QiMeng-EvoPartition: Rethinking the Impact of Partitioning for Automated Pipeline Design

Qicheng Wang[1,2,3], **Rui Zhang**[1], **Shuyao Cheng**[1], **Chongxiao Li**[1,2,3], **Pengwei Jin**[1]
Zidong Du[1], **Xing Hu**[1], **Qi Guo**[1], **Yunji Chen**[1,2*]
[1] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences
[2] University of Chinese Academy of Sciences
[3] Cambricon Technologies

Abstract

As a key technique for improving throughput by increasing clock frequency and reducing Cycles per Instruction (CPI), pipeline design increasingly relies on automated methods with the growing scale of modern circuits. However, existing automated pipelining methods often decouple partitioning from CPI optimization, implicitly assuming that partitioning affects only clock frequency. We observe that different partitioning strategies significantly alter the pipeline stage differences between logic gates, thereby impacting the average number of stall cycles and ultimately affecting CPI. Thus, automated pipeline partitioning should jointly reduce CPI and improve clock frequency while ensuring functional correctness, resulting in a multi-objective optimization problem. To address this issue, we propose EvoPartition, an evolutionary partitioning framework for automated pipeline design. Specifically, EvoPartition first inserts virtual nodes to guarantee functional correctness. Then, through iterative evolution for reducing CPI and improving clock frequency, pipeline stage assignments of all nodes are determined. Experimental results on ISCAS85 and EPFL show that EvoPartition achieves a 7.34% CPI optimization by reducing stage differences by 16.23%, and further improves throughput by 7.22% compared to the state-of-the-art.

1 Introduction

Pipelining is a fundamental technique in digital circuit design, widely used to enhance clock frequency and reduce the Cycles per Instruction (CPI) [Dubey and Flynn, 1990; Ramamoorthy and Li, 1977; Zeng *et al.*, 2022; Zou *et al.*, 2023; Qi *et al.*, 2024]. By breaking down complex operations into simpler stages, the pipeline increases its clock frequency, allowing for parallel instruction execution, which ultimately reduces CPI and enhances circuit throughput. An efficient pipeline should hinge on accurately eliminating data dependencies in circuit loops, particularly the potential read-after-write (RAW) [Aagaard and Leeser, 1994; Hennessy and Patterson, 2011], to reduce pipeline stalls while preserving functional correctness [Shen and Lipasti, 2013].

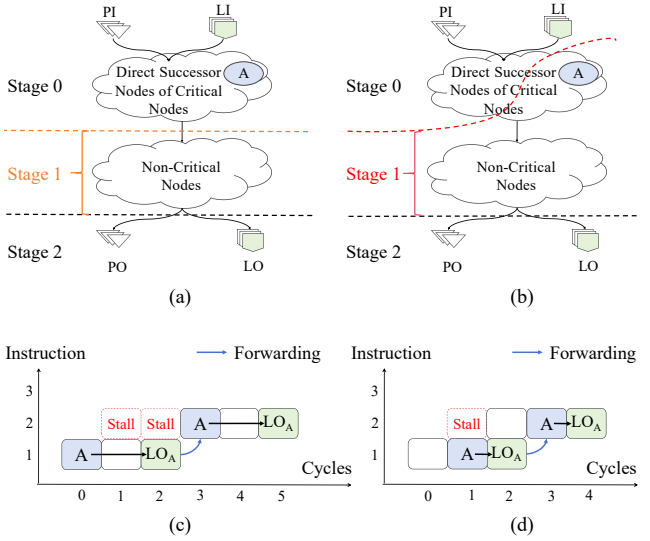


Figure 1: A simplified example of how pipeline stage partition affects the average stall cycles. (a)(b) Logic gate A is assigned to stage 0 and stage 1, respectively, while data flow is $LI \rightarrow A \rightarrow LO_A$. Under the same forwarding path $LO_A \rightarrow A$, assigning A to stage 0 requires two stall cycles to avoid the data conflicts (c), while assigning it to stage 1 reduces to one stall cycle (d).

Historically, pipeline design has been predominantly guided by manual analysis, which is not only labor-intensive but also error-prone. To reduce manual workload and improve optimization quality, automated pipeline technology [Cheng *et al.*, 2024; Nurvitadhi *et al.*, 2011; Liu *et al.*, 2019; Zagieboylo *et al.*, 2022; Majumder and Bondhugula, 2023; Shahane *et al.*, 2023] is proposed to minimize the average stall cycle after a RAW conflict occurs for mitigating performance degradation by optimizing the data scheduling strategy. However, partitioning techniques in existing automated methods operate under a critical misconception: pipeline partitioning affects only the clock frequency of the circuit and will not change the CPI. Consequently, existing automatic partition technologies evenly distribute the logic gates to maximize clock frequency.

We observe that partitioning can also impact the pipeline

*Yunji Chen (cyj@ict.ac.cn) is the corresponding author.

CPI, providing a new direction for further optimization. When using forwarding between logic gates to eliminate RAW conflicts, even if the forwarding path is fixed, different partitioning will cause changes in the pipeline stage differences, thereby affecting the average number of stall cycles required in forwarding. The average stall cycles directly determine CPI: the more stall cycles a pipeline has on average, the higher the CPI and the worse the design throughput. A simplified example is shown in Fig. 1. Node A can be assigned to pipeline stage 0 in Fig. 1(a), or pipeline stage 1 in Fig. 1(b). Even the two partitioning methods maintain identical clock frequencies, they have a significant difference in the CPI. When A is at stage 0 in Fig. 1(c), it is two stages away from node LO_A, i.e. their pipeline stage difference is 2, and needs to wait for two stall cycles, with a CPI of 3; and when it is allocated to stage 1 in Fig. 1(d), their pipeline stage difference is 1 so that only one stall cycle is required and the CPI is reduced to 2.5. Summarily, the partitioning result directly affects the pipeline stage differences between logic gates with data flow, thereby affecting the average pipeline stall cycles and further determining CPI.

Based on the above analysis, an ideal pipeline partition can reduce the CPI of circuits while improving the clock frequency. The challenge of pipeline partition are twofold: ensuring design correctness and efficiently searching for better design throughput. On the one hand, pipeline partitioning is prone to introducing data conflicts, which may result in functional incorrectness. On the other hand, pipeline partitioning aims to minimize CPI while maximizing clock frequency, thereby forming a multi-objective optimization problem in which the two objectives are mutually constrained and difficult to optimize.

