ParaDRo: A Parallel Deterministic Router Based on Spatial Partitioning and Scheduling

Chin Hau Hoo
Department of Electrical & Computer Engineering
National University of Singapore
chinhau.hoo@u.nus.edu

Akash Kumar
Center for Advancing Electronics
Technische Universität Dresden
akash.kumar@tu-dresden.de

ABSTRACT

Routing of nets is one of the most time-consuming steps in the FPGA design flow. Existing works have described ways of accelerating the process through parallelization. However, only some of them are deterministic, and determinism is often achieved at the cost of speedup. In this paper, we propose ParaDRo, a parallel FPGA router based on spatial partitioning that achieves deterministic results while maintaining reasonable speedup. Existing spatial partitioning based routers do not scale well because the number of nets that can fully utilize all processors reduces as the number of processors increases. In addition, they route nets that are within a spatial partition sequentially. ParaDRo mitigates this problem by scheduling nets within a spatial partition to be routed in parallel if they do not have overlapping bounding boxes. Further parallelism is extracted by decomposing multi-sink nets into single-sink nets to minimize the amount of bounding box overlaps and increase the number of nets that can be routed in parallel. These improvements enable ParaDRo to achieve an average speedup of 5.4X with 8 threads with minimal impact on the quality of results.

KEYWORDS

FPGA; EDA; Parallel Routing

ACM Reference Format:
Chin Hau Hoo and Akash Kumar. 2018. ParaDRo: A Parallel Deterministic Router Based on Spatial Partitioning and Scheduling. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages.
https://doi.org/10.1145/3174243.3174246

1 INTRODUCTION

Moore's law has enabled the fabrication of FPGAs of increasing capacity, and modern FPGAs can accommodate state-of-the-art designs that are larger and more complex. To fully utilize the capacity of modern FPGAs, high-quality EDA tools are required. However, the long execution time of the tools causes a productivity gap where the capacity of FPGAs is growing faster than the ability of engineers to effectively utilize it. It is also one of the major factors that are preventing the wide adoption of FPGAs as a general computing platform.

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174246

Among the tools for packing, placement and routing for FPGAs, routing contributes almost 45% to the total execution time for Titan [7] benchmarks. One possible way of reducing the FPGA route time is by using a faster processor. Unfortunately, the failure of Dennard's scaling limits the maximum speed of a single processor. Therefore, recent works focus on the development of parallel FPGA routers to reduce the time spent in routing. Existing parallel routers are based on Pathfinder [6], which is one of the most widely used algorithms for sequential FPGA routing. The Pathfinder algorithm works by iteratively increasing the costs of using overused routing resources (RRs) until congestion is eliminated.

The sequential nature of Pathfinder poses a huge challenge to the design of a scalable **and** deterministic parallel FPGA router. In fact, existing works generally sacrifice scalability for determinism and vice versa. To guarantee determinism, the congestion costs seen by a net must be the same across different runs of the Pathfinder algorithm. In other words, race conditions on the congestion costs must be prevented by ensuring that nets routed by different threads/processes do not modify the congestion cost associated with the same RRs. Some existing works [3, 10] achieve this by spatially partitioning the FPGA and routing nets that are located within different partitions in parallel. Since nets in different partitions access disjoint subgraphs of the RR, there is no possibility of race conditions. However, the maximum speedup is limited by the fact that nets within the same partition must be routed sequentially. In this paper, we improve upon the idea of spatial partitioning and attempt to extract further parallelism by scheduling nets within a partition to be routed in parallel while guaranteeing determinism.
In summary, our contributions are as follows:

- A parallel FPGA router based on spatial partitioning with enhancements to improve speedup.
- Multi-sink net decomposition and bounding box minimization heuristic to increase the number of nets within a partition that can be routed in parallel.
- Mapping of the net scheduling problem to a graph coloring problem and a fast heuristic to solve it.

The rest of this paper is organized as follows. Section 2 describes background of the FPGA routing problem. Section 3 surveys the existing works on parallel FPGA routing. Section 4 explains the design of ParaDRo and its various enhancements. Section 5 presents the experimental setup while Section 6 discusses the performance of ParaDRo in terms of speedup and quality of results. The paper is concluded in Section 7.

2 BACKGROUND

Given a set of nets to be routed, routing connects the source to the sinks of each net using the RRs in the FPGA such that each RR is utilized by at most one net. The RRs are modeled as a graph where the nodes represent the wires in the FPGA and input/output pins of

the computation elements while the edges of the graph represent the programmable switches between the wires and input/output pins.

One of the most effective algorithms for solving the FPGA routing problem is Pathfinder [6]. In Pathfinder, Dijkstra's shortest path algorithm is used to find a path from the RR node of the source to the RR nodes of the sinks of a net. The cost of a RR node during the neighbor expansion of Dijkstra's algorithm is a weighted sum of the delay of the node and two additional cost components, the present and history congestion costs. The present congestion costs are updated after routing every net while the history congestion costs are updated after routing all the nets. The rationale behind the congestion costs is to penalize the overuse of RRs so that the overuse is eliminated eventually.

Since the present congestion costs are updated after routing each net, the costs seen by a net depends on the routes taken by previous nets. As a result, the present congestion costs introduce dependencies among nets and pose challenges to effective parallelization of the Pathfinder algorithm.

3 RELATED WORKS

In this section, we review existing works on parallel deterministic FPGA routers.

TDR [1] is a parallel router where each processor is allocated a disjoint subgraph of the RR graph and a subset of nets to be routed. To generate the disjoint subgraphs, TDR requires the FPGA to have the disjoint switch box topology. Since processors do not share RR, there is no need to synchronize congestion costs among them during routing. As a result, TDR achieved close to linear speedup.

Gort and Anderson [3] proposed a parallel router where congestion costs are synchronized among processors using the message passing interface (MPI). The costs are sent in a non-blocking manner to other processors once a processor finishes routing a net. Determinism is guaranteed by receiving the congestion costs in a blocking manner. Unfortunately, the blocking receive reduces the speedup because of increased stall time when there is load imbalance among the processors. Gort and Anderson tried to solve the problem by spatially partitioning the FPGA. A processor can route nets whose bounding boxes are within a partition without communicating with other processors that are routing nets in other partitions.

Another parallel router based on spatial partitioning is by Shen and Luo [10]. While [3] routes nets within a partition with multiple processors, Shen and Luo route nets within a partition sequentially. Therefore, the communication overhead of Shen and Luo is lower because there is no need to synchronize congestion costs among processors.

Gort and Anderson [2] found that 68% of the total route time is spent on the maze expansion step of Pathfinder and attempted to parallelize the step. In sequential maze expansion, a minimum cost node is popped from the priority queue, and neighbors of the node are sequentially pushed into the queue after the cost of using them are calculated by a single thread. On the other hand, Gort exploited fine-grained parallelism by calculating the costs of a node's neighbors in parallel with multiple worker threads and pushing the neighbors into the priority queue of the thread that calculated their costs. The worker threads wait at a barrier to ensure that all neighbors are expanded before moving on to the next minimum cost node.

