

HoPart: Hop-Constrained Partitioning with Routing Support for Multi-FPGA Systems

Yuan Huang, Longkun Guo*, Weijie Fang, and Jiawei Lin
 School of Mathematics and Statistics, Fuzhou University, Fujian, China
 lkguo@fzu.edu.cn

Abstract—Multi-FPGA platforms are indispensable for VLSI emulation and prototyping, but remain fundamentally constrained by limited inter-FPGA I/O bandwidth. Techniques such as time-division multiplexing and FPGA hopping partially alleviate this bottleneck but substantially increase partitioning and routing complexity and exacerbate timing closure. As modern FPGA-based applications impose stringent timing budgets, design flows must be explicitly delay-aware. In this paper, we present HoPart, a Hop-constrained partitioning approach that enforces per-path hop limits. A core ingredient of our approach is the joint optimization of path delay and congestion during partitioning. In addition, we propose a routing algorithm that adaptively adjusts the number of edges (hops) along each path based on real-time criticality metrics. This strategy reduces interconnect resource usage on non-critical paths while minimizing delay on timing-critical ones. Extensive experiments on public benchmark suites demonstrate that HoPart reduces maximum path delay by up to 30% compared with the state-of-the-art MaPart, while maintaining efficient utilization of inter-FPGA interconnect.

Index Terms—Partitioning, multi-FPGA system, timing-driven routing

I. INTRODUCTION

Multi-FPGA systems are essential for logic emulation and rapid prototyping of large-scale designs, providing flexibility and cost-efficiency [1]. They consist of multiple FPGAs interconnected via physical wires or programmable networks. Partitioning divides large circuits into smaller subcircuits assignable to individual FPGAs, which is critical for performance and delay optimization as designs grow in size and complexity [2].

As shown in Fig. 1, FPGAs are interconnected through physical wiring or programmable networks [3]. Limited I/O resources restrict interconnections, and time-division multiplexing (TDM) increases capacity [4] but adds delay. Without full interconnectivity, signals must pass through intermediate FPGA nodes, leading to longer paths and higher delays. Therefore, the routing of source-to-sink signal paths plays a critical role in determining system latency and timing closure, especially under constrained inter-FPGA connectivity. The FPGA interconnect topology is essential to optimize the system's frequency in multi-FPGA systems [5].

Partitioning for multi-FPGA systems faces challenges, as signal delay depends on hop count and TDM ratio after system-level routing [6]. A joint partitioning and routing approach was

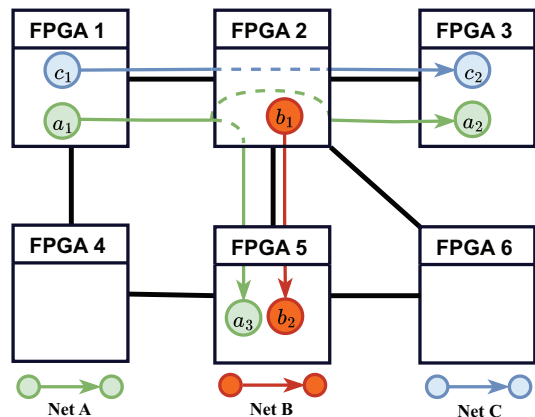


Fig. 1: Example of partitioning and routing in a multi-FPGA system with six FPGAs and seven bidirectional physical wires.

introduced to improve integration, but ignoring the maximum path length may degrade timing performance [7]. Later, a constraint-sensitive algorithm was introduced to minimize interconnects requiring hops under multiple design constraints [8]. TopoPart [9] and TopoPart+ [10] aim to avoid hops during partitioning, but achieving non-hop solutions in large circuits remains difficult. The optimization of path delay was also explored by adjusting net weights within the hmetis framework, effectively reducing the maximum path delay [11]. SPARK [12] proposed an adaptive FPGA interconnection scheme considering maximum TDM and hop counts. However, both [11] and [9] overlook FPGA interconnections during partitioning and develop approaches relying on system-level refinement with limited performance improvements.

MaPart [13] identifies that the maximum hop count significantly impacts the maximum delay and proposes a partitioning framework that minimizes the maximum hop count. However, when addressing hop-constrained partitioning, MaPart primarily relies on the assignment stage to satisfy hop constraints and lacks a systematic approach for managing these constraints during the refinement phase. TritonPart [14] and EasyPart [15] encounter similar limitations.

Cutting net routing also affects system performance. The 2019 ICCAD competition introduced the system-level FPGA routing problem using TDM technology [16], leading to key academic advances. System-level multi-FPGA systems routing algorithms have been proposed in [17]–[19] to reduce the maximum TDM ratio. PathFinder [20] utilizes edge criticality and congestion information to ensure that timing-critical paths

This work is supported by National Natural Science Foundation of China (No. 12271098), Guangzhou Leading Science and Technology Talent Program (No. 2025A04J7076), and Key Project of the Natural Science Foundation of Fujian Province (No. 2025J02011).

*Corresponding author.

utilize core resources, while non-critical paths bypass them, thereby partially optimizing system delay. However, excessive detouring may increase delays on non-critical paths.