To address the above issue, we propose EvoPartition, a novel evolutionary partitioning framework for automated pipeline design. The key insight of EvoPartition is to obtain a reasonable partitioning by constructing a specialized evolutionary framework, under the premise of ensuring functional correctness, with the aim of improving clock frequency and reducing CPI. Specifically, the proposed EvoPartition is divided into three steps: Correctness-Oriented Initialization, Performance-Oriented Evolution, Individual Post-Processing. 1) In the Correctness-Oriented Initialization step, a population is initialized based on the given circuit, where each individual represents a pipeline partitioning scheme. In this process, the virtual nodes are inserted to ensure the correctness of the evolution process. 2) In the Performance-Oriented Evolution step, the individuals are evolving by applying evolutionary operators in evolutionary iterations. To achieve our goal of reducing CPI and improving clock frequency, a novel fitness score is introduced as the selection criterion, The fitness score uses pipeline stage differences as a substitute for CPI and the longest critical path length as a surrogate for clock frequency, enabling efficient iterative evolution of individual. 3) In the Individual Post-Processing step, the best individual partition is post-processed by removing the virtual nodes, and then the stage assignment is completed by even partitioning the remaining nodes without violating the topological order. Different from the conventional equal partitioning strategy which solely focuses on clock frequency, EvoPartition also holistically considers the impact of pipeline partitioning on the average number of pipeline stall cycles, thereby optimizing CPI and the overall design throughput.

We deploy and evaluate EvoPartition on standard datasets for EDA algorithms, including ISCAS85 [Hansen *et al.*, 1999] and EPFL [Amarú *et al.*, 2015]. Compared to the state-of-the-art, the proposed partition strategy reduces the average number of stall cycles by 16.23%, and thus reduces the pipeline CPI by 7.34%. With little design timing overhead, the overall design throughput is about 7.22% over the state-of-the-art.

The contributions of this article are as follows:

- We observe that partitioning also influences pipeline CPI, as it directly determines the stage differences, which in turn affect the average pipeline stall cycles and ultimately CPI.
- We propose EvoPartition, which iteratively evolves pipeline stage partitions under a functional-correctness constraint to jointly reduce CPI and improve clock frequency.
- Evaluation on ISCAS and EPFL benchmarks shows that EvoPartition outperforms the state-of-the-art, reducing CPI by 7.34% and improving overall throughput by 7.22%.

2 Related Work

Most pipelining techniques aim to improve clock frequency through scheduling or register insertion, without explicitly addressing pipeline stage partitioning [Liu *et al.*, 2019; Fauth *et al.*, 1995; Kroening and Paul, 2001; Marinescu and Rinard, 2001; Wang *et al.*, 2024]. High-level tools [Campbell and Day, 2003; Soviani *et al.*, 2006; Nurvitadhi *et al.*, 2010; Kim *et al.*, 2015; Zagieboylo *et al.*, 2022; Dong *et al.*, 2023] focus on semantic scheduling, while circuit-level tools [Ganusov *et al.*, 2016; Iştoan and De Dinechin, 2017; Lee and Gerstlauer, 2017; Parrot *et al.*, 2021; Hu *et al.*, 2023; Zhong *et al.*, 2024; Wang *et al.*, 2025] apply retiming or graph cuts but adopt conservative strategies that may introduce stalls. Cheng et al. [Cheng *et al.*, 2024] perform gate-level pipelining with forwarding, yet still assume stage partitioning only affects frequency, overlooking CPI impact.

Meanwhile, pipeline partitioning can be formulated as a graph partitioning problem and is often addressed using heuristic methods, such as KL-based algorithms [Kernighan and Lin, 1970; Fiduccia and Mattheyses, 1988; Krishnamurthy, 1984] and simulated annealing [Kirkpatrick *et al.*, 1983]. However, these approaches are prone to local optima and incur high computational cost. Although genetic algorithms [Bagley, 1967; Baruch *et al.*, 1999; Areibi, 2000] have also been explored for this task, these methods mainly change the physical grouping of circuit components and do not modify the actual pipeline structure. As a result, pipeline behavior remains unchanged, and the impact on CPI is not considered. We instead propose a specialized evolutionary partitioning framework for automated pipeline partitioning that ensures functional correctness while reducing CPI and improving clock frequency.

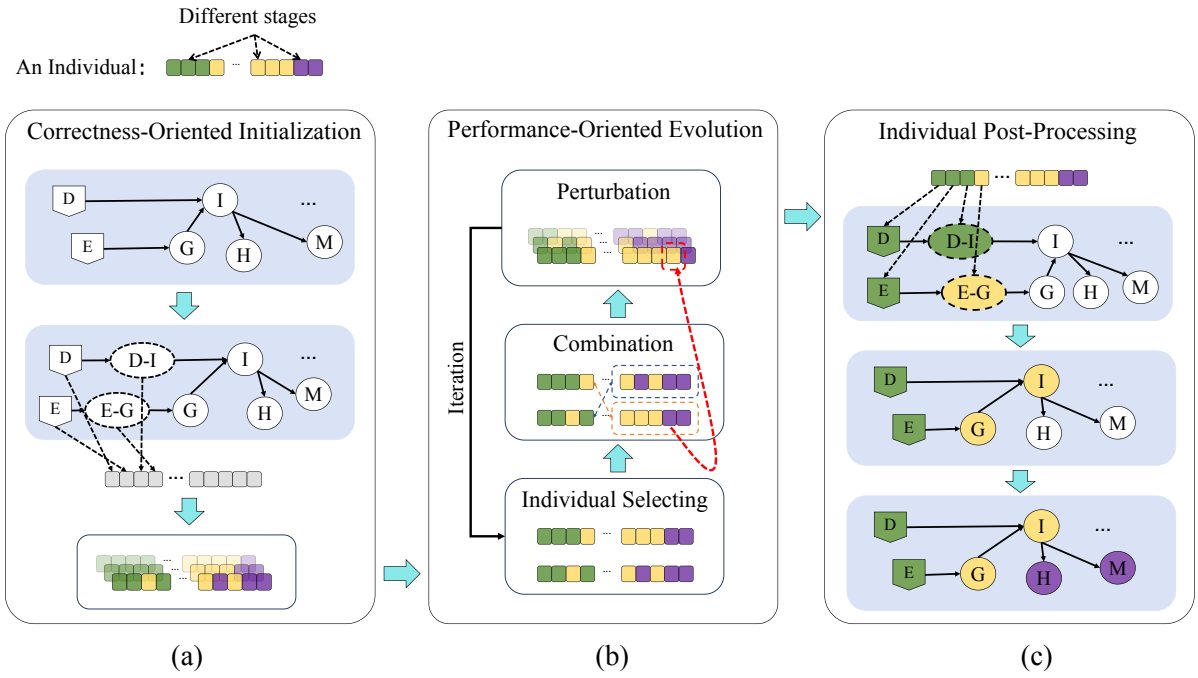


Figure 2: The overview of EvoPartition, which is mainly divided into three steps: (a) Correctness-Oriented Initialization. (b) Performance-Oriented Evolution. (c) Individual Post-Processing.