Another fine-grained parallel router is Corolla [11], a GPU-accelerated router based on Bellman Ford's algorithm. Since running Bellman Ford's algorithm on the entire RR graph increases the problem size unnecessarily, Corolla executes the algorithm only on the subgraph within the bounding box of the net being routed. The subgraph is dynamically expanded until a legal routing solution is found. To extract sufficient parallelism, Corolla applies the operator formulation [9] where each RR node is mapped to a GPU thread to be processed in parallel. In addition to the fine-grained parallelism, Corolla also exploits coarse-grained parallelism where multiple nets are routed in parallel as long as they do not have overlapping bounding boxes.

4 PARADRO

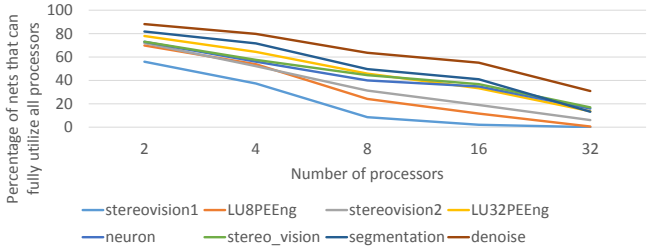

Figure 1: Percentage of nets that fully utilize all processors versus number of processors

Designing a scalable **and** deterministic parallel FPGA router is a non-trivial task. The deterministic parallel routers presented in Section 3 are based on either partitioning or fine-grained parallelism. Partitioning based routers [1, 3, 10] route multiple nets in parallel only if the nets access disjoint subgraphs of the RR graph to prevent race conditions. Instead of routing multiple nets in parallel, fine-grained routers parallelize either the classic maze expansion algorithm [2] or the Bellman Ford algorithm [11] to route a single net.

Unfortunately, routers based on partitioning [3, 10] do not scale well. As shown in Figure 1, the percentage of nets that can fully utilize all the processors reduces as the number of processors increases, resulting in diminishing speedup. While the figure assumes that the FPGA is partitioned into equally sized segments, the trend shown in the figures applies to the partitioning described in [3, 10]. Therefore, we propose ParaDRo, a Parallel Deterministic Router, based on spatial partitioning and scheduling to address the limitations of existing partitioning-based routers.

ParaDRo is based on recursive spatial bi-partitioning of the FPGA. The bi-partitioning forms a perfect binary tree where each node represents a region on the FPGA and contains nets whose bounding boxes fit entirely within the region. An example of a bi-partitioning tree is shown in Figure 2. Since the regions represented the nodes at each level of the tree are spatially independent, nets in one node can be routed in parallel with nets in other nodes while still guaranteeing determinism. For example, in Figure 2, nets A and B can be routed in parallel with nets D and E. Unfortunately, this simple bi-partitioning approach does not produce a good speedup because the number of interior (non-leaf) nodes at any level of the tree is less than the number of threads, which results in underutilization of the threads. In Figure 2, the number of nodes at level zero is one, which

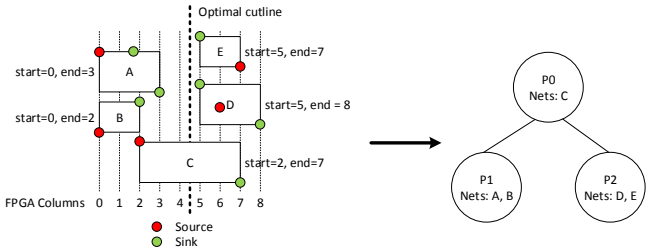

Figure 2: An example of a 2-level bi-partitioning tree. Partition P0 contains nets that cross the partition cutline whereas partitions P1 and P2 contain nets whose bounding boxes entirely fit into the left and right partition respectively.

is less than two, the number of threads that the tree is generated for. ParaDRo addresses this problem with three enhancements.

Firstly, interior nodes are further bi-partitioned to extract additional parallelism. The direction of this extra bi-partitioning is orthogonal to the direction that created the initial children of each interior node.

Secondly, ParaDRo schedules nets in interior nodes to be routed in parallel as long as they have non-overlapping bounding boxes. In order to minimize the overlap between nets, ParaDRo reduces the bounding box size of nets by decomposing multi-sink nets into single-sink nets called virtual nets. The bounding box size of virtual nets is further reduced by restricting the routes of a virtual net to be along the perimeter of its bounding box. Since there are two ways of routing virtual nets along the perimeter (bottom half or top half), a heuristic is introduced to choose the optimal bounding box shape for virtual nets. This heuristic is similar to global routing where coarse-grained routing channels are assigned to nets to minimize the overall congestion. After determining the bounding box shapes of the virtual nets, a schedule is generated for each node in the bi-partitioning tree to route the nets in the node in parallel whenever possible. The scheduling problem is reduced to a graph coloring problem where the graph models the nets to be routed and the overlap among them. The overlap graph is built such that virtual nets of the same net are scheduled to be routed sequentially. Therefore, subsequent sinks can reuse the existing route tree in a similar manner as VTR.

Thirdly, due to the smaller bounding boxes of the virtual nets, it was observed that ParaDRo fails to converge to a congestion-free state for some benchmarks circuits. Therefore, ParaDRo reroutes only congested nets when the number of congested nets is less than a predefined threshold. In this stage, ParaDRo expands the bounding box of the virtual nets to an axis-aligned bounding box covering the source and sink of the net to increase the routing flexibilty when eliminating congestion.

4.1 Recursive Bi-partitioning

ParaDRo starts by recursively splitting the FPGA into two disjoint regions while minimizing the difference in the workload of routing the nets in the two regions as shown in Algorithm 1. When determining where to cut the FPGA into two regions, ParaDRo leverages on the discrete nature of the FPGA architecture so that only a finite number of cutlines needs to be considered. For example, if the number of rows and columns in an FPGA are n_{rows} and $n_{columns}$, there are only $n_{rows} + 1$ horizontal and $n_{columns} + 1$

Algorithm 1 The recursive bi-partitioning algorithm

```
1:  procedure RECURSIVE_BIPART(bb, nets, n_levels, cur_level, node)
2:      if cur_level < n_levels − 1 then
3:          nets_before, nets_after, opt_cut ← get_opt_cut(bb, nets)   ▷ Algorithm 2
4:          nets_crossing ← nets − nets_before − nets_after
5:          node.nets ← nets_crossing
6:          bb_before, bb_right ← split_bounding_box(bb, opt_cut)
7:          recursive_bipart(bb_before, nets_before, n_levels, cur_level + 1, node.right)
8:          recursive_bipart(bb_after, nets_after, n_levels, cur_level + 1, node.left)
9:      else
10:         node.nets ← nets
11:     end if
12: end procedure
```

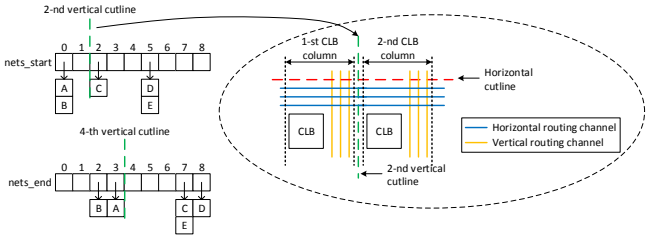

Figure 3: *nets_start* and *nets_end* generated based on the nets in Figure 2

vertical cutlines respectively. While the discrete nature of FPGAs significantly reduces the search space for an optimal cut, a naive way of performing bi-partitioning might lead to an inefficient algorithm. For example, finding the most load balanced cutline by calculating the total workload of nets that are located before and after every possible cutline leads to an inefficient $O(kN)$ algorithm where k is the number of cutlines and N is the number of nets. The bottleneck in this example lies in repeatedly determining the side of every net relative to every possible cutline, which requires the list of nets to be iterated k times.