To overcome these limitations, we propose HoPart, a partitioning method with routing support that simultaneously enforces resource and hop constraints during partitioning, jointly optimizes path delays and congestion, and incorporates a congestion-aware routing algorithm to reduce the delay. The contributions of this paper can be summarized as follows:

- Propose a partitioning approach for multi-FPGA systems that satisfies both maximum hop count and resource constraints. This approach novelly removes violating nodes from the initial partition and uses an alternating violation-correction method to reassign them. Additionally, it adopts a timing-driven refinement process to enhance partition quality and minimize maximum delay in subsequent routing stages.
- Design a routing algorithm that dynamically evaluates the level of criticality of paths and adaptively adjusts the number of edges on each path. This approach efficiently minimizes interconnect resource usage for non-critical paths while reducing delay on timing-critical paths.
- Extensive experiments were carried out to demonstrate that HoPart reduces maximum path delay by 30% and runtime by 59% compared to the state-of-the-art MaPart. Moreover, the violation correction mechanism in HoPart eliminates 95.79% of violating nodes.

The rest of this paper is organized as follows. Sec. II introduces the relevant terminology and problem formulation. Sec. III presents a detailed explanation of partitioning. Sec. IV describes the detailed procedure of congestion-aware routing. Sec. V discusses the benchmarks and experimental results. Finally, Sec. VI concludes the paper.

II. PRELIMINARIES

In this section, we formulate the problem of minimizing the maximum delay for multi-FPGA systems and outline the workflow of our work.

A. Problem Formulation

Given a circuit graph $H(V, E)$, an FPGA graph $G(\hat{V}, \hat{E})$, the algorithm must complete two tasks:

- 1) Determine the FPGA to which each node is assigned.
- 2) Route each net following the edges in \hat{E} .

In this paper, the assigned FPGA of a node $v \in V$ is denoted as $\hat{v} \in \hat{V}$. As in [13], the delay of the path $P(u, v)$ is calculated as follows:

$$\text{delay}^p(u, v) = \sum_{\hat{e}(\hat{u}, \hat{v}) \in P(u, v)} \text{tdm}(\hat{u}, \hat{v}). \quad (1)$$

The delay of net e is defined as follows:

$$\text{delay}^e(e) = \max\{\text{delay}^p(s(e), v) \mid v \in d(e)\}, \quad (2)$$

where $s(e)$ is the source node of net e , $d(e)$ denotes the set of sink nodes of net e . The constraints of the problem are as follows:

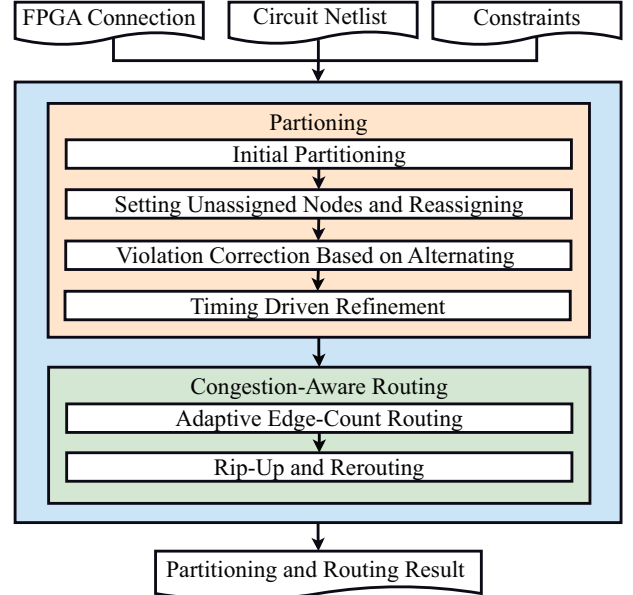


Fig. 2: Overall flow of HoPart

- **Resource constraint:** Each node consumes resources, and after partitioning, the total consumed resources on each FPGA must not exceed its limit.
- **Maximum hop count constraint:** As the maximum delay is positively correlated with the maximum hop count MH [13], each path is required to have a hop count no greater than a predefined upper bound.

Let R_i denote the resource of FPGA i , r^v represent the resource of node v , and F_i is the set of nodes assigned in FPGA i . The objective function can be formulated as follows:

$$\begin{aligned} \min \quad & f = \max\{\text{delay}^e(e_i) \mid e_i \in E\} \\ \text{s.t.} \quad & \sum_{v \in F_i} r^v \leq R_i, i = 1, 2, \dots, |\hat{V}|, \\ & \max\{\text{hop}(e) \mid \forall e \in E\} \leq MH. \end{aligned} \quad (3)$$

The hop count $\text{hop}(e)$ is defined as:

$$\text{hop}(e) = \max\{\text{dis}(s(\hat{e}), \hat{u}) - 1 \mid \forall u \in d(e)\}, \quad (4)$$

where $\text{dis}(s(\hat{e}), \hat{u})$ denotes the distance from $s(\hat{e})$ to \hat{u} in the graph $G(\hat{V}, \hat{E})$, with $s(\hat{e})$ and \hat{u} representing the FPGAs to which $s(e)$ and u are mapped, respectively.

