RippleFPGA: A Routability-Driven Placement for Large-Scale Heterogeneous FPGAs

(Invited Paper)

Chak-Wa Pui, Gengjie Chen, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Peishan Tu, Hang Zhang, Evangeline F. Y. Young, Bei Yu

Department of Computer Science and Engineering,
The Chinese University of Hong Kong, NT, Hong Kong
{cwpui,gjchen,wkchow,lamkc,jkuang,pstu,hzhang,fyyoung,byu}@cse.cuhk.edu.hk

ABSTRACT

As the complexity and scale of FPGA circuits grows, resolving routing congestion becomes more important in FPGA placement. In this paper, we propose a routability-driven placement algorithm for large-scale heterogeneous FPGAs. Our proposed algorithm consists of (1) partitioning, (2) packing, (3) global placement with congestion estimation, (4) window-base legalization, and (5) routing resource-aware detailed placement. Experimental results show that our proposed approach can give routable placement results for all the benchmarks in the ISPD2016 contest and can achieve good result compared to the other wining teams of the ISPD2016 contest.

1. INTRODUCTION

Field-programmable gate array (FPGA) is an integrated circuit designed to be reconfigurable by a customer after manufacturing. The most common FPGA architecture, island-style, consists of a 2D array of configurable logic blocks (CLBs), I/O pads, routing channels and other types of resources such as random access memory blocks (RAMs) and digital signal processing blocks (DSPs). Within all these components, CLB remains an important resource since the major part of a given net-list is technology-mapped to CLBs.

Compared to ASICs, FPGAs have faster time-to-market and simpler design cycle, but FPGAs used to be selected for less demanding designs due to its limited performance. However, with its unprecedentedly increasing logic density, FPGA has become more competitive with ASICs especially in application specific implementations.

Placement is a key part in the FPGA design flow, which takes roughly half of the compilation time [3]. Given a circuit net-list, an FPGA placer produces a valid mapping of all blocks onto the target FPGA to optimize a certain objectives. Earlier works on placement for FPGAs using approaches like simulated annealing and min-cut partitioning cannot produce high quality placement result with scalable running time. With increasing design complexity and logic capacity in today's FPGAs, routability plays an ever more important role in placement. Therefore a high-quality, scalable routability-drive placement algorithm is needed.

ICCAD '16, November 07-10, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2966986.2980084

1.1 Previous Work

The numerous previous works on FPGA placement can be classified into three major categories: (1) simulated-annealing based approach, (2) partitioning-based approach, and (3) analytical approach. The most famous academic tool VPR [11] applies simulated annealing as its main tool to optimize objectives such as wirelength, timing, etc. Although it can achieve high quality result, its running time becomes a major drawback when placing a large circuit. Partitioning-based approaches like [12] shorten the running time by recursively partitioning a design and placing them hierarchically. However the partitioning-based methods may result in bad quality because the problem is solved locally after partitioning and these methods are not able to consider the global optimality. Compared to the above two methods, analytical approaches are more favourable especially as the gap between FPGAs and ASICs becomes smaller . Not only the industrial placers migrate from traditional simulated-annealing-based placement to an analytical approach, but also several new academic placement tools using analytical approach are shown to produce competitive results with much less running time on FPGAs. In [7], SimPL [9] is applied to FPGA placement, which yields the potential of using analytical methods in FPGA placement. In [10], NTUplace is used as the basic framework of the proposed analytical FPGA placer. Besides the placers mentioned above, there are also other analytical placers like LLP [15], StarPlace [16] and QPF [17].

However, these analytical placers mainly focus on how to migrate the traditional ASIC placement methods to FPGA placement and do not consider those FPGA specific routing congestion issues in their algorithms, which may cause problems in complicated designs.

1.2 Our Contributions

In this paper, we propose a routability-driven analytical placement algorithm for heterogeneous FPGAs. The major contributions are summarized as follows:

- An effective and efficient method is proposed to measure the routing congestion distribution on an FPGA.
- Several methods are proposed to reduce routing congestion, including partitioning, multi-stage congestion-drive global placement and a routing-resource-aware detailed placement.

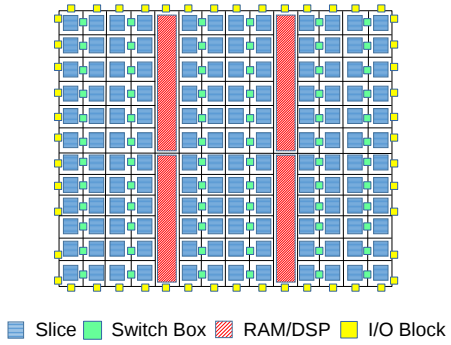


Figure 1: Our target FPGA architecture, showing different basic components.

- A complete framework for heterogeneous FPGA flat placement is proposed, which takes a net-list of basic logic elements as input.

The remainder of this paper is organized as follows. Section 2 gives an introduction to our targeting FPGA architecture and the problem formulation. Section 3 first gives an overview of our algorithm and then introduces its details which include partitioning, packing, congestion estimation, global placement, legalization, and detailed placement. Section 4 shows the experimental results, and we finally conclude in Section 5.