3 Methods

In this work, we propose EvoPartition, an evolutionary partitioning framework for automated pipeline design. The goal of EvoPartition is to improve clock frequency while reducing CPI, under the constraint of ensuring functional correctness. Based on our observation, to reduce CPI, EvoPartition is designed to minimize the pipeline stage differences between critical nodes for reducing average stall cycles caused by RAW conflicts, while load balancing to improve clock frequency. At the same time, the circuit topology order should be maintained to ensure functional correctness. It is a multi-objective optimization problem which poses significant challenges.

To achieve these goals, EvoPartition constructs a specialized evolutionary framework that is tailored to pipeline partitioning, as evolutionary algorithms exhibit excellent performance in high-dimensional multi-objective optimization. Given an unpartitioned pipeline circuit, the framework of EvoPartition is mainly divided into three steps, as shown in Fig. 2: 1) *Correctness-Oriented Initialization:* Based on the given circuit, a population is initialized where each individual specifies a complete stage assignment for all critical nodes. To ensure functional correctness in subsequent iterative evolution, virtual nodes are inserted between the critical input nodes and their direct successor nodes. 2) *Performance-Oriented Evolution*: In evolutionary iterations, the individuals are evolved by applying evolutionary operators and are selected by a novel fitness function, progressively improving clock frequency while reducing CPI. The fitness function employs pipeline stage differences as a surrogate for CPI and the longest critical path as a proxy for clock frequency, thereby reducing computational overhead while achieving equivalent optimization effectiveness. 3) *Individual Post-Processing*: The best individual partition in the iteration process is post-processed by removing the virtual nodes, while its non-critical nodes are evenly partitioned under ensuring topological order. The framework finally outputs the final stage partition in which all nodes have determined the allocation.

3.1 Correctness-Oriented Initialization

In the correctness-oriented initialization step, we aim to complete the population initialization. EvoPartition takes an unpartitioned pipeline circuit as input, represented as an And-Inverter Graph (AIG) $G = (\mathcal{V}, E)$, where $\mathcal{V}$ denotes gates and $E \subseteq \mathcal{V} \times \mathcal{V}$ captures data dependencies. Each individual in EvoPartition is a pipeline stage assignment of circuit nodes, represented as a partition function $P : \mathcal{V} \rightarrow \{0, 1, .., S-1\}$, where S represents the total number of pipeline stages, $P(V)$ indicates that node $V \in \mathcal{V}$ is assigned to the $P(V)$-th pipeline stage. However, evolving the stage assignments for all nodes leads to an excessively large search space, substantially increasing computational complexity. Actually, pipeline CPI is predominantly determined by a small set of nodes where occur frequent data flows and exit complex data dependencies. We define them as critical nodes, denoted as $V_c \in \mathcal{V}$, including Latch Input nodes (LI), their corresponding Latch Output nodes (LO), and LI nodes' direct successors AND nodes. For example, as shown in Fig. 3(a), nodes D and G both serve as direct predecessors of node I, and are all critical

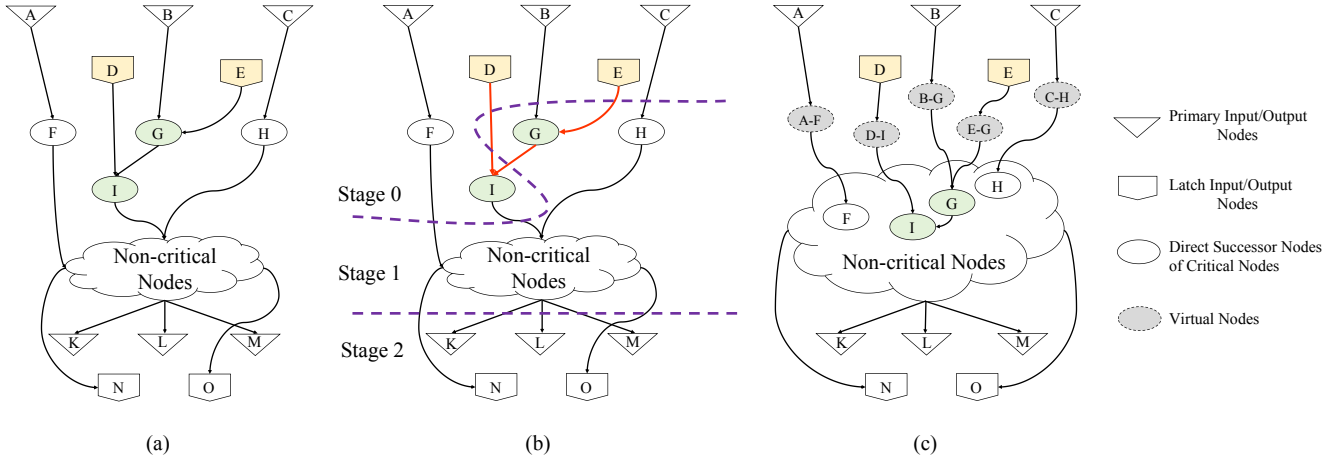


Figure 3: (a) Critical nodes simultaneously participate in data transfers from multiple input nodes. (b) Pipeline stage assignment conflicts arise across multiple paths, violating the topological order. (c) Virtual nodes are inserted between critical input nodes and their successors, preventing conflicts between the data flow and the assigned pipeline stage ordering.

nodes. Other non-critical nodes typically do not form significant data dependency paths and have limited impact on CPI, whereas their partitioning primarily affects the circuit's clock frequency. Therefore, we apply evolutionary optimization only to critical nodes, i.e., each individual is a pipeline stage assignment of critical nodes, which is sufficient to achieve comparable optimization benefits to full-node evolution.