In ParaDRo, the list of nets is only iterated once as shown in Algorithm 2. During the iteration, the contents of four arrays *nets_start*, *nets_end*, *total_workload_before* and *total_workload_after* are computed. These arrays allow the optimal cutline to be determined in $O(N + k)$ instead of $O(kN)$.

nets_start and *nets_end* are 2D arrays with the i-th element being an array of nets whose bounding boxes start and end at the i-th FPGA row/column respectively. The bounding box of a net can be represented as a 4-tuple $\langle xmin, xmax, ymin, ymax \rangle$, and the meaning of the start and end of a net bounding box depend on the orientation of the cutline. For a vertical cutline, the start and end of a bounding box are $xmin$ and $xmax$. Similarly, for a horizontal cutline, the start and end of a bounding box are $ymin$ and $ymax$. As shown in Algorithm 2, the start and end of a net are used to index into *nets_start* and *nets_end* respectively when adding the net to the arrays. Figure 3 shows an example of *nets_start* and *nets_end* that are generated based on the five nets shown in Figure 2. It also shows how a cutline is positioned relative to the CLBs and routing channels.

Algorithm 2 Fast heuristic to determine the optimal cutline

```
1: procedure GET_OPT_CUT(bb, nets)
2:     nets_start ← {{}}
3:     nets_end ← {{}}
4:     total_workload_before ← {0}
5:     total_workload_after ← {0}
6:
7:     for net ∈ nets do
8:         nets_start[net.start] ← nets_start[net.start] ∪ net
9:         nets_end[net.end] ← nets_end[net.end] ∪ net
10:        total_workload_before[net.end] ← total_workload_before[net.end] + net.workload
11:        total_workload_after[net.start] ← total_workload_after[net.start] + net.workload
12:    end for
13:    total_workload_before ← prefix sum of total_workload_before starting from first element
14:    total_workload_after ← prefix sum of total_workload_after starting from last element
15:
16:    opt_cut ← −1
17:    min_diff ← ∞
18:    for cut ∈ cut indices do
19:        diff ← |total_workload_before[cut] − total_workload_after[cut + 1]|
20:        if diff < min_diff then
21:            min_diff ← diff
22:            opt_cut ← cut
23:        end if
24:    end for
25:    return ⋃_{i=0}^{opt_cut} nets_end[i], ⋃_{i=opt_cut+1}^{last} nets_start[i], opt_cut
26: end procedure
```

nets_start and *nets_end* allow the nets that are located after and before a specific cutline respectively to be determined easily. The list of nets located after the n-th cutline is simply $\bigcup_{i=n}^{last} nets\_start[i]$. Similarly, the list of nets located before the n-th cutline is $\bigcup_{i=0}^{n-1} nets\_end[i]$. For example, from Figure 3, the list of nets located after the 2-nd cutline is $\{C, D, E\}$ while the list of nets located before the 4-th cutline is $\{A, B\}$.

total_workload_before and *total_workload_after* are 1D arrays whose i-th element represents the total workload of all nets before the $(i+1)$-th cutline and after the i-th cutline respectively. They are generated in 2 steps. The first step is done while iterating through the list of nets. For each net, its workload, which is approximated by the number of sinks, is added to the element of *total_workload_after* and *total_workload_before* at indices equal to the start and end of the net's bounding box respectively. The second step performs in-place prefix sums on both the arrays to generate the final values. The prefix sums are calculated starting from the first and last element for *total_workload_before* and *total_workload_after* respectively. The steps are illustrated in Figure 4 and 5.

After generating the arrays, the optimal cutline can be determined by finding an index *opt_cut* such that the absolute difference between $total\_workload\_before[opt\_cut]$ and $total\_workload\_after[opt\_cut + 1]$ is minimum. Based on *opt_cut*, the nets that are before and after the cutline can be obtained directly as $nets\_before = \bigcup_{i=0}^{opt\_cut} nets\_end[i]$ and $nets\_after = \bigcup_{i=opt\_cut+1}^{last} nets\_start[i]$

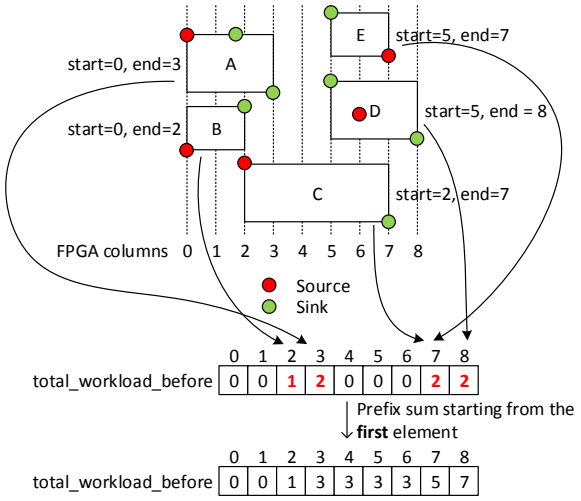

Figure 4: The process of generating *total_workload_before*

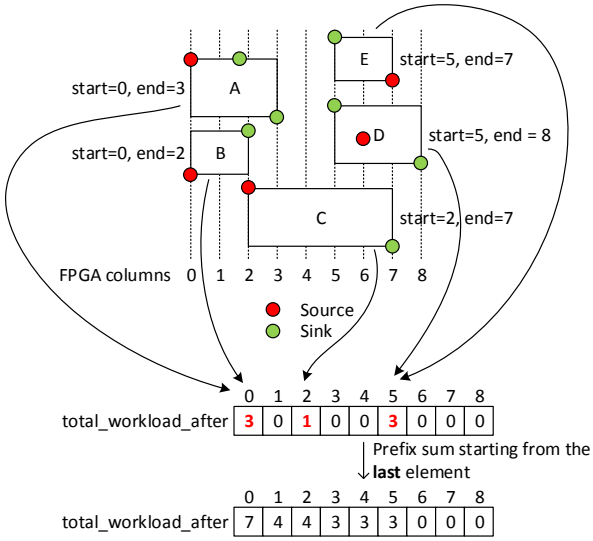

Figure 5: The process of generating *total_workload_after*

respectively. Nets that are not in *nets_before* and *nets_after*, *nets_crossing*, are contained in the parent node of the two child nodes that contain nets in *nets_before* and *nets_after*. The biparitioning process is repeated for nodes that contain *nets_before* and *nets_after*. ParaDRo also repeats the bi-partitioning process for the nets in *nets_crossing* to extract further parallelism. For simplicity purposes, the bi-partitioning process of *nets_crossing* is not shown in Algorithm 1.