2. PRELIMINARIES

2.1 Target Architecture

The layout of our targeting heterogeneous FPGA architecture is shown in Figure 1. The smallest unit of the architecture is called logic element, which can be LUT, flip-flop (FF), RAM, DSP or IO. Each type of logic element can only be put into sites of its own type. To be specific, LUTs and FFs can only be put into sites called slice while RAMs, DSPs and IOs can only be put into sites called RAM, DSP and IO respectively. Unlike ASIC where each site can only have one cell, the target FPGA architecture allows multiple cells in a site. If two connected cells are put into the same site, routing the nets between them uses less routing resources compared to putting them in different sites. Among different types of sites, the legalization rules for slice are more complicated than others, whose only rule is that the number of logic elements cannot be larger than the sites' capacity. For slices, only one CLB is allowed to be placed in a slice and a CLB can contain up to 8 basic logic elements (BLEs) and each BLE can contain up to 2 LUTs and 2 FFs. Since there are internal wires inside a BLE, it will take less routing resources to route the net connecting the output of a LUT and the input of a FF if we put them in the same BLE. To simplify the terminology, we called BLE, RAM, DSP and IO as *element* in the following.

2.2 Problem Formulation

Given an FPGA with logic elements, its architecture and a design net-list, we need to map the net-list to the logic elements of the FPGA and determine their positions to minimize routed wirelength such that (1) each logic element is assigned to a legal position on the FPGA, and (2) the placement legalization rules of each site are satisfied.

3. PROPOSED ALGORITHM

3.1 Overview

Similar to ASIC placement, our placer solve the problem in three steps: (1) global placement, (2) legalization, and (3) detailed placement. Global placement gives the location of each logic element across the chip such that a given cost metric (e.g. wirelength) is minimized. Legalization gives each logic element a legal position while minimizing the disturbance to the global placement result. Finally, detailed placement further improves the solution.

In this paper, we propose a routablity-driven placement algorithm for large-scale heterogeneous FPGAs based on an upper-bound lower-bound global placement framework. Our algorithm consists of the following stages: (1) partitioning, which gives an initial location for each logic element and adjusts the area of the logic elements; (2) packing, which packs logic elements into BLEs based on their connections; (3) congestion-driven global placement with BLE, which gives a global placement result for legalization; (4) legalization, which determines the exact location of each logic element; and (5) detailed placement, which refines the placement with greedy algorithms. Several FPGA-specific techniques will be introduced in the following sections.

3.2 Partitioning

In this section, we utilize partitioning technique (please refer to [2] for a survey) to divide all logic elements into different components, followed by cluster reallocation to generate initial position for each logic element. In Section 4, this partitioning based pre-processing step is proved to be very useful in solving the congestion problem.

3.2.1 *Identify Sub-circuits*

Algorithm 1 Partitioning

```
Input: A given netlist(logic elements, nets)
for cell in logic elements do
    G.addVertex(cell);
end for
for cell0, cell1 in logic elements do
    if cell0.isConnect(cell1) then
        G.addEdge(cell0,cell1,#connection);
    end if
end for
minClusterSize ← netlist.cells.size() * 0.25;
maxCutSize ← netlist.nets.size() * 0.05;
clusterTree.root ← G;
cut ← G.getCut(minClusterSize, maxCutSize);
clusters ← G.getCluster(cut);
queue.push(clusters);
clusterTree.addChild(root,clusters);
while queue.isNotEmpty() do
    cluster ← queue.pop();
    cut ← cluster.getCut(minClusterSize, maxCutSize));
    if cut != NULL then
        clusters ← cluster.getCluster(cut);
        queue.push(clusters);
        clusterTree.addChild(cluster,clusters);
    end if
end while
```

Our sub-circuits identification algorithm is shown in Al-

Table 1: The cluster number obtained by our partitioning algorithm

Design	#Clusters	Design	#Clusters
FPGA-1	3	FPGA-7	3
FPGA-2	3	FPGA-8	3
FPGA-3	3	FPGA-9	3
FPGA-4	3	FPGA-10	3
FPGA-5	2	FPGA-11	3
FPGA-6	3	FPGA-12	3

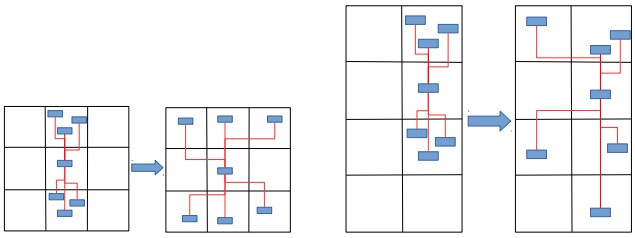


Figure 2: Spreading in different kinds of chips: (a) chip with balance height and width before and after spreading, (b) chip with imbalance height and width before and after spreading.

gorithm 1. In lines 1-8, we first model the given net-list as a graph $G(V, E)$. We let each logic element as a vertex in G and there is an edge between two vertices if and only if their corresponding logic elements are connected by a net. Furthermore, we set the weight of an edge $e_{uv} \in E$ to be the number of nets that connect vertices u and v.