To construct an initial population, a straightforward method is applying random perturbations to an even partitioning based on the longest critical path. However, such random initialization can violate the topological constraint, introduce data conflicts, and ultimately lead to functional incorrectness. As shown in Fig. 3(b). Node I is assigned to pipeline stage 0 based on the data flow path $D \rightarrow I$, while node G may be assigned to pipeline stage 1 according to the path $E \rightarrow G \rightarrow I$. However, since node G is a predecessor of node I, the pipeline stage number of G must not exceed that of I according to the topological constraint, otherwise, it may lead to functional incorrectness.

To resolve such conflicts fundamentally, we insert virtual nodes between critical input nodes and their successors, e.g., D–I, E–G shown in Fig. 3(c). After inserting virtual nodes, the topological constraints among critical nodes are decoupled, so they are no longer directly coupled with each other. This ensures that each critical node has a unique predecessor, thereby avoiding data conflicts and guaranteeing functional correctness. Ultimately, the initial population is generated by applying random perturbations to an even partitioning based on the longest critical path after inserting virtual nodes.

3.2 Performance-Oriented Evolution

In the performance-oriented evolution step, the critical nodes are assigned to the optimal pipeline stages through a fitness-guided evolution. The evolutionary process is a multi-objective optimization that aims to reduce CPI while simultaneously improving clock frequency. Based on our observations, we found that smaller pipeline stage differences between critical nodes enable fewer average stall cycles required to eliminate RAW dependencies by forwarding, thereby achieving a lower CPI. Therefore, we reduce CPI by minimizing the pipeline stage differences between critical nodes. At the same time, improving clock frequency requires balancing critical path lengths across pipeline stages to achieve load balancing. Therefore, we aim to improve clock frequency by minimizing critical path lengths. Finally, we construct a multi-objective fitness score that jointly optimizes CPI and clock frequency by minimizing the pipeline stage differences between critical nodes and the length of the critical path. The fitness score F is formulated as:

$$F = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N}\left(\frac{1}{M_j}\sum_{k=1}^{M_j}(P(V_c^{LO_i}) - P(V_c^{LI_{jk}}))^2\right)} \times \max_{s\in S}\mathcal{P}_s, \quad (1)$$

where $P(V_c^{LO_i})$ is the pipeline stage number of the i-th critical output node and $P(V_c^{LI_{jk}})$ is the pipeline stage number of the k-th successor of the j-th critical input node. N denotes the total number of critical input/output nodes, and M_j denotes the number of successor nodes corresponding to the j-th critical input node. $\mathcal{P}_s$ represents the length of the critical path in pipeline stage s, and $\max_{s\in S}\mathcal{P}_s$ is the maximum critical path length across all stages. Our ultimate goal is to minimize F, thereby achieving our multi-objective optimization of reducing CPI and improving clock frequency.

Fig. 2(b) illustrates the evolution workflow in the performance-oriented evolution step. Starting with the initial population, EvoPartition performs an iterative fitness-guided evolution to continuously improve the individual quality. At each iteration, we randomly sample n individuals from the population and select the two individuals with the lowest objective fitness score as the winning individuals for generating new individuals. Based on two winning individuals, a new evolved individual is generated by combining their critical-

node stage assignments. Specifically, the new individual combines the assignments on one side of the split point from one individual and those on the other side from the other, encouraging reuse of the high-quality stage assignments while increasing diversity. We then apply perturbation operation to generate multiple individuals for enhancing population diversity: the stage assignments of some critical nodes within an individual are modified with a predefined probability. To avoid premature convergence, we incorporate a simulated annealing mechanism during the perturbation step. If the perturbed individual has better fitness, it is accepted directly; otherwise, it is accepted with a probability that decreases with temperature. Note that, by decoupling the topological constraint among critical nodes through inserting virtual nodes, the evolutionary process does not generate individuals that violate topological order, thereby guaranteeing the correctness of the evolved individuals. The iteration will be repeated after the perturbation operation completed. The evolution process terminates when the predefined iteration budget is reached, and the best individual selected throughout the entire evolution serves as the final critical-node partitioning.

3.3 Individual Post-Processing

In the individual post-processing step, the best individual partition in the iteration process is post-processed by removing the virtual nodes and even partitioning the non-critical nodes.

We firstly need to remove the virtual nodes, restoring the original AIG. We assign pipeline stages to the successors of the virtual nodes, since these nodes correspond to critical nodes in the original AIG. The pipeline stage of each such node is assigned to the maximum stage among its predecessor virtual nodes. This strategy prevents these nodes from being assigned to an excessively early stage, otherwise increasing pipeline stage differences and leading to a higher CPI. On this basis, the logic levels $\mathcal{L}$ of the critical nodes can be computed according to:

$$\mathcal{L}(V_c^i) = P(V_c^i) \times \max_{s \in S} \mathcal{P}_s, \tag{2}$$

where V_c^i is the i-th node among all critical nodes and $\mathcal{L}(V_c^i)$ denotes the logic level of the i-th critical node.

After that, the partitioning of non-critical nodes will be guided by the logic levels of the resulting critical nodes. Considering that non-critical nodes rarely contribute to major data dependency paths and have a negligible effect on CPI, their partitioning mainly influences the circuit's clock frequency. Therefore, a simple equal distribution strategy can be adopted, assigning these non-critical nodes as evenly as possible across pipeline stages. This helps to control the length of the critical path in each stage and prevents clock frequency degradation caused by load imbalance. Specifically, all non-critical nodes are traversed in the topological order of the original AIG, and their logic levels are computed according to:

$$\mathcal{L}(V_{\text{nc}}^i) = \max_{v \in \text{Pred}(V_{\text{nc}}^i)} \mathcal{L}(v) + 1, \tag{3}$$

where V_{nc}^i denotes the i-th non-critical node, and $\text{Pred}(V_{\text{nc}}^i)$ denotes the set of all predecessor nodes of the i-th non-critical node. Finally, the pipeline stages of all nodes are determined based on their logic levels, according to:

$$P(V^i) = \mathcal{L}^{V^i} / \max_{s \in S} \mathcal{P}_s, \tag{4}$$

where V_i denotes the i-th node in the set of all nodes.

As a result, the final stage partition is obtained, in which the stage partition of all nodes is determined.

4 Evaluation

We conducted comprehensive experiments to illustrate that EvoPartition can effectively reduce CPI and increase throughput. We first compare the overall design performance with the state-of-the-art. Next, we demonstrate the effectiveness of the use of critical nodes assignment. Then we further compare EvoPartition with other partitioning algorithms. To demonstrate the potential application of automated pipeline partitioning, we demonstrate that the proposed method can find a better-performance pipeline design than human expert design.

