PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems

Tuan D. A. Nguyen
National University of Singapore
Department of Electrical & Computer Engineering, Faculty of Engineering
4 Engineering Drive 3, Singapore 117583
tuann@u.nus.edu

Akash Kumar
Technische Universität Dresden
Center for Advancing Electronics Dresden
(cfaed) Chair of Processor Design
Würzburger Str. 46, Dresden, Germany 01187
Akash.kumar@tu-dresden.de

ABSTRACT

Partial reconfiguration (PR) is gaining more attention from the research community because of its flexibility in dynamically changing some parts of the system at runtime. However, the current PR tools need the designer's involvement in manually specifying the shapes and locations for the PR regions (PRRs). It requires not only deep knowledge of the FPGA device, the system architecture, but also many trial-and-error attempts to find the best-possible floorplan. Therefore, many research works have been conducted to propose automatic floorplanners for PR systems. However, one of the most significant limitations of those works is that they only consider the PRRs and ignore all other static modules. In this paper, we propose a novel PR floorplanner called PRFloor. It takes into account all components in the system. The main ideas behind PRFloor are the unique *recursive pseudo-bipartitioning* heuristic using a new, simple, yet effective Nonlinear Integer Programming-based bipartitioner. The PRFloor performs very well in the experiments with various synthetic PR system setups with up to *130* modules, *24* PRRs and *85%* of the FPGA resource. The average maximum clock frequency obtained for the actual PR systems implemented using PRFloor is even *3%* higher than the similar systems without PR capability.

Keywords

partial reconfiguration; FPGA floorplan; bipartition; NLP

1. INTRODUCTION

Nowadays, the number of applications that must be incorporated into a single FPGA-based system is increasing rapidly which requires more hardware resources. One of the solutions is to increase the size of the chips. This is not an efficient or scalable solution because of the size and power consumption constraints. Consequently, dynamic partial reconfiguration [25] is gaining special interests from the research community. Nevertheless, it is not trivial to successfully implement these kinds of system. One of the reasons is the limitation of the current PR-supported EDA tools, such as Xilinx PlanAhead and Vivado. The designers have to specify the shapes and locations (hereafter called *placements* for simplicity) for all PR regions (PRRs) manually. Moreover, to make a good floorplan, it is essential to plan the layouts of **all** modules in the system with respect to the connections between them and the resource requirements. This process requires expertise in FPGA architecture, the knowledge of the connections between components and many trial-and-error attempts to find the best possible floorplan. Additionally, since the PR systems are getting more complicated with hundreds of components [8,19], it is almost impossible to do the work by hand. Hence, having an automatic floorplanner for PR FPGA-based design is imperative.

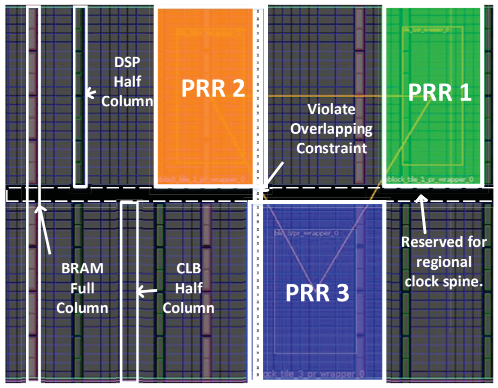

Figure 1: The resources in one clock region of Xilinx Virtex-6 XC6VLX240T. These are BRAM, DSP and CLB, distributed in a columnar fashion. There are 3 rectangular PRRs placed at different locations in the same clock region. The placement of PRR 2 and PRR 3 violates the PR constraint.

Research on floorplanning for VLSI design has been underway since long time ago with many sophisticated advancements [13]. Some well-known methods introduced in [13] are Stockmeyer, Normalized Polish expression, sequence pair, recursive partitioning, etc. However, floorplanning in FPGA design has different challenges and restrictions, especially in state-of-the-art FPGAs. These devices have a variety of resources: Configuration Logic Block (CLB), Block RAM (BRAM), Digital Signal Processing (DSP), PCIe, GTX and so on. All of these resources are predefined, already placed and non-uniformly distributed on the FPGA fabric. They

FPGA'16, February 21-23, 2016, Monterey, CA, USA
© 2016 ACM. ISBN 978-1-4503-3856-1/16/02...$15.00
DOI: http://dx.doi.org/10.1145/2847263.2847270

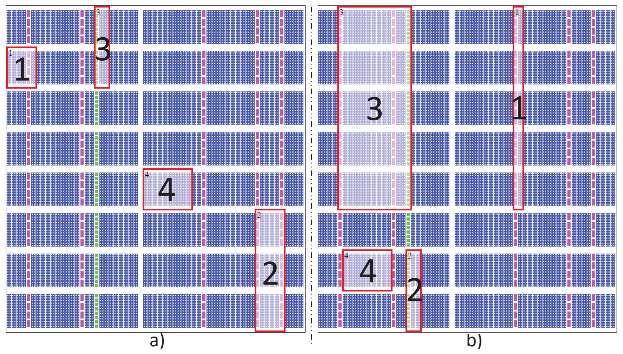

Figure 2: The floorplans given by the tool [20] at *floorplacer.necst.it*. The columns in blue, red and green are CLBs, BRAMs and DSPs, respectively. The weight costs for CLB, BRAM and DSP are 1, 12 and 60. The red boxes represent PRRs.

are also arranged in a columnar-fashion as can be seen in Fig. 1. Thus, the aforementioned techniques cannot be applied directly to FPGA. Moreover, in FPGA-based PR systems, the following constraints must be satisfied. **(1)** There is at *most* one PRR in any column in one clock region [25] as shown in Fig. 1. **(2)** In Xilinx PlanAhead, the proxy logics [25] are used for the input/output signals to/from PRRs. Therefore, the number of CLBs along the edges of the PRRs should be large enough to avoid routing congestion and to enhance the timing of the design.