B. Overall Flow

Given the input circuit netlist, FPGA connections, and constraints, the flow of our method is shown in Fig. 2, which consists of two main stages:

- **Partitioning:** The initial partitioning is performed using an open-source partitioner (In this paper, we use KaHyPar¹). Subsequently, nodes are gradually removed until the partitioning is free of violations. For the removed nodes, a non-hop assignment is first attempted. If some nodes remain unassigned, a violation correction method is applied iteratively until all nodes are successfully assigned.

¹<https://github.com/kahypar/mt-kahypar>

Finally, a timing-driven refinement is applied to drive the nodes of each cut net closer together, alleviating congestion in multi-FPGA systems.

- **Congestion-Aware Routing:** Before routing, the source-to-sink pairs are first sorted, and the number of FPGAs from source to sink is adaptively adjusted based on the criticality of each path. After all paths are routed, a rip-up and rerouting process is applied iteratively until the maximum delay no longer changes.

III. HOP AND RESOURCE-CONSTRAINED PARTITIONING

This section introduces our multi-FPGA partitioning method, which efficiently handles hop and resource constraints through node clustering, greedy mapping, iterative violation correction, and timing-driven refinement.

A. Initial Partitioning

Unlike MaPart and TopoPart, which utilize a propagation algorithm to determine the candidate FPGA set for all unassigned nodes based on each assigned node and subsequently seek the optimal assignment for the unassigned nodes within the candidate FPGA set, our approach first partitions the circuit graph into a specified number of regions before considering the impact of the maximum-hop constraint. In the initial partitioning stage, our goal is to cluster highly correlated nodes within the same partition, a process that can be efficiently achieved using existing open-source partitioners.

However, these open-source partitioners only ensure high intra-partition connectivity and fail to capture inter-partition connectivity effectively. Therefore, after obtaining the initial partitions, we further map them onto multi-FPGA systems. First, we randomly map each partition to a unique FPGA and then apply a greedy strategy to swap the mappings in pairs, minimizing *cost*. The *cost* is a measure of the tightness of the cut nets, and it is calculated as follows:

$$cost = \sum_{e \in E} w_e \sum_{v \in d(e)} \hat{dis}(s(\hat{e}), \hat{v}), \quad (5)$$

where $\hat{dis}(s(\hat{e}), \hat{v})$ is the distance from $s(\hat{e})$ to \hat{v} on $G(\hat{V}, \hat{E})$.

B. Marking Unassigned Nodes and Reassigning

After completing the initial partitioning, we first identify the set of FPGAs with resource overflows. For any resource-over-utilized FPGA i , we first sort its nodes in descending order based on their *effort* values, which are computed as follows:

$$effort(u) = c_i r^u, \forall u \in F_i, \quad (6)$$

c_i represents the criticality of the resource in FPGA i and F_i denotes the set of nodes assigned to FPGA i .

$$c_i = \left(\sum_{v \in F_i} r^v - R_i \right) / R_i. \quad (7)$$

Each node $u \in F_i$ is examined: if it can be moved to another FPGA without increasing the cut, it is moved; otherwise, it is removed and marked as unassigned until the resource usage of FPGA i satisfies the prescribed limits. For any source-to-sink pair whose hop count exceeds the bound, the sink node is set

as unassigned. This procedure guarantees that all assignments remain feasible under the imposed constraints.

Next, the nodes marked as unassigned are reassigned. To ensure that the hop count of all nets does not exceed MH , MaPart constructs the graph $G^*(V^*, E^*)$ based on MH and performs non-hop partitioning. Specifically, let $V^* = \hat{V}$, and for a given MH , add an edge $(u, v) \in E^*$ if $dis(u, v) \leq MH$ for all $u, v \in V^*$. The resulting partitioning naturally satisfies the non-hop constraint, and when mapped back to the original multi-FPGA system, ensures that the hop count of all nets remains below MH . The remaining steps focus on mapping the non-hop-constrained partitioning of the circuit graph $H(V, E)$ onto the FPGA graph $G^*(V^*, E^*)$. The first step in this process is the candidate FPGA propagation algorithm, which determines the candidate FPGA for each unassigned node based on the current assignments.

In TopoPart's candidate FPGA propagation algorithm, each propagation of an assigned node updates the candidate FPGAs for many unassigned nodes. However, as the number of assigned nodes increases, this process may lead to excessive runtime or memory overflow. To address this, MaPart points out that when using BFS to construct the influence set of assigned nodes, nodes that have already been processed during propagation do not need to be added to the BFS queue again. By ignoring these nodes, the resulting set remains equivalent to the original one, and this optimization becomes particularly effective when many nodes are assigned.

After computing the candidate FPGAs for all unassigned nodes, we employ a greedy strategy to assign them as proposed in [9], in order to assign each unassigned node to the FPGA within its candidate FPGA set that minimizes the cut.

C. Violation Correction Based on Alternating

After the procedures in Sec. III-B, the nodes that fail to be assigned due to maximum hop count or resource constraints are defined as violating nodes (CV_nodes).