We then partition G into two clusters recursively until the sizes of the resulting clusters are too small or the size of the minimum cut is too large as shown in lines 9-24. The resulting clusters of our partitioning algorithm are stored as leaves of a tree called clusterTree, which is constructed from line 15 to line 24. In our implementation, we set the minimum size of a cluster no smaller than 25% of the total number of vertices in G and the minimum cut should be at most 5% of the total number of nets. By limiting the sizes of the cut and the cluster, we can identify highly connected sub-circuits of relatively large sizes. As shown in Table 1, most of the designs in our experiment can be separated into clusters using our partitioning algorithm.

3.2.2 *Reallocate Clusters*

In traditional ASIC placement, to decrease the routing congestion level in an area, we will usually reduce the routing demand of that area by decreasing the cell density there as Figure 2(a) shows. However, this kind of methods cannot work on chips that are significantly imbalanced between its height and width. Not only because they have more cells in one direction which results in more demand of routing resources, but also because the lack of area in the other direction will limit the benefit of cell spreading.

As Figure 2(b) shows, given a net-list, the routing congestion cannot be resolve even if we spread the cells across the chip because we cannot decrease the congestion any more once our placement reach the horizontal bound of the chip.

If a net-list can be split into several clusters such that each of them is highly connected and there are only a few

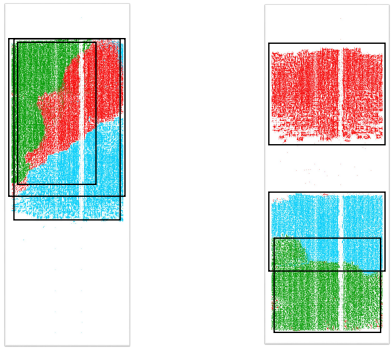


Figure 3: The difference of spreading in global placement: (a) without reallocation using partitioning (b) with reallocation.

connections between the clusters, routing congestion can be reduced if they are placed separately. Given two sparsely connected clusters, if we put them vertically (or horizontally, depending on the aspect ratio of the chip) instead of mixing them up as shown in Figure 3(a), the congestion can be reduced since there will be more spaces in the other direction for spreading within a cluster and their demands in the vertical (horizontal) routing resources will be distributed to different parts of the chip as shown in Figure 3(b).

The initial position of each cluster is determined by the lower bound computation which minimizes the HPWL, and changing the initial positions of the clusters may result in larger HPWL. In order to minimize the disturbance to the lower bound result, we will reallocate the clusters using the method as shown in Algorithm 2. In global placement, the initial position of each cluster is determined by the connections to IO and the connection between the clusters. Since the connections to IO have relatively small contribution in the routed wirelength, our reallocation algorithm only minimizes the disturbance to the connections among clusters as shown in Algorithm 2. First, in line 4, we sort the clusters by their y coordinates, which are the average of the y coordinates of the cells belonging to the clusters. The clusters are then placed from the bottom of the chip one by one. The height of each cluster is calculated in line 7 and its width is equal to the width of the chip. Finally we will spread the cells within each cluster as sparsely as possible while maintaining their relative locations as shown from line 8 to line 15.

3.2.3 *Adjust Cell Area*

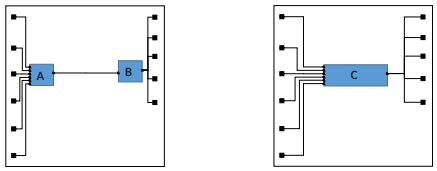
We also use the partitioning result to adjust the sizes of the logic elements. Given a netlist, if the clusterTree obtained in Algorithm 1 has only one node, the connections between different components in the net-list are very strong. Thus, it is very likely to cause routing congestion and we will increase the size of each logic element by about 50% in our implementation so that the elements can be placed more sparsely to reduce routing congestion.

3.3 Packing

In many previous works [4,5,11], LUTs and FFs are packed into CLBs according to some architecture-specified rules and

Algorithm 2 Reallocate cluster

Input: A given netlist(logic elements, nets)
1: clusterTree ← partitioning(netlist);
2: clusters ← clusterTree.getLeaves();
3: $ly \leftarrow 0$;
4: sort clusters by the y coordinate
5: **for** cluster in clusters **do**
6: insts ← clusers.getInsts(cluster);
7: $dy \leftarrow$ areaOfInsts/chipWidth;
8: sort insts by the x coordinate;
9: **for** i ← 1 : sizeof(insts) **do**
10: insts[i].$x \leftarrow i$/insts.size() ∗ chipWidth;
11: **end for**
12: sort insts by the y coordinate;
13: **for** $i \leftarrow 1$: sizeof(insts) **do**
14: insts[i].$y \leftarrow i$/insts.size() ∗ $dy + ly$;
15: **end for**
16: $ly \leftarrow ly + dy$;
17: **end for**



(a) (b)

Figure 4: The difference in shortest wirelength if merge BLE A and B to form BLE C: (a) minimum wirelength before packing (b) minimum wirelength after packing.

optimization rules. By doing this, FPGA placement resembles ASIC standard cell placement. As mentioned in Section 2.1, since the interconnect inside a BLE is hardwired, packing LUTs and FFs into BLEs can reduce the number of nets need to be routed. Instead of packing LUTs and FFs into CLBs like in other applications, our packing algorithm stops at the BLE level, which gives more flexibility to the later placement stages. Since BLE serves as the basic element in the later stages of our algorithm, the packing result will not violate the legalization rules.

