Potter: A Parallel Overlap-Tolerant Router for UltraScale FPGAs

Xinshi Zang
The Chinese University of Hong Kong
xszang@cse.cuhk.edu.hk

Wenhao Lin
The Chinese University of Hong Kong
whlin23@cse.cuhk.edu.hk

Jinwei Liu
The Chinese University of Hong Kong
jwliu@cse.cuhk.edu.hk

Evangeline F.Y. Young
The Chinese University of Hong Kong
fyyoung@cse.cuhk.edu.hk

ABSTRACT

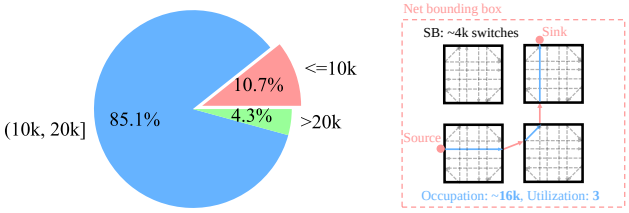
Routing is a time-consuming stage in FPGA compilation, and various parallel approaches have been proposed to accelerate it by concurrently routing non-overlapping nets. However, the requirement for non-overlapping nets limits the potential for large-scale parallelism, primarily due to two factors: (1) large circuits inherently contain many nets with overlapping bounding boxes, and (2) in modern FPGAs, such as Xilinx UltraScale FPGAs, a net with a large bounding box often has high occupancy but low utilization of the routing resources. To overcome these limitations, we present Potter, a novel parallel overlap-tolerant router designed to maximize parallelism. Our approach employs recursive partitioning to divide nets into balanced partitions with minimized overlap and allows for routing these partitions in parallel. Additionally, we propose an innovative mechanism for updating the congestion factors to enhance PathFinder in handling routing resource overflows. Evaluations on the FPGA 2024 contest benchmarks demonstrate that Potter achieves significant performance improvements, with average speedups of 12× and 8× compared to RWRoute and Vivado, respectively, while also reducing wire lengths by 4% and 45%. Notably, in some congested benchmarks, Potter exhibits a substantial 30× speedup over RWRoute.

ACM Reference Format:
Xinshi Zang, Wenhao Lin, Jinwei Liu, and Evangeline F.Y. Young. 2024. Potter: A Parallel Overlap-Tolerant Router for UltraScale FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3676536.3676783

1 INTRODUCTION

The prolonged compilation time in FPGA development has long been a concern in the FPGA community. Reducing the physical design time and accelerating the development cycle is crucial and urgent for promoting FPGA adoption and applications, particularly in the era of AI and GPUs.

Routing is a computationally intensive and time-consuming stage in the physical design of FPGAs. After logic modules are assigned physical locations on an FPGA, the routing process involves connecting logic nets using FPGA routing resources, such as wire segments and configurable switches. Due to the limited availability of routing resources and the significantly large number of logic nets in modern circuit designs, finding a valid routing solution that optimizes wire length or timing without causing resource overflow often requires many iterations of rip-up and reroute.



(a) Overlapping net number **(b) Resource utilization**

Figure 1: The number of overlapping nets in CoreScore 1700 in RWRoute [16] and a routing example in UltraScale FPGA.

To expedite FPGA routing, many works have been studying parallel routers for FPGAs [11], exploring both coarse-grain and fine-grain parallelism. Coarse-grain parallelism involves dividing nets into independent partitions that can be routed in parallel, while fine-grain routers parallelize the path searching for individual nets. These two forms of parallelism are orthogonal and complement each other. In this work, we focus on investigating coarse-grain net-level parallelism.

Most existing approaches [3, 9, 13, 14, 17] for net-level parallel routing employ recursive partitioning to divide nets into different partitions and perform concurrent routing for non-overlapping partitions. Although the requirement for non-overlapping nets can ensure an accurate congestion state for routing, it largely limits the potential for large-scale parallelism, especially in large circuit designs and modern FPGAs primarily for two reasons. Firstly, large circuits usually exhibit severe net overlapping. Take the CoreScore 1700, the largest circuit in FPGA 2024 contest benchmarks [4], as an example. As shown in Fig. 1a, nearly 89.4% nets overlap with more than 10,000 other nets under the default bounding box extension in RWRoute [16], an academic sequential router for Xilinx FPGAs. Secondly, the actual utilization of routing resources is significantly lower than the resources occupied by the net bounding box. As shown in Fig. 1b, each switch box (SB) in Xilinx UltraScale FPGA contains thousands of configurable switches while the routes of a single net only require a few switches.

