space Programming Mod (PGAS). pp. 1–11 (2014)

[94] Kamruzzaman, M., Swanson, S., Tullsen, D.: Load-Balanced Pipeline Parallelism. In: International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2013)

[95] Kingma, D., Ba, J.: Adam: A Method for Stochastic Optimization. In: International Conference on Learning Representations (ICLR) (2014)

[96] Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. In: International Conference on Learning Representations (ICLR) (2015)

[97] Kipf, T., Welling, M.: Semi-Supervised Classification with Graph Convolutional Networks. In: International Conference on Learning Representations (ICLR) (2017)

[98] Konda, V.R., Borkar, V.S.: Actor-Critic–Type Learning Algorithms for Markov Decision Processes. In: Journal on Control and Optimization. pp. 94–123 (1999)

[99] Krishnakumar, A., Arda, S.E., Goksoy, A.A., Mandal, S.K., Ogras, U.Y., Sartor, A.L.: Runtime Task Scheduling Using Imitation Learning for Heterogeneous Many-core Systems. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 4064–4077 (2020)

[100] Lai, K.M., Huang, T.W., Ho, T.Y.: A General Cache Framework for Efficient Generation of Timing Critical Paths. In: ACM/IEEE Design Automation Conference (DAC) (2019)

[101] Lai, K.M., Huang, T.W., Lee, P.Y., Ho, T.Y.: ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 278–283 (2021)

[102] Lai, T.Y., Huang, T.W., , Wong, M.: Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 1–6 (2017)

[103] Lan, Q., Pan, Y., andMartha White, A.F.: Maxmin Q-learning: Controlling the Estimation Bias of Q-learning. In: International Conference on Learning Representations (ICLR) (2020)

[104] Lee, I.T.A., Leiserson, C., Schardl, T., Zhang, Z., Sukha, J.: On-the-Fly Pipeline Parallelism. In: ACM Transactions on Parallel Computing (TOPC) (2015)

[105] Lee, W.L., Jiang, S., Lin, D.L., Chang, C., Zhang, B., Chung, Y.H., Schlichtmann, U., Ho, T.Y., , Huang, T.W.: iG-kway: Incremental k-way Graph Partitioning on GPU. In: ACM/IEEE Design Automation Conference (DAC) (2025)

[106] Lee, W.L., Lin, D.L., Chiu, C.H., Schlichtmann, U., Huang, T.W.: HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 1031–1040 (2025)

[107] 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). pp. 1–6 (2024)

[108] Leijen, D., Schulte, W., Burckhardt, S.: The Design of a Task Parallel Library. In: ACM SIGPLAN Notices. pp. 227–242 (2009)

[109] Leiserson, C.: The Cilk++ Concurrency Platform. In: The Journal of Supercomputing. pp. 244–257 (2010)

[110] Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous Control with Deep Reinforcement Learning. In: International Conference on Learning (ICLR) (2016)

[111] Lin, C.X., Huang, T.W., Guo, G., Wong, M.: A Modern C++ Parallel Task Programming Library. In: ACM Multimedia Conference (MM). pp. 2284–2287 (2019)

[112] Lin, C.X., Huang, T.W., Guo, G., Wong, M.: An Efficient and Composable Parallel Task Programming Library. In: IEEE High-performance and Extreme Computing Conference (HPEC) (2019)

[113] Lin, C.X., Huang, T.W., Wong, M.: An Efficient Work-Stealing Scheduler for Task Dependency Graph. In: IEEE International Conference on Parallel and Distributed Systems (ICPADS). pp. 64–71 (2020)

[114] Lin, C.X., Huang, T.W., Yu, T., Wong, M.: A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In: ACM Great Lakes Symposium on VLSI (GLSVLSI). pp. 183–188 (2018)

[115] Lin, D.L., Huang, T.W.: A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In: IEEE High-performance and Extreme Computing Conference (HPEC) (2020)

[116] Lin, D.L., Huang, T.W.: Efficient GPU Computation using Task Graph Parallelism. In: European Conference on Parallel and Distributed Computing (Euro-Par). pp. 435–450 (2021)

[117] Lin, D.L., Huang, T.W.: Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. In: IEEE Transactions on Parallel and Distributed Systems (TPDS). pp. 3041–3052 (2022)

[118] 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). pp. 151–166 (2024)

[119] 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: ACM International Conference on Parallel Processing (ICPP). pp. 1–12 (2022)

[120] 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 Design Automation Conference (DAC). pp. 1–6 (2023)

[121] Lin, S., Guo, G., Huang, T.W., Sheng, W., Young, E., Wong, M.: GCS-Timer: GPU-Accelerated Current Source Model Based Static Timing Analysis. In: ACM/IEEE Design Automation Conference (DAC). pp. 1–6 (2024)

[122] Lin, Y., Li, W., Gu, J., Ren, H., Khailany, B., Pan, D.Z.: ABCDPlace: Accelerated Batch-Based Concurrent Detailed Placement on Multithreaded CPUs and GPUs. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 5083–5096 (2020)

[123] Liu, S., Pu, Y., Liao, P., Wu, H., Zhang, R., Chen, Z.: FastGR: Global Routing on CPU–GPU With Heterogeneous Task Graph Scheduler. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 2317–2330 (2023)

[124] Loff, J., Hoffman, R., Griebler, D., Fernandes, L.: High-Level Stream and Data Parallelism in C++ for Multi-Cores. In: Brazilian Symposium on Programming Languages (SBLP). pp. 41–48 (2021)