Experimental Setups. We use the open-source EDA tool Yosys [Wolf *et al.*, 2013] to preprocess all circuits and convert them into the AIG format required by EvoPartition. Subsequently, based on the pipeline partitions generated by EvoPartition, we perform logic synthesis using the Design Compiler tool on a 65nm standard cell library to obtain accurate performance metrics. All algorithms are executed on a computing platform equipped with two Intel Xeon Gold 6230 CPUs.

Baselines. We compare the proposed method with two types of baselines, including the entire automated pipelining methods and the heuristic partitioning methods. For automated pipelining, we compare with both the state-of-the-art method [Cheng *et al.*, 2024] and the mostly used one in the current EDA flow. Additionally, we also compare the proposed EvoPartition with three heuristic partitioning methods including Simulated Annealing [Kirkpatrick *et al.*, 1983], Kernighan–Lin algorithm [Kernighan and Lin, 1970], METIS [Karypis and Kumar, 1998] to demonstrate the effectiveness.

Benchmarks. We tested the feasibility of our method on four standard open source benchmarks, including ISCAS85 [Hansen *et al.*, 1999], EPFL [Amarú *et al.*, 2015], OpenCores [OpenCores, 1999], and RTLLM [Lu *et al.*, 2024]. In the ISCAS85 and EPFL benchmarks, in order to expand the diversity of the benchmarks, we set up Single-Function and Multi-Function in the same way as the state-of-the-art method [Cheng *et al.*, 2024], where multi function circuits are combined from single function circuits.

Metrics. To comprehensively evaluate the pipeline performance improvements achieved by the proposed EvoPartition, we adopt the following four key metrics: 1) CPI: the average number of clock cycles required to execute a single instruction. This is the most important performance metric in automated pipelining. 2) Clock Frequency: the maximum operating frequency the circuit can achieve after logic synthesis. 3) Throughput: the number of inputs that can be executed per unit time in the pipeline. 4) Average Stall Cycles: the average number of stall cycles observed during the simulation of 1000

Case	Single-Function									Multi-Function								
	EDA			[Cheng *et al.*, 2024]			EvoPartition (ours)			EDA			[Cheng *et al.*, 2024]			EvoPartition (ours)		
	CPI	CLF	Thpt	CPI	CLF	Thpt	CPI	CLF	Thpt	CPI	CLF	Thpt	CPI	CLF	Thpt	CPI	CLF	Thpt
ISCAS/c17	3.00	5.56	1.85	1.96	3.84	1.96	**1.53**	3.125	**2.04**	3.00	3.03	1.01	1.80	3.03	1.70	1.80	3.03	1.70
ISCAS/c432	3.00	3.12	1.04	1.99	2.86	1.43	**1.98**	2.86	**1.44**	3.00	2.56	0.85	1.80	2.56	1.42	**1.76**	2.56	**1.45**
ISCAS/c499	3.00	2.70	0.90	1.61	2.70	1.68	1.61	2.70	1.68	3.00	2.38	0.79	1.80	2.43	1.35	**1.78**	2.44	**1.37**
ISCAS/c880	3.00	2.50	0.83	1.99	2.50	1.26	1.99	2.50	1.26	3.00	2.94	0.98	1.81	2.63	1.46	1.81	2.63	1.46
ISCAS/c1355	3.00	2.50	0.83	1.62	2.50	1.54	**1.59**	2.50	**1.57**	3.00	2.70	0.90	1.80	2.94	1.63	**1.78**	2.94	**1.65**
ISCAS/c1908	3.00	2.38	0.79	1.63	2.38	1.46	**1.59**	2.38	**1.49**	3.00	2.50	0.83	1.81	2.43	1.34	**1.76**	2.43	**1.38**
ISCAS/c3540	3.00	2.22	2.22	1.97	2.17	1.10	1.97	**2.22**	**1.13**	3.00	2.38	0.79	1.80	1.96	1.09	**1.76**	1.96	**1.11**
ISCAS/c5315	3.00	2.50	2.50	1.67	2.17	1.30	1.67	2.17	1.30	3.00	2.63	0.88	1.94	2.27	1.17	1.94	2.27	1.17
ISCAS/c6288	3.00	1.29	0.43	1.61	1.29	0.81	1.61	1.29	0.81	3.00	1.29	0.43	1.80	1.25	0.69	**1.58**	1.25	**0.79**
EPFL/adder	3.00	2.44	0.81	2.43	1.67	0.69	2.43	1.67	0.69	3.00	1.92	0.64	2.18	1.69	0.78	2.18	1.69	0.78
EPFL/arbiter	3.00	2.40	0.80	2.40	2.04	0.85	**2.37**	**2.08**	**0.88**	3.00	2.38	0.79	2.17	2.00	0.92	2.17	2.00	0.92
EPFL/bar	3.00	3.33	1.11	2.27	2.50	1.10	2.27	**2.63**	**1.16**	3.00	3.22	1.07	2.19	2.71	1.24	**2.15**	**2.78**	**1.30**
EPFL/cavlc	3.00	3.57	1.19	1.62	3.57	2.21	**1.55**	3.57	**2.30**	3.00	3.33	1.11	1.79	2.86	1.60	**1.78**	2.86	**1.61**
EPFL/ctrl	3.00	3.85	1.28	1.61	3.85	2.38	**1.25**	3.33	**2.67**	3.00	3.12	1.04	1.81	3.03	1.67	**1.79**	3.03	**1.69**
EPFL/dec	3.00	6.25	2.08	1.99	3.85	1.93	**1.42**	3.85	**2.71**	3.00	2.78	0.93	2.12	2.78	1.31	**1.63**	2.78	**1.71**
EPFL/div	3.00	0.12	0.04	2.40	0.11	0.04	**1.70**	**0.12**	**0.07**	3.00	0.11	0.04	2.40	0.11	0.05	**1.73**	**0.12**	**0.07**
EPFL/i2c	3.00	2.78	0.93	2.01	2.17	1.08	**1.89**	2.17	**1.15**	3.00	3.33	1.11	2.10	2.56	1.22	2.10	2.56	1.22
EPFL/int2float	3.00	3.85	1.28	1.54	2.86	1.85	**1.36**	2.86	**2.10**	3.00	2.94	0.98	1.82	3.23	1.77	**1.36**	2.94	**2.16**
EPFL/log2	3.00	0.48	0.16	1.62	0.37	0.23	**1.54**	0.37	**0.24**	3.00	0.37	0.12	1.62	0.38	0.23	**1.58**	**0.47**	**0.30**
EPFL/max	3.00	1.67	0.56	2.41	1.61	0.67	**2.37**	1.61	**0.68**	3.00	1.69	0.56	2.34	1.69	0.72	**2.20**	**1.72**	**0.78**
EPFL/multiplier	3.00	0.56	0.19	2.39	0.56	0.24	**1.90**	0.55	**0.30**	3.00	0.54	0.18	2.19	0.56	0.26	**1.86**	0.55	**0.30**
EPFL/priority	3.00	2.56	0.85	1.98	2.33	1.17	**1.46**	2.33	**1.59**	3.00	1.78	0.59	1.96	1.56	0.80	1.96	1.56	0.80
EPFL/router	3.00	6.67	2.22	1.002	6.51	6.50	1.002	6.51	6.50	3.00	2.04	0.68	2.00	2.17	1.09	**1.41**	2.08	**1.48**
EPFL/sin	3.00	0.95	0.32	1.62	0.71	0.44	**1.24**	0.71	**0.57**	3.00	0.77	0.26	1.80	0.71	0.40	**1.59**	**0.94**	**0.59**
EPFL/sqrt	3.00	0.58	0.19	2.26	0.58	0.26	**2.15**	0.57	**0.27**	3.00	0.11	0.04	2.39	0.11	0.04	**2.26**	0.11	**0.05**
EPFL/voter	3.00	1.22	0.41	1.76	1.18	0.67	**1.72**	1.18	**0.68**	3.00	0.71	0.24	1.99	1.43	0.72	**1.72**	1.43	**0.83**
Average	-	-	-	100.00%	100.00%	100.00%	92.74%	98.25%	106.97%	-	-	-	100.00%	100.00%	100.00%	92.58%	100.10%	107.46%