In general, our packing algorithm consists of three stages: (1) global placement, (2) forming BLEs that consists of only one LUT and at least one FFs, (3) merging two BLEs into one if at least one of them do not contain FFs and their LUTs have many connections.

In stage one, a few iterations of global placement are performed to give the physical location of each logic element considering their connections globally. Packing locally optimizes HPWL by reducing the number of nets need to be routed, but the global optimality may be overlooked. We need this global placement step to consider HPWL globally. An example is given in Figure 4. If A and B are not merged, their optimal positions are as shown in Figure 4(a). If they are merged, their resulting HPWL will be longer as shown in Figure 4(b). Therefore in the later two stages, we will avoid packing logic elements whose physical locations are too far away from each other.

In stage two, we form BLEs which consists of only one LUT and at least one FFs. We first construct a graph $G(V, E)$ where V represents LUTs and FFs and E represent nets connecting the LUTs' output pins and the FFs' input pins. We solve a maximum cardinality matching problem on G, each matching represents a BLE obtained in stage two. For the remaining LUTs and FFs, we will further packed them into BLEs in stage three.

In stage three, we merge two BLEs into one if at least one of them do not contain FFs and their LUTs have strong connections. We first let each of the remaining LUTs and FFs that has not been packed in stage two form a BLE on its own. Similar to stage two, we construct a Graph $G(V, E)$ where V represents BLEs and there is an edge $e_{uv} \in E$ iff at least one of the BLEs represented by u, v do not contain FFs and their LUTs share more than two input pins. Furthermore, we let the weight of an edge e_{uv} be the number of input pins u, v share. We then solve a maximum weight matching problem on G, each matching indicates that the two corresponding BLEs should be merge into one.

In summary, we get three types of BLEs from our packing algorithm above: (1) BLEs containing both LUTs and FFs, (2) BLEs containing LUTs only, (3) BLEs containing FFs only.

3.4 Global Placement

3.4.1 *Our Global Placement Framework*

Our global placement framework is based on Ripple [8], which can be divided to two major parts: (1) lower-bound computation and (2) upper-bound computation.

In the lower-bound computation, the global placement problem for wirelength minimization can be formulated as a quadratic optimization problem, where the Bound2Bound (B2B) [14] net model is used to capture the HPWL objective. By minimizing the quadratic objective, we obtain a lower-bound computational result with minimized wirelength. Since the lower-bound computation aims at optimizing the HPWL, the result will have many overlaps. In the upper-bound computation, we will divide the chip into bins and spread the cell until the cell density of each bin is small enough. A pseudo-net is then added for each cell, that connects the location of the cell obtained from the previous round of upper-bound computation and the cell itself. By iteratively calling the lower-bound and upper-bound computation and increasing the pseudo-net weights, we can obtain a converged global placement result with very few overlaps and minimized wirelength.

In heterogeneous FPGA placement, we also need to consider the types of logic elements available in the sites. Otherwise, due to type mismatch, legalization may still cause large displacement to the global placement result even if the result has very few overlaps. In our global placement, fence constraint is used to avoid placing logic element in site whose type does not match with that logic element, e.g. placing DSPs into sites for CLBs. When spreading DSPs in the upper-bound computation, the cell density of the sites whose types are not DSP will be set to zero, so DSP will only go to the location where it can be placed. Similarly, RAMs and other logic elements are spread to sites of their types only.

3.4.2 *Three Stage Optimization*

Based on the framework, our global placement can be divided into three stages. In the first two stages, we search for

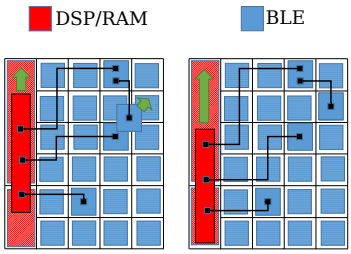


Figure 5: The mismatch between global placement and legalization (a) Legalize DSP/RAM and BLE with the same displacement (b) Legalizing DSP may induce more displacement.

good placement result in terms of HPWL. Since the physical locations for the elements on FPGAs are discrete, legalizing high-pin large-area elements like DSPs and RAMS may cause large disturbance to the global placement result. As Figure 5(a) shows, with the same displacement, DSPs and RAMs will increase the HPWL much more than BLEs because there are a lot of pins on DSPs and RAMs. Moreover, since the sizes of DSP and RAM are much bigger than that of other elements, legalizing DSPs and RAMs tends to induce more displacement as Figure 5(b) shows. We thus divide the traditional global placement into a two-stage process to reduce the disturbance caused by legalizing high-pin large-area elements. In each stage, we run global placement once with the same setting except that we perform a legalization process for RAMs and DSPs after stage one and fix their positions in later stages.