To maximize routing parallelism, we propose a novel **p**arallel **o**verlap-**t**olerant rou**ter** called Potter, designed for UltraScale FPGAs. Potter takes the placement solution and the FPGA architecture in FPGA Interchange Format (FPGAIF) [5] as input and generates

This research was partially supported by ACCESS - AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10.
https://doi.org/10.1145/3676536.3676783

the routing solution in FPGAIF, which can be loaded into Vivado after conversion. We propose a two-stage overlap-driven partitioning by first utilizing recursive partitioning to divide nets into different partitions and then constructing overlap-tolerant net batches with balanced workloads and minimized resource overlap. Furthermore, we enhance the congestion updating mechanism in the negotiation-based routing algorithms by dynamically adjusting the present and historical congestion factors, enabling Potter to converge quickly to an overflow-free routing solution in congested designs. The main contributions of this work are as follows:

- We propose a novel parallel routing flow that can be seamlessly integrated into the physical design flow of the industrial tool Vivado.
- We explore multi-threaded overlap-tolerant routing and develop a two-stage overlap-driven partitioning approach to minimize resource overlap across different threads and balance the workloads in threads.
- We devise a novel congestion updating mechanism to enhance the negotiation-based routing algorithms in solving resource overflow.
- In the FPGA 2024 contest benchmarks, our proposed router, Potter, achieves significant improvements over the state-of-the-art methods in terms of both routing time and wire length of the critical path. In congested designs, Potter demonstrates a maximum speedup of 30× compared to RWRoute. On average, Potter achieves a speedup of 12× with a wire length reduction of 4% on RWRoute, and a speedup of 8× with a wire length reduction of 45% on Vivado.

The remainder of this paper is organized as follows. Sec. 2 provides an overview of related works on FPGA routing. Sec. 3 describes the architecture of UltraScale FPGAs. Our methodology is presented in detail in Sec. 4, and the experimental results are discussed in Sec. 5. Finally, we conclude this paper in Sec. 6.

2 RELATED WORKS

In this section, we present existing works on accelerating FPGA routing through algorithmic improvements in sequential routing and parallel routing.

2.1 Sequential Routing

Sequential routing algorithms based on the negotiation mechanism have been widely adopted and proven effective in existing FPGA routers [7, 8, 12, 16]. PathFinder [7] is the most popular sequential routing algorithm that uses the negotiation mechanism. In PathFinder, the routing cost of a node is defined as $(b+h)\times p$, where b is the base cost, and h and p are the historical and present congestion costs, respectively. The negotiation mechanism gradually increases h and p of congested nodes and encourages congested nets to utilize nodes with lower congestion costs. VTR 8 [8] improved the routing times of the high-fanout nets through adaptive incremental routing. CRoute[12] proposed a connection-based router to selectively route congested connections instead of the whole net and incorporated bias into the cost function. Based on CRoute, RWRoute[16], which is adopted in RapidWright [6], is a sequential router targeting UltraScale FPGA architecture. It provides a lightweight timing model and supports partial routing.

2.2 Parallel Routing

With the availability of a larger number of threads provided by modern CPUs, several parallel algorithms have been developed to accelerate the routing process in FPGAs [1–3, 10, 11, 13, 14, 17]. Recursive partitioning is a popular approach in the parallel routers [3, 10, 13, 14, 17], which effectively divides the nets into different partitions and routes non-overlapping partitions in parallel. In addition to concurrent routing for non-overlapping nets, Gort et al. [1] and ParaFRo [2] also studied the asynchronous routing for dependent nets which can overlap with each other. Gort et al. [1] evenly distributed all nets among individual threads without considering net dependencies and gradually reduced the active threads in later iterations to ensure convergence. Apart from the static scheduling strategy of Gort et al. [1], ParaFRo [2] also employed a greedy scheduling strategy that allows idle threads to fetch tasks from the pool of nets to be routed. Although these two approaches achieved significant speedup compared to other works, they did not handle net overlapping between different threads well, resulting in difficulties in routing convergence and a large variance in routing results.

Given the limitations of existing approaches, in this work, we propose a more efficient overlap-tolerant multi-threaded parallel routing based on a two-stage partitioning strategy which both minimizes the net overlaps and balances the workload in different threads. Furthermore, we devise novel dynamic congestion factors that enhance existing sequential routing approaches to converge quickly in congested designs.

3 PRELIMINARIES

In this section, we provide an overview of the UltraScale FPGA architecture and describe the routing resource graph.

3.1 Architecture of Ultrascale FPGA

Our study focuses on Xilinx UltraScale FPGAs. The architecture, as depicted in Fig. 2, consists of various resources, including configurable logic blocks (CLBs), block RAM (BRAM), digital signal processing (DSP) units, IO interfaces, and interconnect (INT) blocks. The INT blocks play a crucial role in connecting non-adjacent blocks in different positions and are regularly interleaved with other functional blocks. Each INT block contains thousands of programmable interconnect points (PIPs), which are configurable switches. Each PIP connects a pair of wires which can horizontally or vertically span 1, 2, 4, or 12 INT blocks.

3.2 Routing Resource Graph

The routing resource graph (RRG) is constructed from the INT block network and represents the wires and PIPs in the FPGA. Fig. 3 illustrates a section of the RRG built from one INT block in the FPGA, where each wire and PIP are treated as an RRG node and edge, respectively. UltraScale FPGAs have a large number of interconnect blocks and functional blocks to support the implementation of complex circuits. A complete RRG for UltraScale FPGAs contains over 28 million nodes and 100 million edges, presenting a significant challenge for path searching compared to RRGs in academic hypothetical architectures.