Table 1: Comparison of CPI, Clock Frequency(GHz), and Throughput across three methods (EDA, [Cheng *et al.*, 2024], Ours) under Single and Multi Function cases. The cases with horizontal lines show an improvement of more than 20% compared to the state-of-the-art method.

inputs. We use this metric to demonstrate how critical nodes assignment exactly impacts the CPI.

4.1 Overall Evaluation on CPI and Throughput

We conducted comparisons on multiple standard circuit benchmarks among the proposed EvoPartition, EDA optimization, and the state-of-the-art method [Cheng *et al.*, 2024]. As shown in Table 1, the EvoPartition achieved an average CPI reducing of 7.26% on the Single-Function benchmark compared to [Cheng *et al.*, 2024], bringing a throughput improvement of approximately 6.97%. Besides, across seven typical circuits, EvoPartition achieved a CPI improvement exceeding that of [Cheng *et al.*, 2024] by more than 20%. On the constructed Multi-Function benchmark, the optimization effect of the EvoPartition is still quite significant. Compared with [Cheng *et al.*, 2024], EvoPartition still maintains an optimization margin of 7.42% in terms of CPI optimization. Furthermore, the final throughput was improved by 7.46% compared to [Cheng *et al.*, 2024], meanwhile, in eight typical circuits the optimization margin exceeds 20%.

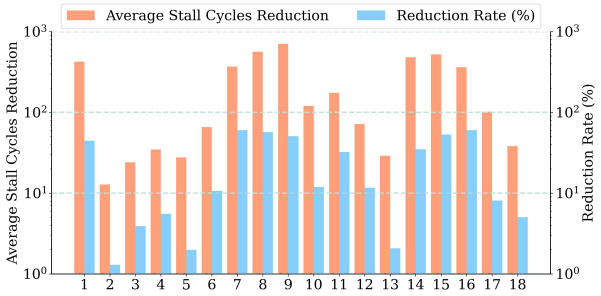
In both benchmarks, the clock frequency of the EvoPartition is lower than that of the EDA tools. We believe this is because traditional EDA tools use the single objective of clock frequency as the optimization direction and ignore reducing CPI. However, automated pipeline partitioning is inherently a multi-objective optimization problem, which seeks trade-offs between clock frequency and CPI. Consequently, optimizing a single objective alone is insufficient. Therefore, our method uses multi-objective optimization and seeks the Pareto optimality between clock frequency and CPI.

Across both the Single-Function and Multi-Function benchmarks, EvoPartition achieves an average 7.34% reduction in CPI and a 7.22% improvement in throughput compared with the state-of-the-art method [Cheng *et al.*, 2024]. Overall, the above results show that the proposed EvoPartition can reduce the CPI by rationally partitioning the pipeline stages of critical nodes without sacrificing excessive clock frequency, thereby improving the pipeline throughput.

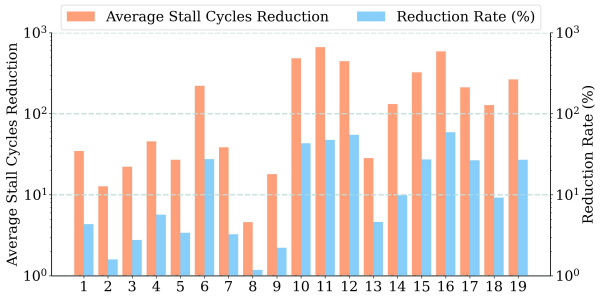
4.2 The Effectiveness of Critical Nodes Assignment

In EvoPartition, pipeline stage differences are employed as a proxy for optimizing CPI. Therefore, to validate the effectiveness of this proxy, we measured the correlation between pipeline stage differences and CPI at critical nodes. We randomly sampled some circuits from the Single-Function and Multi-Function benchmarks to calculate the correlation. The results show that the Pearson coefficient of the pipeline stage differences between critical nodes and the actual CPI is 0.86, which has a strong positive correlation.

At the same time, we quantitatively analyzed the reduction in the average stall cycles brought by the EvoPartition in comparison to the state-of-the-art method [Cheng *et al.*, 2024] over 1000 simulation runs. The results are presented in Fig. 4, where the horizontal axis is the circuit case and the vertical axis is a logarithmic coordinate, the orange bars represent the absolute the average number of stall cycles reduced by EvoPartition, and the blue bars indicate the proportion of the average stall cycles reduction. The circuits that exhibited no improvement, i.e., zero difference in the average stall cycles, are not displayed in the figure. Fig. 4(a) shows the results on the Single-Function benchmark. On average, cir-



(a)



(b)

Figure 4: The number and ratio of the average stall cycles reduced on Single-Function and Multi-Function benchmarks.

cuits in the Single-Function benchmark experienced a reduction of 160 stall cycles. Fig. 4(b) presents the results on the Multi-Function benchmark, in which eight circuits achieved a reduction of over 200 stall cycles, with an overall average reduction of 134. Across both Single- and Multi-Function benchmarks, EvoPartition reduced stall cycles by an average of 16.23%, equivalent to a reduction of approximately 150 stall cycles, corresponding to the overall CPI reduced by 7.34%. Notably, for the EPFL/div circuit in the Single-Function benchmark, EvoPartition reduced the average stall cycles by as many as 708, corresponding to a CPI reduction of approximately 30%. These results show that reducing the pipeline stage differences between critical nodes can effectively decrease the average stall cycles caused by RAW conflicts, thereby reducing the overall CPI.