In the third stage, we more focus on routing congestion while maintaining previous good HPWL, where an iterative congestion-driven global placement is applied for a number of iterations. We tend to put more BLEs at locations with low congestion while those with high congestion should have less BLEs. Therefore in each iteration, the sizes of BLEs placed in congested and uncongested areas are inflated and shrunk respectively. In our implementation, we inflate the sizes of BLEs in the top 10% most congested areas and shrink those in the 30% least congested areas. Moreover, we will accumulate the inflation (shrinkage) rate in each iteration. It is because if an area stays congested in several iterations, that area is more likely to have congestion problem and the inflation rate of the cells in that area should be higher in order to lower the cell density in that region. The rationale for accumulating shrinkage rate for cell in least congested areas is similar. After changing the sizes of the BLEs, a short global placement with high pseudo net weight will be executed to adjust the positions of the BLEs to alleviate routing congestion. Since shrinking the sizes of the BLEs may result in placing too many BLEs in a region which may cause large displacement during legalization. Legalization will be applied at the beginning of each iteration, the average displacement of each site during legalization is used to estimate the cell density to ensure that the sizes of the BLEs at locations with large displacement will not be shrunk. By the procedure above, routing congestion can be distributed more evenly across the chip while the global placement result from the first two stages will not be worsened too much. As shown in Section 4, this congestion-driven stage is effective in reducing routing congestion, which results in shorter routed wirelength.

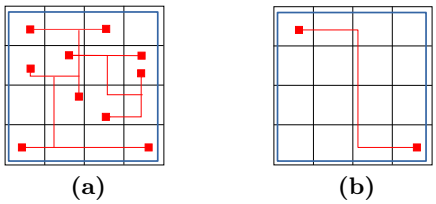


Figure 6: Bounding Boxes of Different Nets: (a)Bounding Box of a 10-Pins Net (b)Bounding Box of a 2-Pins Net.

3.5 Congestion Estimation

To estimate routing congestion of a placement result more precisely in the congestion-driven placement stage described in the previous section, a congestion map is needed.

In [19], the congestion map is built according to the net bounding box (BB). It estimates routing congestion by how many nets may possibly use the routing resources of that site. It accumulates the number of BB overlapping a site by equation (1) to estimate the routing congestion.

$$\text{Cong}_{\text{site}_i} = \#\text{BB overlaps with site}_i. \tag{1}$$

However BB of a net cannot reflect the physical locations of its internal pins, which can result in congestion estimation inaccuracy. Consider a 10-pin net n_a as shown in Figure 6(a) and a 2-pin net n_b as shown in Figure 6(b), they induce the same overlaps on the sites that their BBs cover, but net_a obviously need more routing resource than net_b. In order to take this into account, a weighted sum of the number of bounding box overlapping is used to measure congestion. In FPGAs, routing resources are allocated to switch boxes, so we will first divide the FPGA chip into global routing cell (G-Cells) such each G-Cell represents a switch box. For a G-Cell, the pins of the logic elements inside the sites that it covers are mapped to the G-Cell pins. After obtaining a new netlist between the G-Cells, we can calculate the weighted bounding box overlaps of each sites by,

$$\text{Cong}_{s_i} = \sum_{m \in N_i} \frac{W_m \cdot \text{HPWL}_m}{\#\text{G-Cells covered by net } m},$$

where W_m is a weight proportional to the number of pins of net m and N_i is the set of nets connected to the G-Cells covering site s_i. We give an example of this method in Figure 7: there are two nets on the chip and they contribute 1 and 1.7 to their covering sites respectively in the congestion map, resulting in a congestion map as shown in Figure 7. As Figure 8 shows, our model can estimate routing congestion distribution quite accurately comparing with the routing congestion distribution obtained by Vivado.

3.6 Legalization

During legalization, we need to consider the legality of a move and to minimize the disturbance to the global placement result simultaneously. Unlike ASIC legalization [6,13], our targeting FPGA architecture allows multiple cells in a site and we also need to follow the complex legalization rules.

As shown in Algorithm 3, our legalization algorithm uses a window-base method. We first find all the sites at a same certain distance to the cell as indicated in line 8. These sites

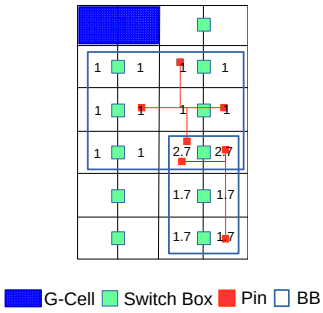


Figure 7: Congestion map example.

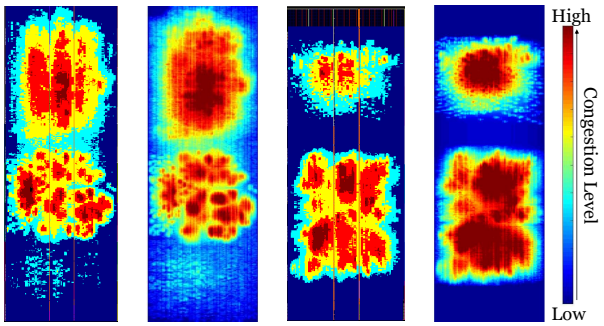


Figure 8: Congestion Maps of e2 and e3 in ISPD2016 contest: (a) actual routing congestion of e2 obtained from Vivado; (b) our estimation of e2; (c) actual routing congestion of e3 obtained from Vivado; (d) our estimation of e3.

form a window, such that putting the cell to any position in the window will result in the same disturbance to the global placement result. HPWL then becomes the prime objective and a legal site for the cell will be found as shown in line 11 to line 19. In our implementation, we assume that the disturbance to global placement caused by a displacement of less than four sites is the same. As this algorithm is highly flexible in choosing the second object (HPWL in ours), it can also be adapted to optimize other objectives like timing, power, etc.