When a node is moved, the candidate FPGA set from the propagation algorithm no longer represents the available FPGAs for unassigned nodes. To address this, we introduce the following concept. For an assigned node v , its movable FPGA set $M(v)$ is the intersection of the FPGAs used by its neighbors and their adjacent FPGAs in the inter-FPGA graph $G^*(V^*, E^*)$. For a violating node u , its sensitive FPGA set $SF(u)$ is defined similarly using its assigned neighbors and their adjacent FPGAs in $G^*(V^*, E^*)$. For each FPGA f , the set of sensitive nodes $SN(f)$ includes all the nodes that violate the rules that can be legally assigned in f . Specifically, if $u \in CV_nodes$ and $f \in SF(u)$, then $u \in SN(f)$.

To compute the gain of assigned node $v \in V_\lambda$ from FPGA f_i to each movable FPGA f_j , as follows:

$$Gain^v(f_i, f_j) = \Delta hop^v(f_i, f_j) + (R_i - \sum_{u \in F_i} r^u), \quad (8)$$

Algorithm 1: Violation-Aware Node Relocation (VANR)

Input: assigned nodes V_γ , violating nodes CV_nodes
Output: Partitioning solution P

- 1 Compute (SF) and (SN) ;
- 2 Sort V_γ and SN ;
- 3 **for** each node $v \in V_\gamma$ **do**
- 4 Update $M(v)$;
- 5 $f^* \leftarrow \arg \max_{f \in M(v)} \Delta hop^v(\hat{v}, f)$;
- 6 **if** $\Delta hop^v(\hat{v}, f^*) > 0$ **or** $|SN(\hat{v})| > 0$ **then**
- 7 Move node v from FPGA \hat{v} to FPGA f^* ;
- 8 **if** $|SN(\hat{v})| > 0$ **then**
- 9 **for** each node $u \in SN(\hat{v})$ **do**
- 10 **if** u could be assigned onto \hat{v} **then**
- 11 Update V_γ and CV_nodes ;
- 12 Update SF and SN ;
- 13 **for** each node $v \in CV_nodes$ **do**
- 14 Update $SF(v)$;
- 15 $f^* \leftarrow \arg \min_{f \in SF(v)} hop^v(f)$;
- 16 **if** node v could assign onto FPGA f^* **then**
- 17 Visit the next violating node;
- 18 **for** each node u assigned onto FPGA f^* **do**
- 19 Update $M(u)$;
- 20 $f' \leftarrow \arg \max_{f \in M(u)} Gain^u(f^*, f)$;
- 21 **if** u can be moved from f^* to f' **then**
- 22 Move node u ;
- 23 **if** v is successfully assigned onto f^* **then**
- 24 Update V_γ and CV_nodes ;
- 25 Visit the next violating node.

where $hop^v(f_i)$ represents the hop count of node v assigned to FPGA f_i , defined as:

$$hop^v(f_i) = \sum_{e \in S(v)} w_e \sum_{u \in d(e)} \hat{dis}(f_i, \hat{u}) + \sum_{e \in D(v)} w_e \times \hat{dis}(s(\hat{e}), f_i), \quad (9)$$

where $S(v)$ and $D(v)$ denote the sets of nets whose source node is v and sink node is v .

Algorithm 1 employs an alternating strategy to move nodes and correct violations, proceeding in two phases, each dedicated to a specific node set.

In the first phase, for each assigned node $v \in V_\lambda$, the FPGA f^* in its movable set $M(v)$ with the maximum gain is selected. If $\Delta hop^v(\hat{v}, f) > 0$ or if FPGA \hat{v} has sensitive nodes, v is moved to FPGA f^* . As node v is relocated to a different FPGA, its original FPGA releases resources that may accommodate violating nodes. The algorithm subsequently attempts to reassign the sensitive violating nodes previously mapped to that FPGA.

An ideal partitioning solution should encourage nodes in peripheral FPGA regions to move toward the core FPGA region. However, as high-gain moves continue, core FPGA resources become constrained, leading to increased competition for critical resources needed by unassigned nodes, making assignment harder. As shown in Fig. 3, the violating node