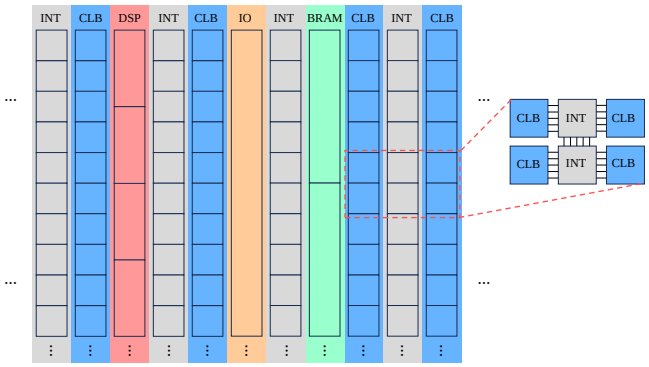


Figure 2: The architecture of UltraScale FPGA.

Each RRG node has a capacity of one, and multiple nets using the same RRG node results in resource overflow. The length of RRG nodes depends on the number of INT blocks the physical wires span, and the long nodes are usually insufficient resources compared with the short ones. After placement, all sources and sinks of logic nets in a circuit design are mapped to nodes on the RRG. The routing task involves finding the shortest paths for all nets without any resource overflow on the RRG.

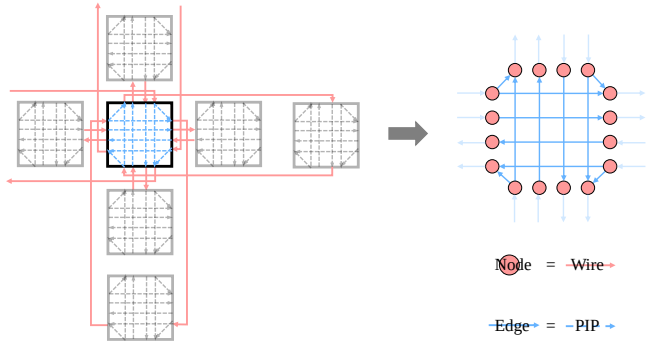


Figure 3: Example of the routing resource graph construction

4 METHODOLOGY

This section outlines the framework of our parallel overlap-tolerant router, as depicted in Fig. 4. We begin by generating a compact routing resource graph (RRG) in the pre-processing stage. The RRG is constructed based on the placed netlist and the FPGA architecture, excluding unnecessary wires and the programmable interconnect points (PIPs). To fully utilize the available CPU threads, we will apply a two-stage overlap-driven partitioning strategy to divide the nets into T workload-balanced and overlap-controlled batches, where T corresponds to the number of threads. In the subsequent main routing stage, we will assign the batches of nets to different threads and route them in parallel. After that, the present and historical congestion costs of the RRG nodes will be updated based on the dynamic congestion factors and resource overflow. This process is repeated until the number of RRG nodes with overflow falls below a threshold τ. In the post-processing stage, we will route the remaining unrouted or congested nets to ensure convergence and remove any redundant routing cycles.

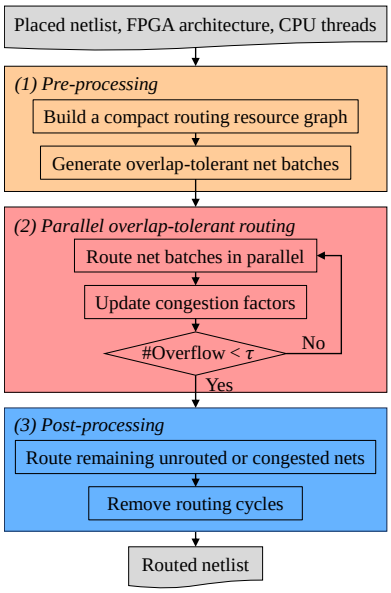


Figure 4: The overall flow of Potter.

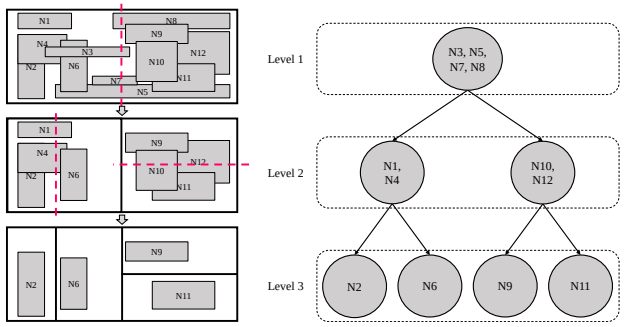
4.1 Compact Routing Resource Graph

As described in Sec. 3, the complete routing resource graph (RRG) of an UltraScale FPGA is very large and complex, resulting in substantial memory consumption and an extensive searching space for routing. To address this issue, RWRoute[16] adopts an incremental RRG construction approach. However, this method is not suitable for parallel routing, as multiple threads would modify the RRG concurrently. In Potter, we construct a compact routing resource graph initially by excluding unnecessary wires for the current netlist.