4.3 Comparison with the Human Design

To further verify the effectiveness of the EvoPartition in pipeline partitioning tasks, we compared it with partitioning schemes manually designed by experienced human experts. As shown in Table 2, in the Single-Function benchmark, EvoPartition reduces the average CPI by 10.31% compared to human-designed. For nine circuits, the improvement exceeds 20%, demonstrating the method's effectiveness across diverse circuit types. In the Multi-Function benchmark, EvoPartition showcases even stronger optimization performance, achieving an average 8.57% CPI reduction over human expert designs. On eight complex circuits, the CPI reduction exceeds 20%, highlighting its effectiveness in handling high-complexity designs. Overall, EvoPartition achieves CPI reduction on 47 out of 52 circuits across both the Single-Function and Multi-Function benchmarks.

	Human-Design	Ours	Rate
Single-Function	1.94	**1.74**	**89.69%**
Multi-Function	2.10	**1.82**	**91.43%**

Table 2: Average CPI comparison between EvoPartition and human expert designs.

	SA	KL	METIS	Ours	Rate
Single-Function	2.09	2.07	2.42	**1.74**	**84.06%**
Multi-Function	2.07	2.10	2.44	**1.82**	**87.92%**
OpenCores	2.55	2.59	2.56	**2.10**	**82.63%**
RTLLM	2.75	2.75	2.81	**2.27**	**82.55%**

Table 3: Average CPI comparison across partitioning methods. Rate indicates the average CPI reduction relative to the best-performing baseline among the three compared methods.

These results demonstrate that, compared to manual partitioning based on expert knowledge, EvoPartition can more precisely identify and optimize pipeline stage differences, effectively avoiding redundant stall cycles and significantly reducing CPI, while ensuring functional correctness.

4.4 Comparison with Other Partitioning Algorithms

To further validate the effectiveness of our proposed method, we compare EvoPartition against three widely-used heuristic-based approaches: Simulated Annealing(SA), the Kernighan–Lin algorithm(KL), and the METIS framework. These methods are commonly adopted in the graph partitioning issue which is the fundamental abstraction underlying pipeline partitioning. Meanwhile, we also expanded our benchmark by incorporating additional circuits sampled from the OpenCores [OpenCores, 1999] and RTLLM [Lu *et al.*, 2024] benchmarks into our evaluation. Table 3 illustrates the average CPI for all results, using CPI as the evaluation metric. As shown in the Table 3, EvoPartition consistently outperforms the three baselines across all benchmarks. Compared to the strongest baseline among the three methods, EvoPartition achieves CPI reduction from 12.08% to 17.45% across four benchmarks. These results further demonstrate the significant advantage of EvoPartition in optimizing CPI, highlighting its superiority over conventional heuristic-based approaches.

5 Conclusions

In this paper, we argue that pipeline partitioning not only affects clock frequency but also has a significant impact on CPI through the pipeline stage differences. Based on this, we propose EvoPartition, a specialized evolutionary partitioning framework for automated pipeline design that guarantees functional correctness while reducing CPI and improving clock frequency. Experiments on multiple standard circuit benchmarks show that EvoPartition reduces the average number of stall cycles by 16.23% on average (to approximately 150), which in turn leads to an average 7.34% reduction in CPI, thereby increasing the pipeline throughput by 7.22% compared with the state-of-the-art.

Ethical Statement

LLMs were used to assist in improving the language quality and readability of this manuscript. All generated content was carefully reviewed and revised by the authors, who take full responsibility for the final manuscript.

Acknowledgments

This work was supported in part by the National Key R&D Program of China (2024YFE0212000), the National Natural Science Foundation of China (Grants No.62525203, 62341411, 62222214, 62502504, U22A2028), and the Strategic Priority Research Program of the Chinese Academy of Sciences (Grants No.XDB0660300, XDB0660301, XDB0660302). It was also supported by the CAS Project for Young Scientists in Basic Research (YSBR-029) and the Youth Innovation Promotion Association CAS.

References

[Aagaard and Leeser, 1994] Mark Aagaard and Miriam Leeser. Reasoning about pipelines with structural hazards. In *International Conference on Theorem Provers in Circuit Design*, pages 13–32. Springer, 1994.

[Amarú *et al.*, 2015] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The epfl combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

[Areibi, 2000] Shawki Areibi. An integrated genetic algorithm with dynamic hill climbing for vlsi circuit partitioning. In *GECCO 2000*, pages 97–102. Citeseer, 2000.

[Bagley, 1967] John Daniel Bagley. *The behavior of adaptive systems which employ genetic and correlation algorithms*. University of Michigan, 1967.

[Baruch *et al.*, 1999] ZOLTAN Baruch, O Cret, and KALMAN Pusztai. Genetic algorithm for circuit partitioning. In *Fifth Int. Conf. on Engineering of Modern Electric Systems: Section Computer Science and Control Systems*, pages 19–23, 1999.

[Campbell and Day, 2003] Jennifer PL Campbell and Nancy A Day. High-level optimization of pipeline design. In *Eighth IEEE International High-Level Design Validation and Test Workshop*, pages 43–48. IEEE, 2003.

[Cheng *et al.*, 2024] Shuyao Cheng, Chongxiao Li, Zidong Du, Rui Zhang, Xing Hu, Xiaqing Li, Guanglin Xu, Yuanbo Wen, and Qi Guo. Revisiting automatic pipelining: Gate-level forwarding and speculation. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

[Dong *et al.*, 2023] Zehao Dong, Weidong Cao, Muhan Zhang, Dacheng Tao, Yixin Chen, and Xuan Zhang. Cktgnn: Circuit graph neural network for electronic design automation. In *11th International Conference on Learning Representations, ICLR 2023*, 2023.