[125] Mack, J., Arda, S.E., Ogras, U.Y., Akoglu, A.: Performant, Multi-objective Scheduling of Highly Interleaved Task Graphs on Heterogeneous System on Chip Devices. In: IEEE Transactions on Parallel and Distributed Systems (TPDS). pp. 2148–2162 (2022)

[126] Mastoras, A., Gross, T.: Understanding Parallelization Tradeoffs for Linear Pipelines. In: ACM International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM). pp. 1–10 (2018)

[127] Mastoras, A., Gross, T.: Unifying Fixed Code Mapping, Communication, Synchronization and Scheduling Algorithms for Efficient and Scalable Loop Pipelining. In: IEEE Transactions on Parallel and Distributed Systems (TPDS). pp. 2136–2149 (2018)

[128] Mastoras, A., Gross, T.: Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines. In: ACM Transactions on Architecture and Code Optimization (TACO). pp. 1–26 (2019)

[129] Mastoras, A., Gross, T.: Load-balancing for Load-imbalanced Fine-grained Linear Pipelines. In: Parallel Computing. pp. 2136–2149 (2019)

[130] Mastoras, A., Yzelman, A.J.N.: Studying the Expressiveness and Performance of Parallelization Abstractions for Linear Pipelines. In: Workshop on Programming Models and Applications for Multicores and Manycores (PMAM). pp. 29–38 (2023)

[131] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning. In: Arxiv.org (2013)

[132] 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). pp. 89–95 (2024)

[133] Mower, M., Majors, L., Huang, T.W.: Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In: IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2) (2021)

[134] Navarro, A., Asenjo, R., Tabik, S., Cascaval, C.: Load Balancing Using Wok-Stealing for Pipeline Parallelism in Emerging Applications. In: ACM International Conference on Supercomputing. pp. 517–518 (2009)

[135] Ottoni, G., Rangan, R., Stoler, A., August, D.: Automatic Thread Extraction with Decoupled Software Pipelining. In: IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 105–118 (2005)

[136] Raman, E., Ottoni, G., Raman, A., Bridges, M., August, D.: Parallel-Stage Decoupled Software Pipelining. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 114–123 (2008)

[137] Rangan, R., Vachharajani, N., Ottoni, G., August, D.: Performance Scalability of Decoupled Software Pipelining. In: IEEE/ACM International Symposium on Code Generation and Optimization (TACO). pp. 1–25 (2008)

[138] Reed, E.C., Chen, N., Johnson, R.E.: Expressing Pipeline Parallelism using TBB Constructs: A Case Study on What Works and What Doesn't. In: Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH). pp. 133–138 (2011)

[139] del Rio Astorga, D., Dolz, M., Fernández, J., García, J.: A Generic Parallel Pattern Interface for Stream and Data Processing. In: Concurrency and Computation: Practice and Experience (2017)

[140] Sanchez, D., Lo, D., Yoo, R., Sugerman, J., Kozyrakis, C.: Dynamic Fine-Grain Schedulig of Pipeline Parallelism. In: IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 22–32 (2011)

[141] Schardl, T., Lee, I.T.A.: OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In: ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programmi (PPoPP). pp. 189–203 (2023)

[142] Suleman, M., Qureshi, M., Khubaib, Patt, Y.: Feedback-Directed Pipeline Parallelism. In: IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 147–156 (2010)

[143] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. In: MIT Press (2018)

[144] Sutton, R.S., McAllester, D., Singh, S., Mansour, Y.: Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: International Conference on Neural Information Processing Systems. pp. 1057—1063 (1999)

[145] Tong, J., Chang, L., Ogras, U.Y., Huang, T.W.: BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 789–793 (2024)

[146] Tong, J., Lee, W.L., Ogras, U.Y., Huang, T.W.: Scalable Code Generation for RTL Simulation of Deep Learning Accelerators with MLIR. In: International European Conference on Parallel and Distributed Computing (Euro-Par) (2025)

[147] Visweswariah, C., Ravindran, K., Kalafala, K., Walker, S., Narayan, S., Beece, D., Piaget, J., Venkateswaran, N., Hemmett, J.: First-Order Incremental Block-Based Statistical Timing Analysis. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). pp. 2170–2180 (2006)

[148] Vogel, A., Griebler, D., Fernandes, L.: Providing High-level Self-adaptive Abstractions for Stream Parallelism on Multicores. In: Journal of Software. pp. 1194–1217 (2021)

[149] Vogel, A., Mencagli, G., Griebler, D., Danelutto, M., Fernandes, L.: Towards On-the-fly Self-Adaptation of Stream Parallel Patterns. In: IEEE International Conference on Parallel, Distributed and Network-Based Processing (PDP). pp. 89–93 (2021)

[150] Watkins, C.J.C.H., Dayan, P.: Q-learning. In: Machine Learning. pp. 279–292 (1992)

[151] Wu, L., Cui, P., Pei, J., Zhao, L.: Graph Neural Networks. In: Springer (2022)

[152] Zamani, Y., Huang, T.W.: A High-Performance Heterogeneous Critical Path Analysis Framework. In: IEEE High-Performance Extreme Computing Conference (HPEC) (2021)

[153] Zhang, B., Chang, C., Chiu, C.H., Lin, D.L., Sui, Y., Chang, C.C., Chung, Y.H., Lee, W.L., Guo, Z., Lin, Y., Huang, T.W.: iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 407–415 (2025)

[154] 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 Design Automation Conference (DAC). pp. 1–6 (2024)

[155] Zhou, K., Guo, Z., Huang, T.W., Lin, Y.: Efficient Critical Paths Search Algorithm using Mergeable Heap. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 190–195 (2022)