There are two types of wires necessary for routing: (1) the wires connecting INT blocks, which are the same for all netlists, and (2) the wires connecting INT blocks with the net pins, which differ across netlists. By including only these two types of wires and the PIPs between them, we build an initial RRG with nodes representing wires. Since there are nodes in the initial RRG that have no outgoing edges (not driving any other nodes) and are useless for routing, we will iteratively remove these nodes until all the remaining nodes in the RRG have at least one child node. This process results in a compact RRG that has a size nearly 30% less than a complete RRG.

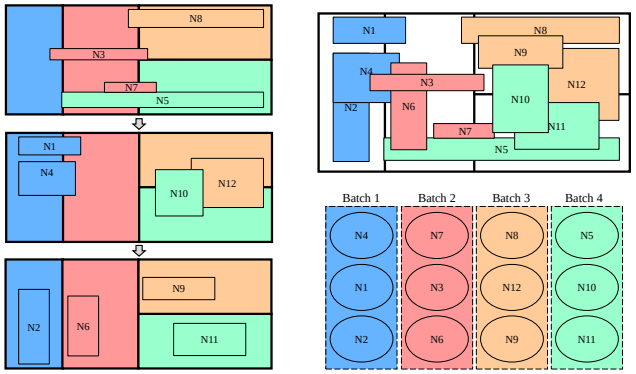
4.2 Partitioning-based Net Scheduling

To achieve high acceleration with multiple threads, it is significant to balance workloads and minimize the dependencies of different threads. We estimate the workload of routing a multi-pin net based on the number of pins, and the resource conflicts are defined as the overlaps of the net bounding boxes in different threads. In the following, we introduce our two-stage partitioning-based net scheduling approach to generate workload-balanced net batches while minimizing conflicts.



(a) Partitioning procedure (b) Binary tree

Figure 5: The first-stage recursive bi-partitioning with the thread number, $T = 4$.



(a) Partitioning procedure (b) Partitioning result

Figure 6: The second-stage overlap-tolerant partitioning

In the first stage, we perform recursive bi-partitioning to construct a complete binary tree. The FPGA device is initially split into two regions by a horizontal or vertical cutline. The nets are then divided into three partitions: the parent partition containing the nets across the cutline, and the two children partitions containing the nets completely within the two regions. An optimal cutline is chosen to minimize the imbalance of the total workloads in the children partitions. We continue this recursive partitioning on the two regions until the number of regions equals the number of threads T. A binary tree is then constructed with tree nodes representing net partitions. The tree nodes from top to bottom at the same tree level correspond to the net partitions generated at the same recursive level. Fig. 5 illustrates an example of this recursive partitioning and the binary tree construction with four threads.

After the recursive bi-partitioning, the FPGA device is split into T regions and nets are assigned to different partitions at different tree levels. Existing partitioning-based parallel routers [3, 14, 17] route the net partitions in a level-by-level manner to ensure no overlaps of net bounding boxes. However, this non-overlapping requirement significantly limits parallelism. To make full use of available threads, we introduce the second-stage overlap-tolerant partitioning. In this approach, nets assigned to each level are re-assigned to T regions such that the out-of-the-region areas of the net bounding boxes are minimized, and the total workload in each region is balanced. We set a maximum workload in each region at level l to $\frac{W_l}{T} \times (1 + r)$, where r is an imbalance ratio, and W_l is the total workload of the nets at level l. Each net at level l is then assigned to a region that still has available workload quota and has the maximum overlap with its bounding box. It is worth noting that there may exist a few high-fanout nets whose workload exceeds the maximum workload of a region, and will result in a significant imbalance in net partitioning. Therefore, we will route those high-fanout nets separately in the post-processing stage. After the overlap-tolerant partitioning is completed in each level, the nets assigned to the same region are grouped into a net batch. Fig. 6 depicts the procedure and the result of this overlap-tolerant partitioning for the example in Fig. 5.

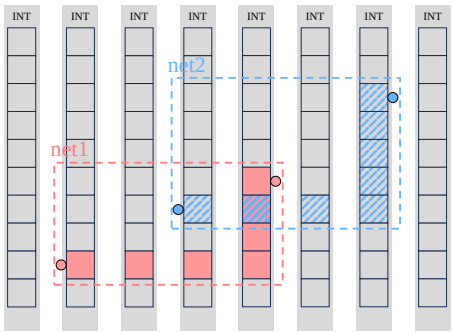


Figure 7: Example of parallel overlap-tolerant routing

4.3 Parallel Overlap-Tolerant Routing

The parallel overlap-tolerant routing is the core component of Potter and contributes the most to the routing time. In this stage, each available CPU thread is responsible for routing a batch of nets. In Potter, each RRG node is assigned a lock to solve the data race issue that arises from multiple threads accessing the data of the node. Fig. 7 illustrates an example of the parallel routing for two nets whose bounding boxes are overlapped. Although these two nets access the same INT block in their routes, the possibility that both net routes use the same RRG node is very small. Even though the same RRG node is used, the resource overflow can be resolved in the later iterations with the increase of the congestion cost on this node.