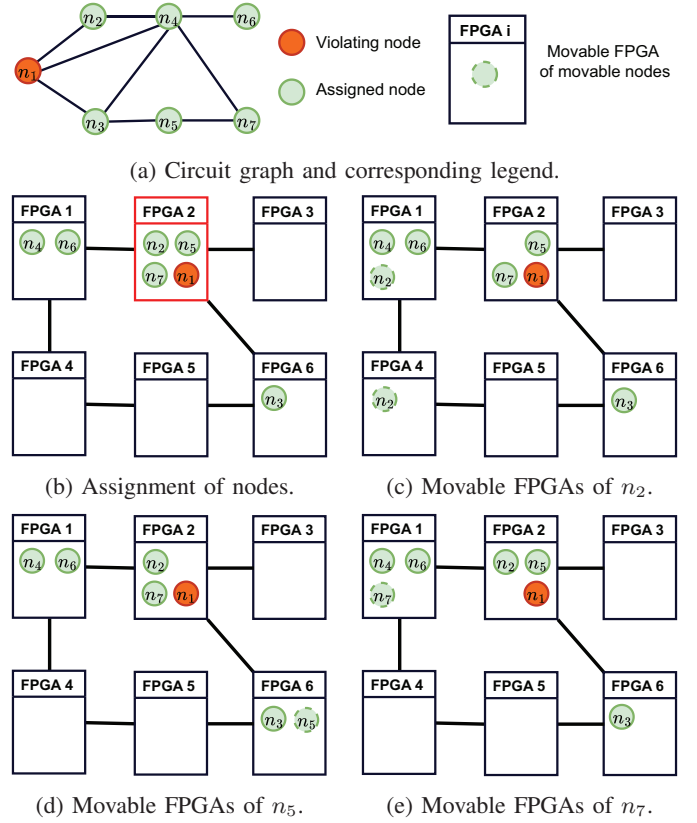


Fig. 3: An example of violating node assignment.

n_1 can only be assigned to FPGA 2 without violating the hop constraint, but resource congestion prevents assignment. Moving some nodes in FPGA 2 to less congested FPGAs can free up resources, allowing a successful assignment of n_1 .

The second phase addresses the aforementioned issue by focusing on FPGA nodes within the core region. For each violating node $v \in CV_nodes$, the algorithm selects the FPGA f^* with the minimal hop from its sensitive FPGA set $SF(v)$. Subsequently, the algorithm attempts to assign v onto FPGA f^* . If direct assignment fails, it iteratively relocates nodes from FPGA f^* to their respective highest-gain FPGA, thereby freeing up resources in FPGA f^* to assign the violating node v . When relocating nodes from resource-constrained FPGAs, repeatedly traversing from the beginning for each violating node is highly inefficient. To address this, we adopt a memory pointer mechanism: for an FPGA f , if the previous sensitive violating node accessed up to the k^{th} node, the next sensitive violating node can begin its search from the $(k + 1)^{th}$ node accordingly.

D. Timing-Driven Refinement

For multi-FPGA system partitioning, minimizing the maximum delay presents greater complexity than traditional objectives, such as minimizing cut size, because the maximum delay cannot be explicitly formulated during the partitioning phase. Consequently, the objective is to determine a partitioning solution that inherently facilitates routing optimization aimed at reducing the maximum delay.

Minimizing the number of cut nets is justified when considering delays arising solely from inter-FPGA connections. For any net, spanning fewer FPGAs from source to sink reduces its delay. Furthermore, because maximum delay is often governed by local congestion, an even distribution of cut nets can substantially alleviate the effects of such congestion.

During the partitioning refinement phase, maintaining solution quality while ensuring an even distribution of nets across the multi-FPGA systems becomes a critical challenge. To address this, we propose a gain function inspired by the FM [21] algorithm, which evaluates the gain of moving node u from FPGA f_i to FPGA f_j as follows:

$$\text{gain}^u(f_i, f_j) = \Delta\text{hop} + \alpha \times (\text{CUT}(i, j) - \overline{\text{CUT}}), \quad (10)$$

where CUT represents the distance matrix of the multi-FPGA system, with $\text{CUT}(i, j)$ representing the total weight of nets cut between FPGA i and FPGA j , computed as follows:

$$\text{CUT}(i, j) = \sum_{e \in E} w_e \times \left(\left(\sum_{u \in N(e)} \mathbf{1}\{u \in F_i\} \right) > 0 \right) \wedge \left(\sum_{u \in N(e)} \mathbf{1}\{u \in F_j\} \right) > 0 \quad (11)$$

Furthermore, we define $\overline{\text{CUT}}$ as the average value of all elements of CUT . After each move operation, the CUT matrix is updated accordingly. As described in Sec. III-C, during the refinement process, assigning node u to any FPGA $f \in M(u)$ guarantees the avoidance of hop violations. Thus, the optimal movement can be found within $M(u)$, improving computational efficiency by not considering all FPGAs. According to practice, we set the values of α to **0.01**, respectively.

IV. CONGESTION-AWARE ROUTING

In this section, we further introduce an adaptive edge-count routing scheme based on [20], which dynamically updates edge weights and efficiently determines shortest-delay paths with edge-count limitations.

A. Adaptive Edge-Count Routing

Negotiated routing proceeds over multiple iterations, during which critical factors on the path and congestion coefficients are updated to adjust edge weights, with routing repeated until convergence. As routing progresses, the number of signal transmissions between FPGAs increases, affecting the TDM ratio and signal delay. In this work, edge weights are initialized at the start of each iteration and updated in real time. In the first iteration, for each source-to-sink pair (s, t) , only the path with the fewest edges is selected. Upon completion, the delay of each source-to-sink path $\text{delay}^P(s, t)$ and the global maximum delay MD are recorded.