Unlike DSPs and RAMs, the legalization of BLEs is much more complicated. First, we need to check whether a move will violate the complex legalization rules. Secondly, when putting BLEs into CLBs, there are various choices such as whether a BLE should be broken into LUTs and FFs, whether a BLE should be merged, and which BLEs should be merged, etc. To decide whether we should break a BLE back into basic logic elements, we need to consider the trade-off between area utilization and the number of nets need to be routed. In our algorithm, we prefer not to break the BLEs obtained by packing since this not only can reduce the computational resources in legality checking significantly but also will result in less routing congestion. To increase the area utilization while still using BLE as the basic element in legalization, we will greedily merge a BLE to one of the BLEs already in a slice. However, if a net-list has complicated control nets, it is likely to fail in legalization if the BLE cluster is not allowed to be changed. Thus, to ensure that legalization can return a legal result, we allow BLEs to be broken if the design cannot be legalize without breaking BLEs. This situation will rarely happen given the premise that the legality check and the merging algorithm for BLE (*IsLegal* in line 13) has good performance. Actually in the experiment, there are only a few very complicated designs that need to break BLEs back into LUTs and FFs in order to legalize the global placement result.

Algorithm 3 LegalizeWin

Input: Cells, their locations and breakBLE
Output: The legal location of each cell
1: **for all** cell in cells **do**
2: winSize ← 0;
3: isPlace ← false;
4: **while** !isPlace **do**
5: sites ← ∅;
6: **while** sizeof(sites) < 20 **do**
7: sites ← sites $\bigcup$ *GetCandSites*(winSize);
8: winSize ← winSize + 1;
9: **end while**
10: sort sites ascending by HPWL;
11: **for all** site in sites **do**
12: **if** IsLegal(site,cell,breakBLE) **then**
13: $\text{disp}_{\text{cell}}$ ← $\text{GetDisp}_{\text{cell}}$(site);
14: loc_{cell} ← GetLoc(site);
15: isPlace ← true;
16: break;
17: **end if**
18: **end for**
19: **end while**
20: **end for**
21: **return** loc, disp;

3.7 Detailed Placement

In our detailed placement, two procedures, (1) slice swapping and (2) BLE moving , are executed in every iteration to refine the placement result. To optimize HPWL in slice swapping, we move a slice to an empty space in its optimal region if it exists. Otherwise, it will be swapped with a slice in its optimal region if the HPWL is improved after the swap. Similarly in BLE moving, a BLE is moved to a slice in its optimal region if the move satisfies the legalization rules.

However, given a placement, HPWL only considers the wirelength but not the actual routed wirelength which is affected by the routing congestion and the routing resources. In order to better optimize routed wirelength in our detailed placement, HPWL is used as primary objective and routing resources are used as secondary objective.

In our algorithm, we prefer putting BLEs in sites that needs less routing resources (RR). Given a BLE and its position, we can calculate its RR by equation (2), which accumulates all the RR needed to connect this BLE to other connected BLEs.

$$\text{RR}_{\text{BLE}_i}(x,y) = \sum_{c \in C_{\text{BLE}_i}} f(x,y,c), \tag{2}$$

where BLE_i is the BLE to be moved, C_{BLE_i} represents the elements connected to BLE_i, $f(x,y,c)$ is a function that returns how many RR it will use to connect BLE_i and ele-

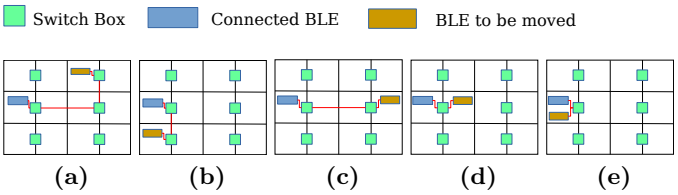


Figure 9: Five kinds of candidate locations for the BLE to be moved: $f(x,y,c) = f_{\text{switchbox}}(x,y,c) + f_{\text{shape}}(x,y,c)$: (a) $4 = 2 + 2$, (b) $3 = 1 + 2$, (c) $2 = 1 + 1$, (d) $1 = 0 + 1$, (e) $0 = 0 + 0$.

ment c if BLE_i is placed at position (x, y).