Following the negotiation mechanism, Potter performs multiple iterations of rip-up and reroute. In each iteration, the net batches are routed in parallel, followed by congestion factor update. For each net in a batch, Potter performs sequential connection-based routing. In the following, we will describe the routing engine and congestion factor update in Potter.

4.3.1 Connection-based routing engine. Similar to RWRoute [16], Potter also adopts connection-based routing as the core engine. The multi-pin nets in a net batch are split into source-sink two-pin nets, referred to as connections. These connections are routed independently but are encouraged to reuse the RRG nodes already used by other connections from the same net. When a multi-pin net is routed through a congested region, Potter only needs to rip

up the congested connections instead of all the connections from the net, which remarkably saves routing time.

Routing a connection involves finding the shortest path that connects the source and sink nodes on the RRG. Potter adopts the path searching algorithm used in RWRoute, with the node cost function defined as follows:

$$f(n) = \sum_{i \in P} c(i) + c(n) + w_1 \times d(n) \tag{1}$$

In Eq. (1), P presents the partial **p**ath from the source node to node n, and $d(n)$ is the Manhattan **d**istance from n to the sink node. $c(n)$ represents the node **c**ore cost, which is related to both the node length and congestion, and is computed as follows:

$$c(n) = \frac{b(n) \times h(n) \times p(n)}{s(n)} + w_2 \times \frac{l(n)}{s(n)} \tag{2}$$

In Eq. (2), $l(n)$ is the node **l**ength, and $b(n)$ is the **b**ase cost related to the node direction and length. $s(n)$ represents the **s**hare factor that is related to the number of connections from the same net that are also using node n. The term $s(n)$ is designed to encourage sharing of RRG nodes from the same net. The **w**eights w_1 and w_2 in Eq. (1) and (2) are tunable. The **h**istorical and **p**resent congestion costs, $h(n)$ and $p(n)$, are defined as follows:

$$\begin{aligned} p(n) &= 1 + u(n) \times p_f \\ h(n) &= \begin{cases} h(n) + (u(n) - 1) \times h_f, & \text{if } u(n) > 1 \\ h(n), & \text{otherwise} \end{cases} \end{aligned} \tag{3}$$

In Eq. (3), the node **u**sage $u(n)$ is the number of nets using node n, and p_f and h_f represent the present and historical congestion factors, respectively.

4.3.2 Congestion factor update. In RWRoute, the default values of p_f and h_f are 2 and 1, respectively. In contrast, the enhanced PathFinder [15] sets p_f to 1 and h_f to 5, as they found that a fast-growing $p(n)$ can result in slow congestion fixing and large variance between different net routing orders. Unlike RWRoute and the enhanced PathFinder, which use constant values for the congestion factors p_f and h_f, we have devised a dynamic updating mechanism for these congestion factors to help our router quickly resolve routing congestion.

The update of p_f is given by Eq. (4), where i represents the iteration number and α_1, α_2, and α_3 are parameters controlling the range and decaying speed of the congestion factors. In the early routing iterations, p_f is relatively large, which can help the router to quickly finish the routing for most non-critical nets. In later iterations, p_f is decreased to focus on fixing congestions by reducing the impact on the net routing orders.

$$p_f = p_f \times (\alpha_1 + \frac{\alpha_2}{1 + e^{i \times \alpha_3}}) \tag{4}$$

Eq. (5) gives the updating formula of h_f. Unlike p_f, h_f is increased gradually to encourage the router to pay more attention to congestion accumulated in recent iterations, which can accelerate fixing of congestion.

$$h_f = \frac{\beta_1}{1 + e^{-i \times \beta_2}} \tag{5}$$

Table 1: Important parameters in Potter

Parameter	Value	Parameter	Value	Parameter	Value
τ	25	r	0.05	w_1	0.8
w_2	0.2	α_1	1.1	α_2	3.3
β_1	2	β_2	0.5	α_3	1