[Dubey and Flynn, 1990] Pradeep K Dubey and Michael J Flynn. Optimal pipelining. *Journal of Parallel and Distributed Computing*, 8(1):10–19, 1990.

[Fauth *et al.*, 1995] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nml. In *Proceedings the European Design and Test Conference. ED&TC 1995*, pages 503–507. IEEE, 1995.

[Fiduccia and Mattheyses, 1988] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pages 241–247. 1988.

[Ganusov *et al.*, 2016] Ilya Ganusov, Henri Fraisse, Aaron Ng, Rafael Trapani Possignolo, and Sabya Das. Automated extra pipeline analysis of applications mapped to xilinx ultrascale+ fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10. IEEE, 2016.

[Hansen *et al.*, 1999] Mark C Hansen, Hakan Yalcin, and John P Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[Hennessy and Patterson, 2011] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[Hu *et al.*, 2023] Yuting Hu, Jiajie Li, Florian Klemme, Gi-Joon Nam, Tengfei Ma, Hussam Amrouch, and Jinjun Xiong. Synctree: fast timing analysis for integrated circuit design through a physics-informed tree-based graph neural network. *Advances in neural information processing systems*, 36:21415–21428, 2023.

[Iştoan and De Dinechin, 2017] Matei Iştoan and Florent De Dinechin. Automating the pipeline of arithmetic datapaths. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 704–709. IEEE, 2017.

[Karypis and Kumar, 1998] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[Kernighan and Lin, 1970] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

[Kim *et al.*, 2015] Lok-Won Kim, Dong-U Lee, and John Villasenor. Automated iterative pipelining for asic design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(2):1–24, 2015.

[Kirkpatrick *et al.*, 1983] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[Krishnamurthy, 1984] Krishnamurthy. An improved min-cut algonthm for partitioning vlsi networks. *IEEE Transactions on computers*, 100(5):438–446, 1984.

[Kroening and Paul, 2001] Daniel Kroening and Wolfgang J Paul. Automated pipeline design. In *Proceedings of the 38th annual Design Automation Conference*, pages 810–815, 2001.

[Lee and Gerstlauer, 2017] Seogoo Lee and Andreas Gerstlauer. Data-dependent loop approximations for

performance-quality driven high-level synthesis. *IEEE Embedded Systems Letters*, 10(1):18–21, 2017.

[Liu *et al.*, 2019] Gai Liu, Joseph Primmer, and Zhiru Zhang. Rapid generation of high-quality risc-v processors from functional instruction set specifications. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[Lu *et al.*, 2024] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE, 2024.

[Majumder and Bondhugula, 2023] Kingshuk Majumder and Uday Bondhugula. Automatic multi-dimensional pipelining for high-level synthesis of dataflow accelerators. *arXiv preprint arXiv:2309.03203*, 2023.

[Marinescu and Rinard, 2001] Maria-Cristina V Marinescu and Martin Rinard. High-level automatic pipelining for sequential circuits. In *Proceedings of the 14th international symposium on Systems Synthesis*, pages 215–220, 2001.

[Nurvitadhi *et al.*, 2010] Eriko Nurvitadhi, James C Hoe, Shih-Lien L Lu, and Timothy Kam. Automatic multithreaded pipeline synthesis from transactional datapath specifications. In *Proceedings of the 47th Design Automation Conference*, pages 314–319, 2010.

[Nurvitadhi *et al.*, 2011] Eriko Nurvitadhi, James C Hoe, Timothy Kam, and Shih-Lien L Lu. Automatic pipelining from transactional datapath specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–454, 2011.

[OpenCores, 1999] OpenCores. Opencores: Open source ip cores. https://opencores.org/, 1999.

[Parrot *et al.*, 2021] Rémi Parrot, Mikaël Briday, and Olivier H Roux. Pipeline optimization using a cost extension of timed petri nets. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, pages 37–44. IEEE, 2021.

[Qi *et al.*, 2024] Penghui Qi, Xinyi Wan, Nyamdavaa Amar, and Min Lin. Pipeline parallelism with controllable memory. *Advances in Neural Information Processing Systems*, 37:46539–46566, 2024.

[Ramamoorthy and Li, 1977] Chittoor V Ramamoorthy and Hon Fung Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, 1977.

[Shahane *et al.*, 2023] Aditya Shahane, Saripilli Swapna Manjiri, Ankesh Jain, and Sandeep Kumar. Graph of circuits with gnn for exploring the optimal design space. *Advances in neural information processing systems*, 36:6014–6025, 2023.

[Shen and Lipasti, 2013] J.P. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.

[Soviani *et al.*, 2006] Cristian Soviani, Olivier Tardieu, and Stephen A Edwards. Optimizing sequential cycles through shannon decomposition and retiming. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 6–pp. IEEE, 2006.

[Wang *et al.*, 2024] Zhihai Wang, Jie Wang, Dongsheng Zuo, Ji Yunjie, Xilin Xia, Yuzhe Ma, Jianye Hao, Mingxuan Yuan, Yongdong Zhang, and Feng Wu. A hierarchical adaptive multi-task reinforcement learning framework for multiplier circuit design. In *Forty-first International Conference on Machine Learning*, 2024.

[Wang *et al.*, 2025] Zhihai Wang, Jie Wang, Xilin Xia, Dongsheng Zuo, Lei Chen, Yuzhe Ma, Jianye Hao, Mingxuan Yuan, and Feng Wu. Computing circuits optimization via model-based circuit genetic evolution. In *The Thirteenth International Conference on Learning Representations*, 2025.

[Wolf *et al.*, 2013] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.

[Zagieboylo *et al.*, 2022] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C Myers. Pdl: a high-level hardware design language for pipelined processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 719–732, 2022.

[Zeng *et al.*, 2022] Jun Zeng, Mingyang Kou, and Hailong Yao. Neuroschedule: A novel effective gnn-based scheduling method for high-level synthesis. *Advances in Neural Information Processing Systems*, 35:23792–23804, 2022.

[Zhong *et al.*, 2024] Ruizhe Zhong, Junjie Ye, Zhentao Tang, Shixiong Kai, Mingxuan Yuan, Jianye Hao, and Junchi Yan. Preroutgnn for timing prediction with order preserving partition: Global circuit pre-training, local delay learning and attentional cell modeling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17087–17095, 2024.

[Zou *et al.*, 2023] Jialv Zou, Xinggang Wang, Jiahao Guo, Wenyu Liu, Qian Zhang, and Chang Huang. Circuit as set of points. *Advances in Neural Information Processing Systems*, 36:32468–32480, 2023.