Figure 2 shows the optimal cut and the associated bi-partitioning tree when ParaDRo is run with two threads. The FPGA is bi-partitioned once for two threads, and $|total\_workload\_before[opt\_cut] - total\_workload\_after[opt\_cut + 1]| = 0$.

4.2 Scheduling

After building the bi-partitioning tree, a directed task graph is generated for each node of the tree to identify nets that can be routed in parallel. The motivation behind this is the decreasing potential parallelism as the depth of a node, which is the number of edges between the node and the root of the tree, decreases. The worst case happens at a depth of zero where the potential parallelism is one because there is only one node, which is the root of the tree.

However, maximizing the number of nets that can be routed in parallel within a node is nontrivial due to a large number of overlapping nets. In order to guarantee determinism, these overlapping nets cannot be routed in parallel. By modeling the overlap between nets as a graph where the nodes represent the nets while the edges represent the bounding box overlap between the nets, it is easy to see that maximizing the number of nets that can be routed in parallel is equivalent to finding the chromatic number in graph coloring. The chromatic number is the minimum number of distinct colors that is required to color every node of a graph such that no adjacent nodes have the same color. A smaller chromatic number results in a higher number of nets that can be routed in parallel.

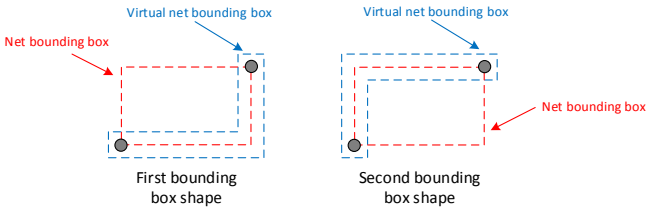

Figure 6: Two possible bounding box shapes for virtual nets

In order to maximize the number of nets that can be routed in parallel, it is important to reduce the bounding box overlap between the nets before performing graph coloring. Therefore, ParaDRo decomposes multi-sink nets into single-sink nets called virtual nets, which have smaller bounding boxes. Since a virtual net has only one sink, its bounding box size can be further reduced by restricting its route to be on the perimeter of its bounding box. Figure 6 shows the two possible routes that are on the perimeter and their associated bounding boxes. Figure 7 shows the process of decomposing two nets, A and B, into virtual nets A1, A2 and B1. It can be seen that virtual net B1 is no longer overlapping with virtual net A1 and A2 after the decomposition and a proper choice of bounding box shapes. The algorithm to make such a choice is described in Section 4.3.

After generating the virtual nets, an overlap graph is built based on their bounding boxes. The vertices of the graph represent the virtual nets while the edges represent a bounding box overlap between two virtual nets. An example of the overlap graph is shown in Figure 8. Since virtual nets of the same net always overlap at the source, the subgraph induced by their corresponding nodes forms a complete graph in the overlap graph. In Figure 8, the complete graph is between nodes A1 and A2. The complete graph ensures that sinks of the same net are routed sequentially so that each sink can reuse the route tree of sinks routed before it. While this is similar to the route tree reuse in VTR, the caveat is that the amount

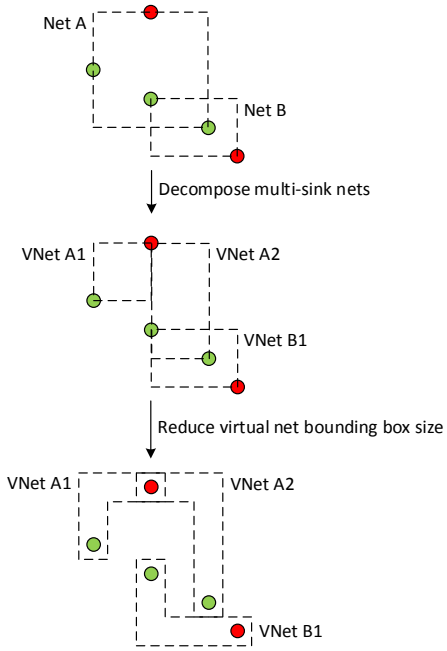

Figure 7: The process of decomposing multi-sink nets and reducing their bounding box sizes

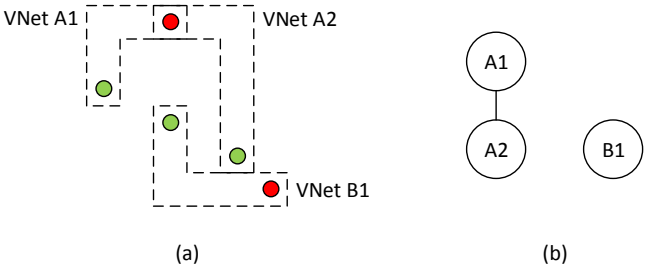

Figure 8: (a) Virtual nets (b) Their corresponding overlap graph

of reuse is lower because the bounding box of each sink is smaller than that of VTR's.

Although graph coloring is an NP-complete problem, there is a greedy algorithm to solve it. The greedy algorithm works by iterating through the nodes of the overlap graph in a certain order and greedily assigns each node a color that is not used by its neighbor. The quality of the coloring depends heavily on the order in which the nodes are colored. In ParaDRo, the nodes are visited in a smallest last ordering. The ordering is generated by repeatedly finding a node with the smallest degree and removing it from the graph until the graph is empty. Then, the nodes are returned in an order reverse to that in which they were found.