From the second routing iteration onward, the results of previous iterations are cleared. After each net is routed, the edge weights of every path $P(s_i, t_j)$ are updated according to the Pathfinder [20] rule:

$$w_{\hat{e}} = A_{ij} \text{tdm}_{\hat{e}} + (1 - A_{ij})(\text{tdm}_{\hat{e}} + h_{\hat{e}})p_{\hat{e}}, \forall \hat{e} \in P(s_i, t_j), \quad (12)$$

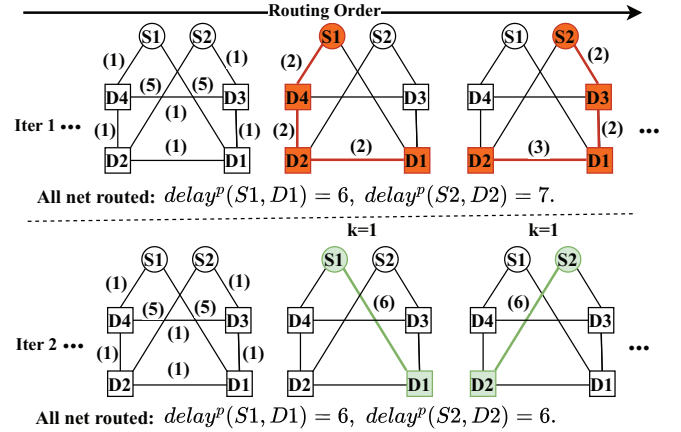


Fig. 4: Routing results with edge-count limitations.

where $\text{tdm}_{\hat{e}}$ denotes the TDM of edge \hat{e} , and $A_{ij} = \frac{D_{ij}}{MD}$ is the critical factor of path $P(s_i, t_j)$, with D_{ij} representing the maximum delay of all paths traversing $P(s_i, t_j)$. The parameters $h_{\hat{e}}$ and $p_{\hat{e}}$ denote the historical and present congestion of edge \hat{e} , respectively, as in [22]. This update strategy prioritizes critical paths on core resources while steering non-critical paths away. Nonetheless, due to the routing order, high-priority signals may not fully account for the resource usage of subsequent paths, potentially causing detours and increased delays, as shown in Fig. 4. In the first iteration, routing minimizes edge weights along the shortest paths. However, overuse of certain edges can increase the delays of previously routed paths. The second iteration enforces edge-count constraints for each path, thereby mitigating the risk of detours and delay increases.

Therefore, when routing each source-to-sink pair (s, t) , we use a more sophisticated method rather than the simple shortest-path routing. Specifically, for each source-to-sink routing, we compute the ideal edge count for the path as follows:

$$k = \left\lceil \frac{\text{delay}^P(s, t) \times (MH + 1)}{MD} \right\rceil. \quad (13)$$

By transforming the shortest-path problem into a shortest-path problem with edge-count limitations and constructing a k -layer graph based on the FPGA topology, the shortest path can then be found within the multi-layer graph using Dijkstra's algorithm. If no path exists in the k -layer graph, k is incrementally increased until a feasible path is identified. This strategy ensures that the number of edges of each path does not exceed $MH + 1$. By constructing a graph with at most $MH + 1$ layers, repeated multi-layer graph construction is avoided. Moreover, since the FPGA graph $G(\hat{V}, \hat{E})$ is much smaller than the circuit graph $H(V, E)$, it does not incur significant memory overhead to directly construct an $(MH + 1)$ -layer graph.

B. Net Sorting and Rerouting

The routing order significantly influences overall performance [23]–[25]. Later-routed nets can avoid congested edges, whereas earlier-routed nets lack this information and are considered less critical. Nets with higher k values are assigned greater priority, when multiple nets share the same k , those with larger maximum hops along paths are given lower priority.

TABLE I: Comparison of MaPart and HoPart across public benchmarks, where **C**, **D**, and **R** denote the cut size after partitioning, the maximum delay after routing, and the overall runtime, respectively.

Benchmark	MaPart [13]			HoPart		
	C	D	R	C	D	R
sparcT1_core	39312	24	283	9703	15	25
neuron	15959	10	112	5769	7	46
stereo_vision	16515	11	110	5582	8	29
des90	27060	18	315	7927	10	142
SLAM_spheric	37498	21	258	13309	14	97
cholesky_mc	24314	15	213	9503	10	50
segmentation	26569	22	101	7307	11	78
bitonic_mesh	28066	20	357	10900	11	383
dart	29655	21	341	10885	15	180
openCV	29994	21	304	7073	12	108
stap_qrd	37375	21	352	11652	13	117
minres	27589	20	346	9189	17	92
cholesky_bdti	33442	20	468	15960	17	403
denoise	25786	16	598	8246	15	107
sparcT2_core	71213	43	834	17849	34	155
gsm_switch	128720	65	1393	25153	29	586
mes_noc	33867	22	841	11717	14	420
LU_Network	71573	42	1362	26569	24	572
LU230	66027	39	709	30096	25	647
sparcT1_chip2	87099	48	2013	27452	30	552
directrf	54676	33	1712	28316	32	1034
bitcoin_miner	118512	86	1383	61237	47	350
case1	472900	268	1791	229184	131	819
case2	673009	329	2653	389463	220	441
case3	825130	418	3405	525397	330	576
case4	974586	485	4904	667462	376	859
case5	1253899	598	5003	794696	455	1527
case6	1340649	656	4999	917004	530	2595
case7	1250922	633	6713	1032416	608	2869
case8	1545967	791	7607	1176862	711	3107
Avg.ratio	1.00	1.00	1.00	0.43	0.70	0.41