$$f(x,y,c) = f_{\text{switchbox}}(x,y,c) + f_{\text{shape}}(x,y,c). \quad (3)$$

The value of $f(x,y,c)$ in equation (3) consists of two parts. The first part $f_{\text{switchbox}}(x,y,c)$ returns how many switch boxes are needed to route the net connecting c and BLE_i and it's calculate by the equation below,

$$f_{\text{switchbox}}(x,y,c) = \begin{cases} 0, & \text{if nSwitchbox}= 1, \\ 1, & \text{if nSwitchbox}= 2, \\ 2, & \text{if nSwitchbox}\geq 3, \end{cases}$$

where nSwitchbox denotes the minimum number of switch boxes thatBLE_i needs to used to connect to c if it is placed in (x, y). The other part considers the RR in terms of the shape of the chip. Without loss of generalty, we assume that the height of the chip is larger than its width. Since we prefer our placement to use horizontal routing resources instead of vertical ones and prefer putting as many connected BLEs in the same site as possible, we use the following equation to calculate $f_{\text{shape}}(x,y,c)$,

$$f_{\text{shape}}(x,y,c) = \begin{cases} 0, & \text{if } \text{BLE}_i, c \text{ are on the same site}, \\ 1, & \text{if } \text{BLE}_i, c \text{ are horizontally aligned}, \\ 2, & \text{otherwise}. \end{cases}$$

An example is given in Figure 9. Since we only use one switch box to route the net in Figure 9(d) and 9(e), their values of $f_{\text{switchbox}}(x,y,c)$ equal 0. While in Figure 9(b) and 9(c), two switch boxes are needed, so the values of their $f_{\text{switchbox}}(x,y,c)$ are 1. In Figure 9(a), three switch boxes are needed to route the net, the value of $f_{\text{switchbox}}(x,y,c)$ is 2. Since the two BLEs are placed in the same site in Figure 9(e), the value of $f_{\text{shape}}(x,y,\text{cell}_i)$ is 0. In Figure 9(c) and 9(d), the two BLEs are horizontally aligned, so the values of their $f_{\text{shape}}(x,y,\text{cell}_i)$ are one. In Figure 9(a) and 9(b), the BLEs are neither in the same site nor horizontally aligned, so the values of their $f_{\text{shape}}(x,y,\text{cell}_i)$ equal two. We then add $f_{\text{shape}}(x,y,\text{cell}_i)$ and $f_{\text{switchbox}}(x,y,\text{cell}_i)$ up to get the resulting RR.

4. EXPERIMENTAL RESULTS

To evaluate our proposed method, the algorithms are implemented in C++. The experiments were performed on a 64-bit Linux workstation with Intel Xeon 3.7GHz CPU and 16GB memory, using the benchmarks provided by ISPD2016 Routability-Driven FPGA Placement Contest [18]. Table 2 shows the statistics of the benchmarks. We compare the quality of our placement algorithm with the placers of the wining teams in the ISPD2016 contest.

Table 2: Statistics of the ISPD2016 contest benchmark.

Design	#LUT	#FF	#RAM	#DSP	#Ctrl set
FPGA-1	50K	55K	0	0	12
FPGA-2	100K	66K	100	100	121
FPGA-3	250K	170K	600	500	1281
FPGA-4	250K	172K	600	500	1281
FPGA-5	250K	174K	600	500	1281
FPGA-6	350K	352K	1000	600	2541
FPGA-7	350K	355K	1000	600	2541
FPGA-8	500K	216K	600	500	1281
FPGA-9	500K	366K	1000	600	2541
FPGA-10	350K	600K	1000	600	2541
FPGA-11	480K	363K	1000	400	2091
FPGA-12	500K	600K	600	500	1281
resource	526K	1100K	1600	770	-

The last row gives the total available resources on the FPGA.

In our experiments, we use the architecture of the commercial device family, Xilinx UltraScale [1], for our comparative study. The device aspect ratio and average spacing between blocks were determined based on the Xilinx UltraScale VU095, the latest 20nm FPGA chip. For fair comparison, all the placers were executed with single thread. We set Vivado using the same configurations in ISPD2016 contest, which uses two threads in routing and limits the running time to be 12 hours.

Table 4 shows our routing wirelength and running time compared to the winners of ISPD2016 contest. Since we cannot get the executables of the placers of the first and third place winners, both the running time and routed wirelength results of those two placers come from the ISPD2016 contest organizer. Note that since our computer configuration is different from that used in the contest, the running time is just listed for reference and cannot be compared directly. From the result, we can see that, our placer show good performance compared to the other three and can successfully generate a routable placement for all test cases.

The results in Table 3 show that the partitioning based pre-processing step mentioned in Section 3.2 and the congestion-driven global placement step described in Section 3.4.2 perform well in reducing congestion.

Table 3: Comparisons of routed wirelength between applying different methods

Design	raw	cong_gp	partition	both
FPGA-1	**362563**	364143	378029	377883
FPGA-2	681418	**677563**	696417	689360
FPGA-3	4027586	3999517	3645846	**3617466**
FPGA-4	**6037293**	6087199	6265158	6357766
FPGA-5	-	-	-	**10455204**
FPGA-6	7801736	7723476	7016684	**6960037**
FPGA-7	**10248020**	10615672	10338763	10580828
FPGA-8	9171179	9392039	**8874454**	9013564
FPGA-9	**12954350**	13437554	-	13834692
FPGA-10	-	10372369	8782789	**8564363**
FPGA-11	-	-	**11226088**	11688504
FPGA-12	-	10286583	-	**8928528**

In the second column, "raw" represents the flow without using partition (Section 3.2) and cong_gp (Section 3.4.2). The third and fourth column give the routed wirelength of applying only one of the methods to "raw" respectively. The last column gives the routed wirelength of applying both cong_gp and partition to "raw".