A simple way of routing the colored virtual nets would be in a bulk synchronous manner where nets of the same color have to finish routing in parallel before nets of the next color can be routed. However, this could lead to unnecessary waiting as shown in Figure 9(a). The bottleneck can be solved by converting the undirected overlap graph into a directed acyclic task graph shown in Figure

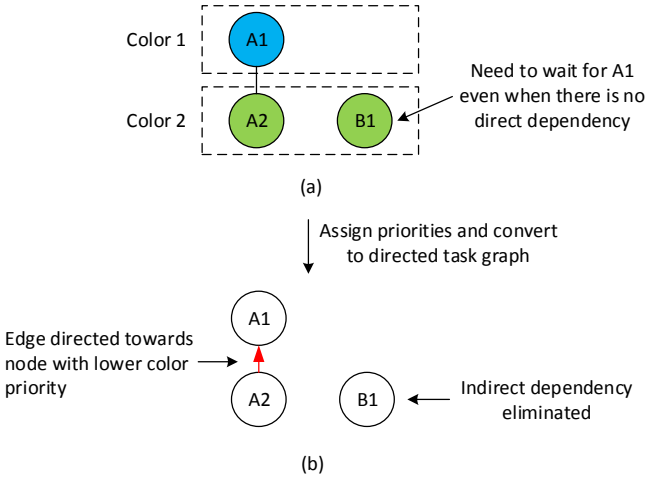

Figure 9: (a) An example of routing the overlap graph in Figure 8(b) where a net (B1) has to wait for another net (A1) to complete routing even when there is no direct dependency between them (b) Solving the waiting problem by converting the undirected overlap graph to a directed task graph

9(b). The task graph is similar to the overlap graph in that the nodes represent the nets but the edges impose an order in which the nodes are routed. Based on the task graph, a net can be routed as soon as all its parents have been routed.

The conversion works by assigning a priority to every color and setting the direction of overlap edges based on the color priority of the nodes that are connected by the edge. The color priority is determined by summing the number of sinks of all nets of the color. Then, edges are set to point from the node with a higher sum to the node with a lower sum. The rationale is that higher fanout nets should be routed first to ensure better convergence. The conversion imposes a partial order on the overlap graph nodes to ensure determinism.

4.3 Global Routing of Virtual Nets

As described in the previous section, there are two possible bounding box shapes of virtual nets. In this section, a heuristic is proposed to determine the shapes that minimize the total overlap between virtual nets. The heuristic is motivated by the following observations. Firstly, reducing the amount of overlap increases the number of nets that can be routed in parallel, which is the main goal of ParaDRo. Secondly, the amount of overlap has a direct impact on the congestion during routing.

Determining the bounding box shapes is similar to assigning routing channels to nets in global routing, and this similarity forms the basic idea of the proposed heuristic. One of the most important factors in the heuristic is the order in which nets are routed. Unfortunately, the actual order can only be determined after the scheduling described in Section 4.2, which requires the bounding box shapes in the first place to generate the overlap graph. In order to break the dependency loop, the net order is approximated by traversing the bi-partitioning tree in a breadth-first manner and adding the virtual nets in the node, after sorting them in decreasing number of sinks of their original net, to the list of virtual nets to be routed, *global_netlist*. The original net of a virtual net refers to the multi-sink net before decomposition. This approximation makes sense because, during scheduling, virtual nets whose original nets have higher number of sinks are scheduled to be routed earlier.

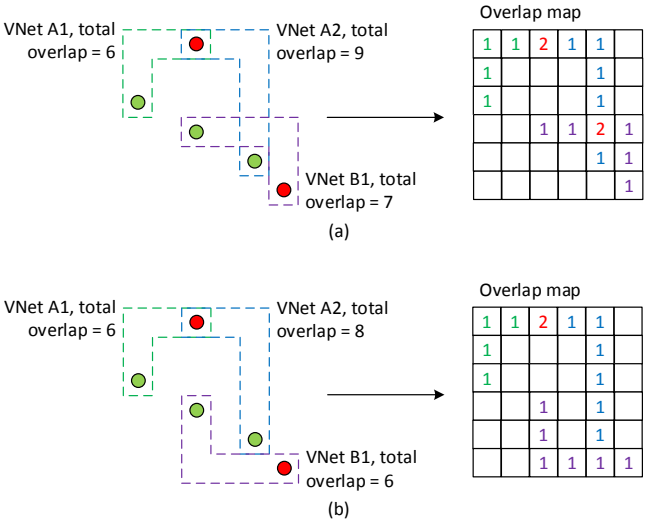

Figure 10: Different virtual net bounding boxes and their corresponding overlap maps

For each virtual net, *vnet*, in *global_netlist*, the optimal bounding box is determined by finding the box with the least overlap with virtual nets ordered before *vnet* in *global_netlist*. The decision to find the locally optimum bounding box is intentional to reduce the algorithmic complexity of the global routing so that its execution time does not outweigh the increased parallelism generated by it. The amount of overlap can be determined quickly by maintaining an overlap map, which is a 2D array of the same size as the FPGA grid. Each element of the array represents a discrete point in the FPGA grid and stores the number of globally routed virtual nets with bounding boxes that contain the point. With the overlap map, the total overlap of the two possible bounding boxes, *Box*1 and *Box*2, of *vnet* can be calculated by summing the elements of the map that corresponds to the points contained in *Box*1 and *Box*2. The box with the least total overlap is set as the bounding box for *vnet*. Figure 10 shows two different bounding box shapes for virtual net B1 and their corresponding overlap maps. It also illustrates how choosing the shape in Figure 10(b) for virtual net B1 leads to a lower total overlap of 6, assuming virtual net A1 and A2 are globally routed before B1. The choice of bounding boxes in Figure 10(b) not only enables virtual nets A2 and B1 to be routed in parallel, but also completely eliminates congestion between the them.

4.4 Actual Routing

After building the bi-partitioning tree and directed task graphs, the actual routing process starts from the root of the bi-partitioning tree. All nets with no parents in the directed task graph associated with the root of the bi-partitioning tree are routed in parallel. During routing, ParaDRo keeps track of the indegree of the nodes in the task graph. After routing a task graph node, the indegree of the children of the node is decremented by one. As soon as the indegree of a task graph node reaches zero, it can be scheduled to be routed

Table 1: Summary of benchmarks used in the experiments

Benchmark	Total nets	Total blocks	Minimum channel width
stereovision1	10,797	1,217	104
LU8PEEng	15,990	2,373	114
stereovision2	34,501	2,926	154
LU32PEEng	53,199	7,536	174
neuron	54,056	3,512	206
stereo_vision	61,883	3,434	228
segmentation	125,592	9,047	292
denoise	257,425	18,600	310

Table 2: Notations

Notation	Meaning
seq	Nets in nodes of the bi-partitioning tree are routed **sequentially**
par	Nets in nodes of the bi-partitioning tree are routed in **parallel**
normal	The bi-partitioning tree of *nets_crossing* is routed **sequentially**
extra	The bi-partitioning tree of *nets_crossing* is routed in **parallel**

in parallel. After routing all the nets in the task graph associated with the root of the bi-partitioning tree, the process is repeated recursively for the two children of the root.

Except for the root of the bi-partitioning tree, multiple task graphs are processed in parallel. In addition, the number of task graphs doubles each time the level of the tree increases by one. Coupled with routing nets in the task graphs in parallel whenever possible, ParaDRo can keep all the threads busy during routing. Another property that minimizes idling time during routing is that a node in the bi-partitioning tree can be routed as soon as its parent is done routing without waiting for other nodes in the same level to complete. This is because the FPGA region associated with a node only overlaps with the region associated with its parent, and nodes at the same level of the bi-partitioning tree are spatially independent.