Table 2: Statistics of FPGA 2024 contest benchmarks

Suite	Name	Nets (K)	LUTs (K)	FFs (K)	DSPs	BRAMs
LogicNets	jscl	28	31	2	0	0
Rosetta	fd	77	46	39	72	62
RapidWright	picoblaze_array	148	76	77	0	0
VTR	mcml	71	43	15	105	142
	lu64peeng	143	90	36	128	303
Corundum	25g	166	73	96	0	221
	100g	183	76	104	0	290
FINN	radioml	110	74	46	0	25
	mobilenetv1	296	202	140	48	562
Titan23	orig_gsm_x6	280	133	160	0	432
	orig_dart_x4	351	299	176	0	0
MLCAD 2023	d181_lefttwo3rds	361	155	203	1,344	405
	d181	535	229	303	1,824	576
BOOM	med_pb	54	36	17	24	142
	soc	274	227	98	61	161
	soc_v2	275	229	99	61	161
ISPD 2016	fpga03	387	214	168	500	590
	example2	449	289	234	200	384
	example2_v2	449	254	234	200	384
Koios 2.0	clstm_like_large	270	89	184	1,289	370
	dla_like_large	509	189	362	2,209	192
	dla_like_large_v2	509	189	363	2,209	192
CoreScore	500	179	96	116	0	250
	500_pb	175	96	116	0	250
	900	321	174	210	0	451
	1200	428	233	280	0	601
	1500	538	292	350	0	720
	1700	617	344	399	0	720

4.4 Post-processing

In the final post-processing stage, we focus on routing the remaining high-fanout nets or congested nets which may be difficult for the parallel overlap-tolerant routing to fix. Like existing partitioning-based parallel routers [3, 14, 17], we will conduct parallel routing for the remaining unrouted or congested nets whose bounding boxes do not overlap. It is worth mentioning that there are only a few nets that need to be rerouted at this stage and the routing time is significantly less than that of the overlap-tolerant routing stage. Finally, the routing tree of each multi-pin net needs to be constructed by merging the common routing paths of its connections. However, there may exist cycles in the resulting routing tree when different routing paths share some RRG nodes. To remove cycles, we will conduct the Dijkstra's algorithm in the routing tree to find the shortest path from each sink node to the source node and remove the edges not in the shortest paths from the tree.

5 EXPERIMENTAL EVALUATION

In this section, we present the detailed experimental results of different methods on the FPGA 2024 contest benchmarks. Potter is implemented in C++, and we summarize the important parameters adopted in Potter in Tab. 1. All experiments are conducted in a high-performance computing (HPC) server with a 2.90GHz CPU. The default number of threads is set as 32 for all methods. We adopt the default evaluator used in the FPGA 2024 contest [4] to evaluate

Table 3: Overall results. T and WL denote the running time and wire length of critical paths respectively. The reported running time includes both IO and routing times. The geomean of results is calculated to reduce the effect of outliers.

Suite	Name	Vivado		RWRoute [16]		CUFR [14]		GRoute		Potter	
		T (s)	WL	T (s)	WL	T (s)	WL	T (s)	WL	T (s)	WL
LogicNets	jscl	75	310	50	226	32	234	57	248	23	221
Rosetta	fd	147	888	162	839	123	804	71	896	45	812
RapidWright	picoblaze_array	220	156	162	109	68	111	67	113	33	110
VTR	mcml	494	666	246	594	94	584	97	685	43	607
	lu64peeng	219	1,728	219	1,412	114	1,333	81	1,571	54	1,308
Corundum	25g	-*	-*	243	396	131	500	78	398	48	401
	100g	-*	-*	226	549	105	549	76	575	46	562
FINN	radioml	154	338	114	277	62	251	61	249	31	258
	mobilenetv1	468	515	457	393	182	320	88	378	66	384
Titan23	orig_gsm_x6	400	406	482	298	188	289	95	307	66	271
	orig_dart_x4	669	1,224	825	609	264	567	124	718	93	639
MLCAD 2023	d181_lefttwo3rds	409	1,159	1,754	809	406	771	165	924	125	727
	d181	631	1,104	4,161	790	595	828	230	966	188	779
BOOM	med_pb	139	823	233	969	141	806	80	896	38	817
	soc	711	2,235	1,264	1,698	632	1,673	279	1,732	230	1,708
	soc_v2	870	1,643	4,976	1,167	1,231	1,020	554	1,239	205	982
ISPD 2016	fpga03	405	1,453	553	892	352	848	133	942	82	873
	example2	386	1,481	566	1,114	312	939	97	1,018	83	1,003
	example2_v2	397	1,265	504	765	257	683	116	702	76	669
Koios 2.0	clstm_like_large	324	390	174	263	95	228	71	339	48	238
	dla_like_large	541	927	381	548	185	520	92	547	74	552
	dla_like_large_v2	544	597	395	339	211	284	95	394	77	311
CoreScore	500	187	751	154	680	74	668	69	678	36	655
	500_pb	227	861	270	687	138	739	79	674	50	660
	900	336	1,586	298	1,199	134	1,216	84	1,345	53	1,215
	1200	462	1,641	413	1,072	182	1,105	91	1,206	70	1,106
	1500	547	1,679	566	1,254	236	1,331	128	1,389	93	1,302
	1700	792	2,182	1,431	1,590	656	1,544	255	1,790	136	1,570
Geomean		354	897	410	642	183	616	106	679	66	617
Ratio of Geomean		5.34	1.45	6.18	1.04	2.76	1.00	1.59	1.10	1.00	1.00

*Vivado failed to route two cases from the corundum suite due to failure in DRC during the routing.

the running time and wire length of the critical paths. Since the running time reported by the contest evaluator also includes the IO time, we will also compare the routing time for each method. We run all experiments five times and report the average results for all methods to reduce the effect of randomness.