The rip-up and rerouting process plays a critical role in refining net routing after the initial routing stage is completed [26]–[28]. Specifically, the algorithm targets the path $P(u, v)$ with the maximum delay and removes any paths that satisfy the following conditions:

- $P(u, v) \cap P(s, t) \neq \emptyset, \forall s \in F_{\hat{u}}, \forall t \in V$;
- $P(u, v) \cap P(s, t) \neq \emptyset, \forall s \in V, \forall t \in F_{\hat{v}}$;
- $P(u, v) \cap P(s, t) \neq \emptyset, \forall s \in V \cap F_{\hat{u}}, \forall t \in V \cap F_{\hat{v}}$.

After their removal, these paths are rerouted to reduce delay for improving overall performance. This rip-up and rerouting process is repeated for iterations until the delay values of all edges are convergent (i.e., no large change during one round). Once the rerouting process terminates, the best solution among all is selected as the final solution. Finally, the congestion-aware routing is considered complete when all nodes sucare cessfully connected through the edges in $G(\hat{V}, \hat{E})$.

V. EXPERIMENTAL RESULTS

We implemented our algorithm in C++ and compiled it using g++ 13.2.0. All experiments were conducted on a Linux workstation with an AMD Ryzen 5700 3.4 GHz CPU.

A. Experimental Results Compared with MaPart

For a fair comparison, HoPart employs the same benchmarks and conditions as in [13]. Since MaPart aims to minimize

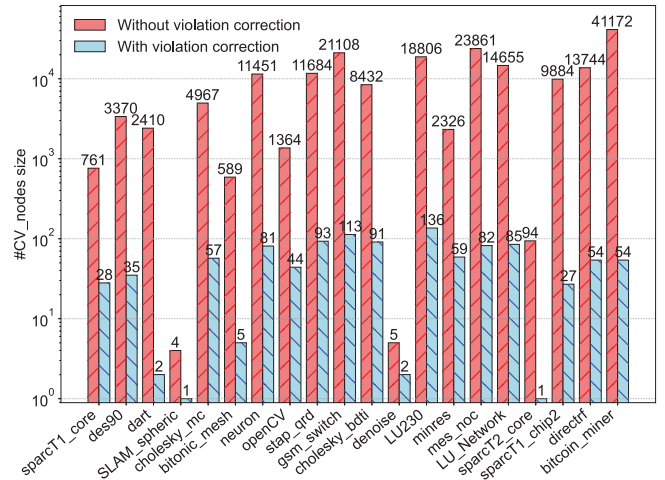


Fig. 5: Size comparison of CV_nodes w/o violation correction.

the maximum hop, we set the MH in our experiments to match the “max-hop” of MaPart to ensure a fair comparison. The experimental results of MaPart are directly taken from its original publication.

Table I presents a comparative analysis between MaPart and our proposed algorithm in terms of cut size, maximum delay, and runtime in all test cases. The maximum delay in the system is the key metric after partitioning and routing. The results demonstrate that HoPart achieves a **30%** reduction in maximum delay compared to the MaPart method. Moreover, the cut size is only 43% of that obtained by MaPart.

B. Violating Nodes Reduction with HoPart

When the balance factor is set to 0.2 as [13], neither TopoPart nor TopoPart+ can achieve a partitioning that satisfies the non-hop constraint on Titan23. To evaluate the effectiveness of HoPart in enforcing the hop count constraint, we analyze the variation in the number of constraint-violating nodes. With MH set to 1, we compare the number of violating nodes before and after applying the violation correction method. The experimental results, visualized in the form of a histogram in Fig. 5, clearly illustrate this difference. Specifically, the “stereo_vision” and “segmentation” benchmarks successfully eliminate all nodes violating the maximum hop count constraint. HoPart reduces the number of violating nodes by 95.79%, significantly improving compliance with the maximum-hop-count constraint.

VI. CONCLUSION

In this paper, we propose HoPart, which integrates a timing-driven refinement with a violation-correction mechanism to enforce predefined per-path hop limits while optimizing path delay. To complement HoPart, we further propose a congestion-aware routing algorithm that dynamically adjusts the hop count along each path, thereby reducing interconnect usage on non-critical paths and minimizing delay on critical ones. Experimental results demonstrate that HoPart achieves a 30% reduction in the maximum delay and a 59% reduction in runtime compared with the state-of-the-art MaPart baseline.