4.5 Serial Equivalence

The number of levels of the bi-partitioning tree affects the route order of virtual nets and in turn the route solution. ParaDRo is implemented in such a way that the number of levels can be set independently of the number of threads because nets from the bi-partitioning tree are simply dispatched to a pool of worker threads to be routed. By keeping the number of levels constant, ParaDRo produces the same result even when the size of worker thread pool is changed. Therefore, ParaDRo is serially equivalent [4]. The number of levels in the bi-partitioning tree can be chosen carefully to strike a good balance between the amount of exposed parallelism and the overhead of task switching.

5 EXPERIMENTAL SETUP

ParaDRo was evaluated on a server equipped with two Intel Xeon E5-2680 V3 without hyper-threading and 128 GB of RAM. The operating system is Red Hat Enterprise 6.8 with Linux kernel version 2.6.32. gcc 7 with optimization flag -O3 was used to compile both ParaDRo and VTR [5].

The benchmarks used for evaluation were obtained from the VTR [5] and Titan [8] packages, and they are summarized in Table 1. The Titan benchmarks were chosen to cover a wide range of circuit sizes with *neuron* being the smallest and *denoise* being the largest out of the 23 Titan benchmarks. The benchmarks were packed and placed with default parameters using VTR of the same version as the Titan paper [8] (7.0 r4292) because the release version of VTR (7.0) crashes when loading Titan benchmarks due to a netlist loading bug. The architecture files used are k6_frac_N10_mem32K_40nm.xml and stratixiv_arch.timing.xml for VTR and Titan benchmarks respectively. The minimum channel widths in Table 1 are obtained by running VTR's routing in binary search mode.

The notations used to refer to different variants of ParaDRo are shown in Table 2.

6 EXPERIMENTAL RESULTS

In this section, two important performance metrics of ParaDRo, speedup and critical path delay, are presented.

6.1 Speedup Across Different Benchmarks

Figure 11 shows the speedups of different ParaDRo variants relative to their single-threaded variant. The time required to build the bi-partitioning tree is not included in the speedup calculation. Bi-partitioning tree generation for 2, 4, and 8 partitions currently requires on average 67.7%, 27.2%, and 15.6% of the routing time of single-threaded ParaDRo respectively. The tree generation takes longer for a smaller number of partitions because of the larger number of nets within a partition. Fortunately, generation of the bounding box overlap graph, which is the most time-consuming part of the tree building, is embarrassingly parallel because the bounding box overlaps of each net can be determined independently of other nets. The parallelization of the overlap graph generation is left as a future work.

It can be seen that LU32PEEng has significantly higher speedup than other benchmarks. On the other hand, stereovision1, neuron and stereo_vision are among the worst performing benchmarks in terms of speedup. The reason for this observation can be found in Figure 13, which shows the total time spent routing nets in all threads normalized to single-threaded ParaDRo. Only the **par extra** variant of ParaDRo is shown in Figure 13 because the total time measures only the raw time spent routing nets, which is the same for all variants of ParaDRo. The total time gives an approximation of the amount of work that is done by the router. In order to achieve good speedup, the amount of work must remain relatively constant as the number of threads increases. Unfortunately, this is not the case for most benchmarks as shown in Figure 13. The workload of ParaDRo depends on the order in which the virtual nets are routed. Therefore, the workload changes depending on the number of threads because a different number of threads generates a different bi-partitioning tree, which results in a different order in which the nets are routed. For LU32PEEng, the workload reduces significantly as the number of threads increases, which explains the super-linear speedup in Figure 11. On the other hand, the workload of stereovision1, neuron, and stereo_vision for more than one thread is actually more than

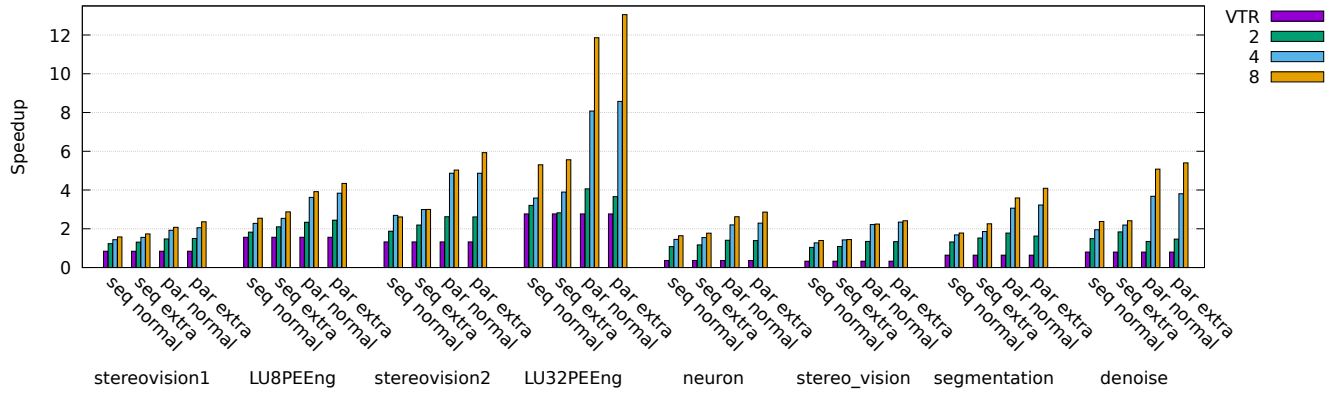

Figure 11: Speedups of different ParaDRo variants and VTR with 140% channel width normalized to single-threaded ParaDRo

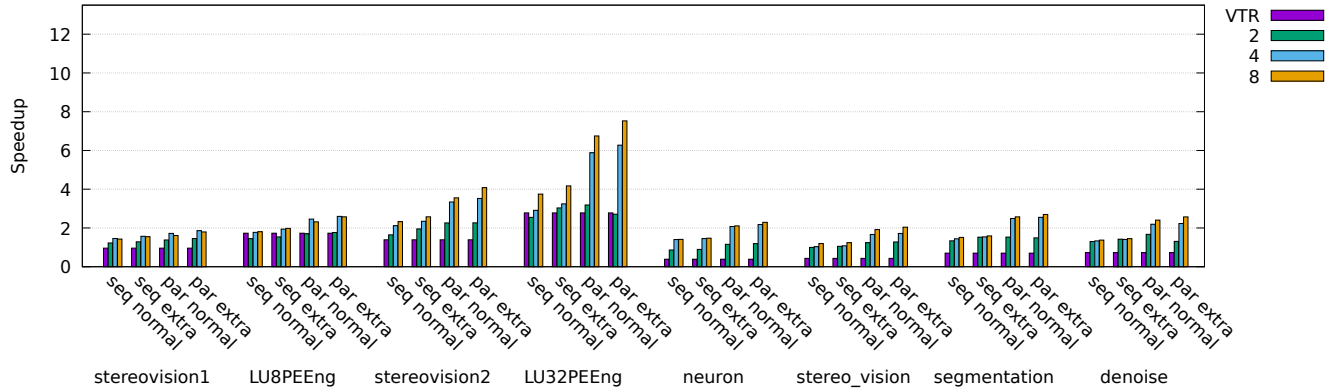

Figure 12: Speedups of different ParaDRo variants and VTR with 120% channel width normalized to single-threaded ParaDRo

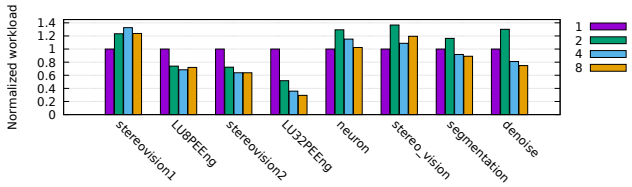

Figure 13: Total net route time of par extra ParaDRo with 140% channel width normalized to single-threaded par extra ParaDRo

that of single thread ParaDRo. Therefore, their speedups are among the worst in Figure 13.