5. CONCLUSION

Although the traditional simulated-annealing based FPGA placement algorithm can achieve high-quality result, the scalability problem becomes unacceptable as the circuits grow larger. Analytical placement algorithm for FPGA becomes

Table 4: Placement quality comparison with ISPD2016 contest winners

Design	RippleFPGA		1st Place		2nd Place		3rd Place	
	WL	TIME(s)	WL	TIME(s)	WL	TIME(s)	WL	TIME(s)
FPGA-1	362563	74	-	-	379932	118	581975	97
FPGA-2	677563	167	677877	435	679878	208	1046859	191
FPGA-3	3617466	1037	3223042	1527	3660659	1159	5029157	862
FPGA-4	6037293	621	5628519	1257	6497023	1449	7247233	889
FPGA-5	10455204	1012	10264769	1266	-	-	-	-
FPGA-6	6960037	2772	6330179	2920	7008525	4166	6822707	8613
FPGA-7	10248020	2170	10236827	2703	10415871	4572	10973376	9196
FPGA-8	8874454	1426	8384338	2645	8986361	2942	12299898	2741
FPGA-9	12954350	2683	-	-	13908997	5833	-	-
FPGA-10	8564363	5555	-	-	-	-	-	-
FPGA-11	11226088	3636	11091383	3227	11713479	7331	-	-
FPGA-12	8928528	9748	9021768	4539	-	-	-	-

The computer used for RippleFPGA is a Linux workstation with 3.7GHz CPU and 16GB memory. The computer used for the three contest placers is a Linux Machine with 2.7GHz CPU and 16GB memory.

the main stream of FPGA placement because it can obtain competitive result with significantly shorter running time compared to other approaches. Moreover, routability has become a main issue in placement when the design complexity grows higher. This paper has proposed an efficient and effective routability-driven analytical placement for large-scale heterogeneous FPGAs. The proposed algorithm consists of five stages: (1) partitioning, (2) packing, (3) global placement with congestion estimation, (4) legalization, and (5) routing-resources-aware detailed placement. The experimental results show that our placement algorithm can achieve good quality and running time compared to the state-of-the-art academic placers.

6. REFERENCES

[1] Xilinx inc. http://www.xilinx.com.

[2] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI Journal*, 19(1):1–81, 1995.

[3] H. Bian, A. C. Ling, A. Choong, and J. Zhu. Towards scalable placement for FPGAs. In *ACM Symposium on FPGAs*, pages 147–156. ACM, 2010.

[4] D. T. Chen, K. Vorwerk, and A. Kennings. Improving timing-driven FPGA packing with physical information. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 117–123. IEEE, 2007.

[5] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 647–654. IEEE Press, 2014.

[6] W.-K. Chow, J. Kuang, X. He, W. Cai, and E. F. Y. Young. Cell density-driven detailed placement with displacement constraint. In *ACM International Symposium on Physical Design (ISPD)*, pages 3–10. ACM, 2014.

[7] M. Gort and J. H. Anderson. Analytical placement for heterogeneous FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 143–150. IEEE, 2012.

[8] X. He, T. Huang, W.-K. Chow, J. Kuang, K.-C. Lam, W. Cai, and E. F. Y. Young. Ripple 2.0: high quality routability-driven placement via global router integration. In *ACM/IEEE Design Automation Conference (DAC)*, pages 152:1–152:6. IEEE, 2013.

[9] M.-C. Kim, D.-J. Lee, and I. L. Markov. SimPL: An effective placement algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(1):50–60, 2012.

[10] T.-H. Lin, P. Banerjee, and Y.-W. Chang. An efficient and effective analytical placer for FPGAs. In *ACM/IEEE Design Automation Conference (DAC)*, page 10. ACM, 2013.

[11] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014.

[12] P. Maidee, C. Ababei, and K. Bazargan. Timing-driven partitioning-based placement for island style FPGAs. 24(3):395–406, 2005.

[13] P. Spindler, U. Schlichtmann, and F. M. Johannes. Abacus: fast legalization of standard cell circuits with minimal movement. In *Proceedings of the 2008 international symposium on Physical design*, pages 47–53. ACM, 2008.

[14] P. Spindler, U. Schlichtmann, and F. M. Johannes. Kraftwerk2: A fast force-directed quadratic placement approach using an accurate net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(8):1398–1411, 2008.

[15] D. Xie, J. Xu, and J. Lai. A new FPGA placement algorithm for heterogeneous resources. In *International Conference on ASIC (ASICON)*, pages 742–746. IEEE, 2009.

[16] M. Xu, G. Gréwal, and S. Areibi. StarPlace: A new analytic method for FPGA placement. *Integration, the VLSI Journal*, 44(3):192–204, 2011.

[17] Y. Xu and M. A. Khalid. QPF: efficient quadratic placement for FPGAs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 555–558. IEEE, 2005.

[18] S. Yang, A. Gayasen, C. Mulpuri, S. Reddy, and R. Aggarwal. Routability-driven FPGA placement contest. In *Proceedings of the 2016 on International Symposium on Physical Design*, pages 139–143. ACM, 2016.

[19] Y. Zhuo, H. Li, and S. P. Mohanty. A congestion driven placement algorithm for FPGA synthesis. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2006.