5.1 Benchmarks and Baselines

The circuits studied in the experiment include all the benchmarks in the FPGA 2024 contest [4]. As shown in Tab. 2, there are 28 circuits in total from 12 different suites of public circuit benchmarks. The target FPGA device is Xilinx UtraScale+ xcvu3p. The placement solutions are generated by Vivado.

We compared Potter with four baselines which are introduced as follows:

- GRoute[1] was the winner in the FPGA 2024 contest. It is implemented in C++ and applies a two-phase multi-threaded routing.
- CUFR[2] [14] was the second place in the FPGA 2024 contest. It is implemented in Java and utilizes a recursive partitioning ternary tree to schedule the multi-net parallel routing.
- RWRoute [16] is an open-source sequential router provided by Java-based RapidWright [6] and serves as a reference router in the FPGA 2024 contest.
- Vivado v2021.1 was used to do the routing task with the command: `"route_design -no_timing_driven -preserve"`, following the settings of the FPGA 2024 contest. It is worth noting that Vivado targets final performance, such as frequency, while the evaluation metric in the experiment is only wirelength.

5.2 Overall Results

The overall results of all the methods are presented in Tab. 3. Among the four baselines, CUFR achieves the best wire length, and GRoute has the best running time. The wire length of Potter is comparable to that of CUFR, with 4% and 45% improvements over RWRoute and Vivado, respectively. Moreover, the running time of Potter is

[1]The executable binary of the contest version was provided by the authors.

[2]The codes were obtained from https://github.com/xszang/parallel-routing.

59% faster than the fastest baseline. These results demonstrate the superiority of Potter in both speed and quality of routing.

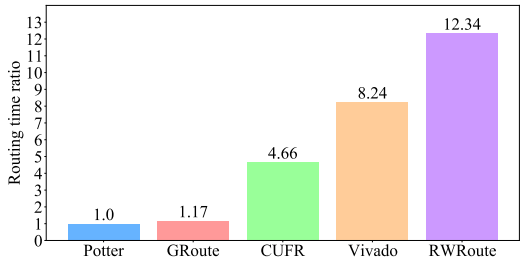


Figure 8: The routing time comparison. The ratios are calculated based on the average routing time of Potter.

Because the IO time may differ in different methods and could become the bottleneck in small cases, we also compare the routing times of different methods, as shown in Fig. 8. Compared with Vivado and RWRoute, Potter achieves an average speedup of 8.24× and 12.34×, respectively. Moreover, in some congested cases, such as MLCAD d181 and BOOM soc_v2, Potter is around 30× faster than RWRoute.

5.3 Stability Analysis

To reduce the idle time of threads and achieve maximum acceleration, Potter does not synchronize the data modification on RRG between different threads, which may bring a little variance in routing results. We analyze the stability of GRoute and Potter in five runs of the experiments. Fig. 9 shows the mean routing time and variance of GRoute and Potter for the ten slowest cases. Compared with GRoute, Potter has much smaller standard variances, which is mainly attributed to Potter's ability to reduce resource conflicts during net scheduling. Due to the low variance, Potter is much more stable and reliable in practice.

5.4 Thread Analysis

We also test Potter and GRoute with different numbers of threads, ranging from 1 to 32. As shown in Fig. 10, compared with GRoute, the routing time improvement of Potter increases more linearly with the number of threads, revealing the scalability of Potter. The speedup slows down when the thread number reaches 32 because the routing resource conflicts are more frequent and severe, requiring more time to fix the resource overflow.

5.5 Ablation Study

Finally, we conduct an ablation study to demonstrate the effectiveness of the dynamic congestion factor updating mechanism in Potter. We replace the dynamic congestion factors in Potter with the constant value used in RWRoute, which we refer to as Potter* in the following. As depicted in Fig. 11, compared with Potter*, the dynamic updating mechanism improves both the routing time and wire length. Notably, the routing time of MLCAD d181 is accelerated by 10.89× and the wire length of BOOM soc_v2 is reduced by 28%.

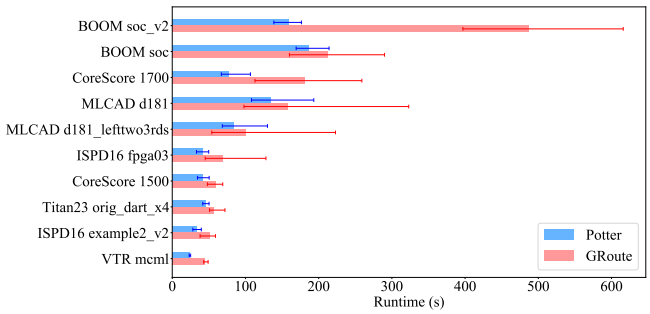


Figure 9: Variance of routing times in five runs. The bar length represents the mean and the line with caps on each bar represents the varying range.