Unfortunately, due to the difference in the route order, the workload when routing LU32PEEng with one thread is higher than that of VTR, causing ParaDRo to be slower than VTR for LU32PEEng. On the other hand, ParaDRo is significantly faster than VTR when routing Titan benchmarks. This is because an enhancement proposed by Gort [3] is added to ParaDRo. The enhancement is motivated by the fact that the Logic Array Block (LAB) in stratixiv_arch.timing.xml has equivalent output pins. The equivalence causes convergence issues because the packer assumes that each net will use only one LAB output pin but nets with multiple sinks tend to use more than one of those pins during routing. In order to solve the problem, the enhancement forces the router to use the same output pin to route the rest of the sinks once the first sink of the net has been routed.

Another observation that can be made from Figure 11 is that enabling the enhancements proposed in Section 4.1 and 4.2 improve speedup by different amounts for different benchmarks. This is because different benchmarks have different amounts of overlap between nets. Benchmarks with less number of overlaps allow more parallelism to be exploited when nets in nodes of the bi-partitioning tree are scheduled to be routed in parallel. For example, routing LU32PEEng with the **par** variant of ParaDRo improves speedup significantly as compared to routing with the **seq** variant of ParaDRo. stereovision1, on the other hand, does not exhibit such a huge improvement.

The speedup of ParaDRo when routing resources are more scarce is shown in Figure 12. It can be seen that the speedup of ParaDRo with only 20% higher than the minimum channel width is worse than that of ParaDRo with 40% higher than the minimum channel width. This is due to the restrictive virtual net bounding boxes when ParaDRo is run with more than one thread. Since virtual nets can only use RR on the perimeter of their bounding box, the flexibility of avoiding congestion is lower than when all the RRs in

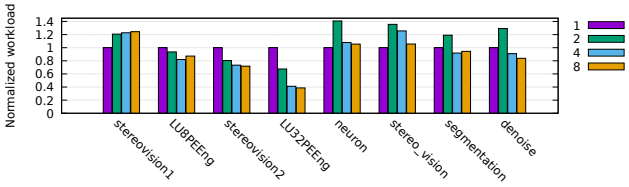

Figure 14: Total net route time of par extra ParaDRo with 120% channel width normalized to single-threaded par extra ParaDRo

the bounding box are available. The reduction in flexibility causes ParaDRo to work harder to resolve congestion as shown in Figure 14.

6.2 Effects of Proposed Enhancements

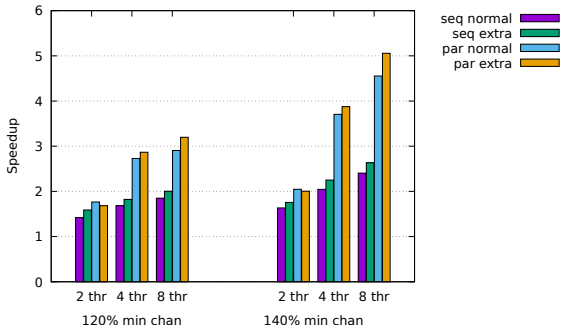

Figure 15: Average self-relative speedups of different ParaDRo variants

In this section, the effects of the enhancements discussed in Section 4.1 and 4.2 on ParaDRo's speedup are shown in Figure 15. The speedups in the figures are averaged across the benchmarks in Table 1.

Figure 15 shows that routing nets in nodes of the bi-partitioning tree in parallel results in higher speedup than routing them sequentially. In addition, the increase in speedup is more significant for 4 and 8 threads where the increase is almost two-fold. This improvement validates the hypothesis in Section 4.2 that spatial partitioning alone does not expose sufficient parallelism to keep the threads busy especially for higher number of threads. On the other hand, the improvement in speedup by routing the bi-partitioning tree of *nets_crossing* in parallel is not as significant as routing the main bi-partitioning tree in parallel.

6.3 Critical Path Delay

Figure 16 and 17 show the critical path delay of ParaDRo normalized to that of VTR with 40% and 20% higher than minimum channel width respectively. It is important to note that the different variants of ParaDRo shown in Section 6.2 are equivalent to one another in terms of the route solution produced. Therefore, their critical path delays are the same and not shown separately in the figures.

As shown in Figure 16 and 17, the worst degradation in critical path delay is only 5%. In addition, single thread ParaDRo is able

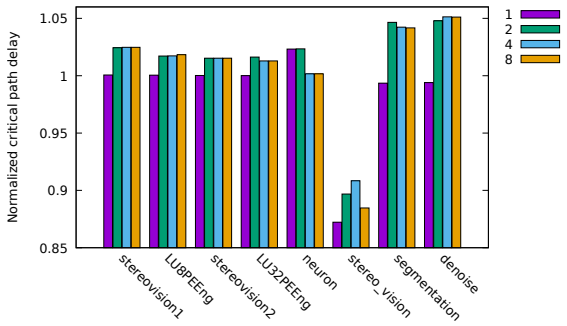

Figure 16: Critical path delay of ParaDRo with 140% channel width normalized to that of VTR

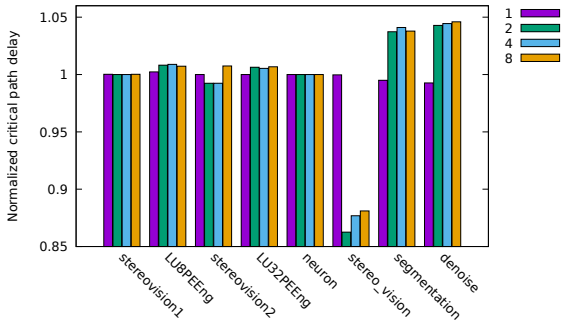

Figure 17: Critical path delay of ParaDRo with 120% channel width normallized to that of VTR

to produce critical path delay that is very close to VTR's for most benchmarks.