REFERENCES

- [1] U. Farooq, I. Baig, M. K. Bhatti, H. Mehrez, A. Kumar, and M. Gupta, "Prototyping using multi-fpga platform: A novel and complete flow," *Microprocessors and Microsystems*, vol. 96, p. 104751, 2023.
- [2] K. Wei, H. Amano, R. Niwase, Y. Yamaguchi, and T. Miyoshi, "Qu-trefoil: Large-scale quantum circuit simulator working on fpga with sata storages," *IEEE Transactions on Computers*, 2024.
- [3] W.-S. Kuo, S.-H. Zhang, W.-K. Mak, R. Sun, and Y. K. Leow, "Pin assignment optimization for multi-2.5 d fpga-based systems," *Proceedings of the 2018 International Symposium on Physical Design (ISPD)*, pp. 106–113, 2018.
- [4] P. Zou, Z. Lin, X. Shi, Y. Wu, J. Chen, J. Yu, and Y.-W. Chang, "Time-division multiplexing based system-level fpga routing for logic verification," *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.
- [5] U. Farooq, H. Mehrez, and M. K. Bhatti, "Inter-fpga interconnect topologies exploration for multi-fpga systems," *Design Automation for Embedded Systems*, vol. 22, no. 1, pp. 117–140, 2018.
- [6] W. N. Hung and R. Sun, "Challenges in large fpga-based logic emulation systems," *Proceedings of the 2018 International Symposium on Physical Design (ISPD)*, pp. 26–33, 2018.
- [7] M.-H. Chen, Y.-W. Chang, and J.-J. Wang, "Performance-driven simultaneous partitioning and routing for multi-fpga systems," *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1129–1134, 2021.
- [8] B. Li, Z. Qi, Z. Tang, X. He, and H. You, "High quality hypergraph partitioning for logic emulation," *Integration*, vol. 83, pp. 67–76, 2022.
- [9] D. Zheng, X. Zang, and M. D. Wong, "Topopart: a multi-level topology-driven partitioning framework for multi-fpga systems," *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–8, 2021.
- [10] B. Li, H. You, S. Bi, and Y. Zhang, "An efficient hypergraph partitioner under inter - block interconnection constraints," *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, 2024.
- [11] S.-H. Liou, S. Liu, R. Sun, and H.-M. Chen, "Timing driven partition for multi-fpga systems with tdm awareness," *Proceedings of the 2020 International Symposium on Physical Design (ISPD)*, pp. 111–118, 2020.
- [12] X. Zang, E. F. Young, and M. D. Wong, "Spark: A scalable partitioning and routing framework for multi-fpga systems," *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI)*, pp. 593–598, 2023.
- [13] B. Li, S. Bi, H. You, Z. Qi, G. Guo, R. Sun, and Y. Zhang, "Mapart: An efficient multi-fpga system-aware hypergraph partitioning framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, pp. 3212–3225, 2024.
- [14] I. Bustany, G. Gasparyan, A. B. Kahng, I. Koutis, B. Pramanik, and Z. Wang, "An open-source constraints-driven general partitioning multi-tool for vlsi physical design," *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, 2023.
- [15] S. Tong, H. Li, J. Xu, C. Pei, W. Yu, S. Liu, and J. Shen, "Easypart: An effective and comprehensive hypergraph partitioner for fpga-based emulation," *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2024.
- [16] Y.-H. Su, R. Sun, and P.-H. Ho, "2019 cad contest: System-level fpga routing with timing division multiplexing technique," *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–2, 2019.
- [17] P. Zou, Z. Lin, X. Shi, Y. Wu, J. Chen, J. Yu, and Y.-W. Chang, "Time-division multiplexing based system-level fpga routing for logic verification," *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.
- [18] T.-W. Lin, W.-C. Tai, Y.-C. Lin, and I. H.-R. Jiang, "Routing topology and time-division multiplexing co-optimization for multi-fpga systems," *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.
- [19] D. Zheng, X. Zhang, C.-W. Pui, and E. F. Young, "Multi-fpga co-optimization: Hybrid routing and competitive-based time division multiplexing assignment," *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 176–182, 2021.
- [20] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for fpgas," in *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays (FPGA)*, pp. 111–117, 1995.
- [21] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," *Papers on Twenty-five years of electronic design automation*, pp. 241–247, 1988.
- [22] Y. Zha and J. Li, "Revisiting pathfinder routing algorithm," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pp. 24–34, 2022.
- [23] W. Lin, H. Wu, P. Gao, W. Luo, S. Cai, and X. Xiong, "Sequential routing-based time-division multiplexing optimization for multi-fpga systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 28, no. 6, pp. 1–10, 2023.
- [24] W. Liu, Y. Xing, S. Cai, W. Li, and X. Xiong, "Optimizing fpga routing with explainable co-learning of congestion and wirelength," *ACM Transactions on Design Automation of Electronic Systems*, 2025.
- [25] J. Wang, J. Mai, Z. Di, and Y. Lin, "A robust fpga router with concurrent intra-clb rerouting," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 529–534, 2023.
- [26] U. Gandhi, E. Aghaeekiasaraee, Sahir, P. Mousavi, I. S. Bustany, M. E. Taylor, and L. Behjat, "Applying reinforcement learning to learn best net to rip and re-route in global routing," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 4, pp. 1–21, 2024.
- [27] J. Wang, X. Jiang, and Y. Lin, "Top-level routing for multiply-instantiated blocks with topology hashing," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2024.
- [28] F. Wang, L. Liu, J. Chen, J. Liu, X. Zang, and M. D. Wong, "Starfish: An efficient p&r co-optimization engine with a*-based partial rerouting," in *2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.