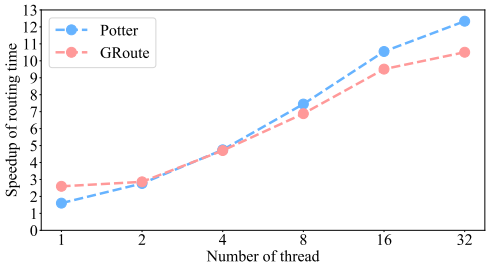


Figure 10: Speedup of routing time with different threads. The speedup is based on RWRoute.

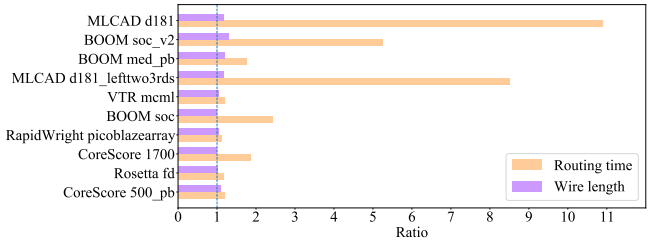


Figure 11: The ratio of Potter* to Potter. Ratios larger than 1 mean degradation of Potter* over Potter.

6 CONCLUSION

We have presented Potter, a fast parallel overlap-tolerant router designed to accelerate routing for UltraScale FPGAs. By minimizing net bounding box overlaps and balancing workloads across different threads, Potter achieves the maximum parallelism for routing. Additionally, we have proposed an algorithmic enhancement to the congestion factor updating in the negotiation-based routing algorithms, which accelerates routing convergence. Experimental results on the FPGA 2024 contest benchmarks demonstrate the superiority of Potter over state-of-the-art methods in terms of routing time and wire length. Moreover, as Potter supports the FPGAIF format, it can seamlessly integrate with the Vivado flow, providing a substantial acceleration to the FPGA development process.

7 ACKNOWLEDGMENT

The authors would like to thank Prof. Gary Grewal and Mr. Dani Maarouf for providing the executable program of GRoute for the FPGA 2024 routing contest.

REFERENCES

[1] Marcel Gort and Jason H Anderson. 2011. Accelerating FPGA routing through parallelization and engineering enhancements special section on PAR-CAD 2010. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 1 (2011), 61–74.

[2] Chin Hau Hoo, Yajun Ha, and Akash Kumar. 2016. ParaFRo: A hybrid parallel FPGA router using fine grained synchronization and partitioning. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–11.

[3] Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 67–76.

[4] Eddie Hung, Chris Lavin, Zak Nafziger, and Alireza Kaviani. 2024. Runtime-First FPGA Interchange Routing Contest @ FPGA'24. https://xilinx.github.io/fpga24_routing_contest/index.html. [Online; accessed 30-April-2024].

[5] Maciej Kurc. 2022. FPGA Interchange Format. https://fpga-interchange-schema.readthedocs.io/. [Online; accessed 30-April-2024].

[6] Chris Lavin and Alireza Kaviani. 2018. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 133–140.

[7] Larry McMurchie and Carl Ebeling. 1995. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. *Third International ACM Symposium on Field-Programmable Gate Arrays* (1995), 111–117. https://api.semanticscholar.org/CorpusID:14212822

[8] Kevin E Murray, Oleg Petelin, Sheng Zhong, Jia Min Wang, Mohamed Eldafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G Graham, Jean Wu, Matthew JP Walker, et al. 2020. Vtr 8: High-performance cad and customizable fpga architecture modelling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 2 (2020), 1–55.

[9] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 118–125.

[10] Minghua Shen and Nong Xiao. 2021. Load Balance-Centric Distributed Parallel Routing for Large-Scale FPGAs. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 242–248.

[11] Mirjana Stojilović. 2017. Parallel FPGA routing: Survey and challenges. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[12] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. 2019. CRoute: A fast high-quality timing-driven connection-based FPGA router. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 53–60.

[13] Dekui Wang, Zhenhua Duan, Cong Tian, Bohu Huang, and Nan Zhang. 2019. ParRA: A shared memory parallel FPGA router using hybrid partitioning approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 4 (2019), 830–842.

[14] Xinshi Zang, Wenhao Lin, Shiju Lin, Jinwei Liu, and Evangeline FY Young. 2024. An Open-Source Fast Parallel Routing Approach for Commercial FPGAs. In *Proceedings of the Great Lakes Symposium on VLSI 2024*.

[15] Yue Zha and Jing Li. 2022. Revisiting pathfinder routing algorithm. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 24–34.

[16] Yun Zhou, Pongstorn Maidee, Chris Lavin, Alireza Kaviani, and Dirk Stroobandt. 2021. RWRoute: An open-source timing-driven router for commercial FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15, 1 (2021), 1–27.

[17] Yun Zhou, Dries Vercruyce, and Dirk Stroobandt. 2020. Accelerating FPGA routing through algorithmic enhancements and connection-aware parallelization. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13, 4 (2020), 1–26.