6.4 Total wirelength

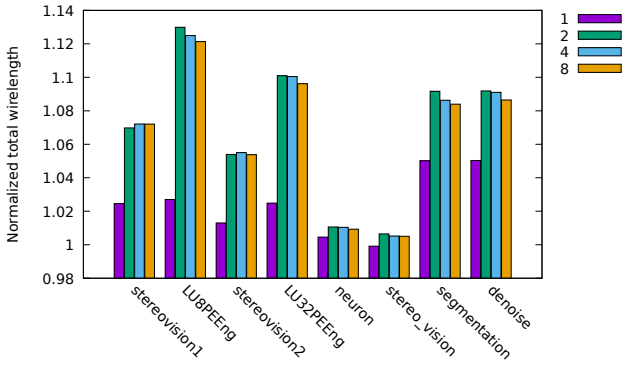

Figure 18: VTR normalized total wirelength of ParaDRo with 140% channel width

Due to the more restrictive bounding boxes of the virtual nets, nets might take longer routes to avoid congestion. In addition, the restriction also reduces the opportunity for route tree reuse in

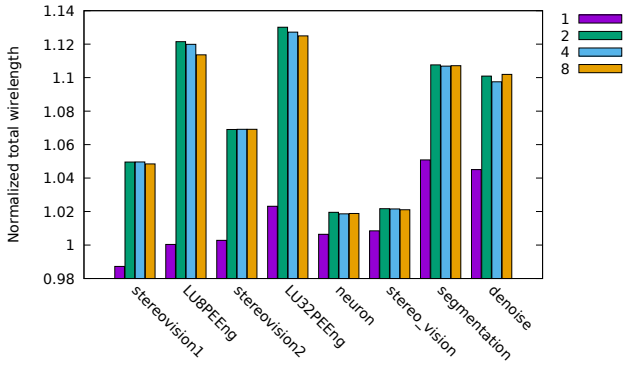

Figure 19: VTR normalized total wirelength of ParaDRo with 120% channel width

ParaDRo as compared to VTR. The effect of these factors on the total wirelength is shown in Figure 18 and 19.

Virtual nets are not generated for single-threaded ParaDRo because nets cannot be routed in parallel. However, the total wirelength of single-threaded ParaDRo is still higher than that of VTR for most benchmarks because the bounding box expansion factor (bb_factor) is lower than that of VTR. This is to be consistent with multi-threaded ParaDRo where the lower bb_factor increases the amount of parallelism by reducing the overlap between nets.

The absence of virtual nets in single-threaded ParaDRo is also the reason why single-threaded ParaDRo has significantly lower total wirelength than multi-threaded ParaDRo in Figure 18 and 19.

6.5 Impact of Serial Equivalence on Speedup

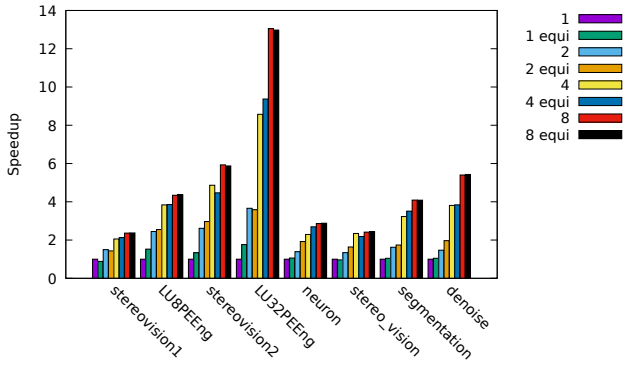

Figure 20: Speedup of serially equivalent versus non serially equivalent ParaDRo relative to single-threaded non serially equivalent ParaDRo

Figure 20 shows the effect of enforcing serial equivalence [4] on the speedup of ParaDRo. The channel width is set to 40% higher than the minimum, and the number of levels in the bi-partitioning tree is fixed to 4. As can be seen in the figure, the performance of serially equivalent ParaDRo is comparable to that of non serially equivalent ParaDRo.

6.6 Comparison with Existing Works

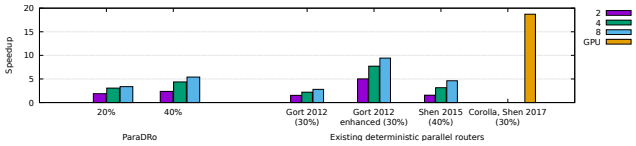

Figure 21: VTR normalized average speedup of ParaDRo and other existing works

Figure 21 compares the speedup of **par extra** ParaDRo to existing works. ParaDRo is faster than Shen's [10] and Gort's [3] router because ParaDRo routes nets within a partition in parallel while Shen and Gort do not.

On the other hand, Gort's enhanced router [3] is faster than ParaDRo because even though both routers are capable of rerouting only congested nets, the former does so for every routing iteration while the latter only does it when the number of congested nets drops below a threshold. Corolla [11] is also significantly faster than ParaDRo but its scalability is unclear because the authors did not evaluate Corolla with varying number of CUDA cores.

7 CONCLUSIONS

In this paper, we propose ParaDRo, a parallel deterministic router based on spatial partitioning. To improve speedup, nets within a partition are scheduled to be routed in parallel. Multi-sink nets are decomposed into single-sink nets, and their bounding boxes are shrunk to increase the number of nets that can be routed in parallel. With these enhancements, ParaDRo achieves a maximum speedup of 5.4X with 8 threads.

ACKNOWLEDGMENTS

This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed) at the Technische Universität Dresden.

REFERENCES

[1] Lucídio AF Cabral, Júlio S Aude, and Nelson Maculan. 2002. TDR: A distributed-memory parallel routing algorithm for FPGAs. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 263–270.
[2] Marcel Gort and Jason H Anderson. 2010. Deterministic multi-core parallel routing for FPGAs. In *FPT*. IEEE, 78–86.
[3] Marcel Gort and Jason H Anderson. 2012. Accelerating FPGA routing through parallelization and engineering enhancements, special section on PAR-CAD 2010. *IEEE TCAD* 31, 1 (2012), 61–74.
[4] Adrian Ludwin and Vaughn Betz. 2011. Efficient and deterministic parallel placement for FPGAs. *TODAES* 16, 3 (2011), 22.
[5] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. 2014. VTR 7.0: Next generation architecture and CAD system for FPGAs. *TRETS* 7, 2 (2014), 6.
[6] L. McMurchie and C. Ebeling. 1995. PathFinder: a negotiation-based performance-driven router for FPGAs. In *FPGA*.
[7] Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2013. Titan: Enabling large and complex benchmarks in academic CAD. In *FPL*. IEEE, 1–8.
[8] Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2015. Timing-driven Titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD. *TRETS* 8, 2 (2015), 10.
[9] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. *ACM Sigplan Notices* 46, 6 (2011), 12–25.
[10] Minghua Shen and Guojie Luo. 2015. Accelerate FPGA routing with parallel recursive partitioning. In *ICCAD*. IEEE, 118–125.
[11] Minghua Shen and Guojie Luo. 2017. Corolla: GPU-Accelerated FPGA routing based on subgraph dynamic expansion. In *FPGA*. 105–114.