In fact, many research works have been conducted to propose such floorplanners [3,6,14,15,17,20,24]. But none of those considers the possible placements for all components in the system with different constraints for static modules and PRRs. They only analyze the connections between PRRs which, in some cases, may not have any direct link in the systems with Network-on-Chip (NoC) [8,19]. Even worse, they overlook the resource requirements of the static modules in the system. It leads to the situation where there are not sufficient resources for static modules. These problems are illustrated in Fig. 2. In Fig. 2a, Region 1 and 2 are communicating indirectly via NoC. Therefore, it would be better if they are placed close to each other to facilitate the place and route (PnR) process in placing the NoC. In Fig. 2b, Region 3 requires only 50 CLBs, 40 BRAMs and 20 DSPs. However, the placement for that region occupies 40 DSPs. Consequently, it is not possible to implement static modules if they require 10 DSPs since there are only 8 blocks left. The authors in [3] were aware of this problem but they suggested treating static modules similar to PRRs which puts unnecessary PR constraints on them. Alternatively, the work described in [2] does floorplan both static modules and PRR but they only support one PRR. As a result, these floorplanners are only suitable to the systems with no or small number of static modules and PRRs.

Besides, one of the most widely used methods in VLSI floorplanning is recursive cut-size driven multilevel netlist bipartitioning [5,13,26]. It reduces the problem size through bipartitioning and finds the appropriate relative locations of the modules in the system. These methods are only applicable for *homogeneous* FPGA as contended in [16,21]. Although [16,21] and even [11] support multi-resource aware bipartitioning, they do not anticipate an important issue: the resources occupied by the placements of module in two

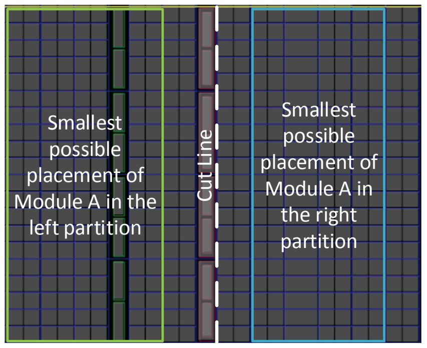
Smallest possible placement of Module A in the left partition
Cut Line
Smallest possible placement of Module A in the right partition

Figure 3: The two possible placements of Module A in two partitions. A requires 4 CLB columns which can be fully satisfied in the right partition. But in the left partition, the placement occupies an extra BRAM column. The bipartitioner may end up assigning A to the left partition because it does not consider the actual resources occupation.

partitions may not be the same. It is because of the non-uniform distribution of FPGA resources as illustrated via an example in Fig. 3. Thus, estimating module resources based solely on the synthesized netlist is not accurate.

Contribution: In this work, all of the above issues are addressed by our novel floorplanner for PR systems, PRFloor. Our contribution is twofold.

- We propose a unique *recursive pseudo-bipartitioning* heuristics using a new, simple, yet effective Nonlinear Integer Programming (NLP) [12] bipartitioner. The NLP bipartitioner supports heterogeneous FPGA. Besides, the resources occupied by the modules in different partitions can be different.

- PRFloor finds the placements for PRRs in the system considering not only the connections between them but also between the static modules and the resources requirements of all modules.

The experiments are carried out using the set of PR systems proposed by [19] with up to *24* PRRs. The largest FPGA utilization is *85%*. The numbers of modules (including PRRs) in the experimental systems are from *99* to *130*. These systems are significantly larger than the experiments reported in most of the PR floorplanners in literature. The longest time taken by the PRFloor is less than *9* minutes for the systems with 24 PRRs. Interestingly, by using the PRFloor, the average maximum clock frequency obtained for the synthesizable PR systems is even *3%* higher than the comparable systems without PR capability. Given the fact that the PR systems usually have lower clock frequency than the similar static ones; this result clearly shows how effective our PRFloor is in producing high quality floorplans. The NLP-based bipartitioner also produces good results. It reduces the average cut-size by *15%* as compared to the state-of-the-art bipartitioner [11] in a set of random systems with up to *300* modules.

The remaining paper is organized as follows. The recent PR floorplanners are discussed in Section 2. The proposed algorithm is presented in Section 3, followed by experimental results in Section 4. Finally, the conclusions and future works are presented in Section 5.

2. RELATED WORK

The general floorplanners for FPGAs have been addressed extensively in classical works such as [1,4,7,22]. Nonetheless, we only consider the ones that support the recent heterogeneous FPGA and are compliant with PR constraints [2,3,6,14,15,20,24]. The application-specific PR-supported floorplanner proposed in [17] is not covered here because it is only applicable to their pipeline architecture.

The Floorplacer in [14] is based on Simulated Annealing (SA). The SA places the PRRs from the bottom-left of FPGA towards the upper-right and tries to minimize the resource wastage of these PRRs. The Floorplacer expects that the remaining area would be feasible for static modules. The communication aspects of the system are ignored. The authors further improved the method in [15]. They added the *distance to corresponding Input/Output Blocks* minimization objective to the SA algorithm. However, the problem of static modules is still not addressed properly.

Another SA method proposed by Bolchini et al. [3] is based on the sequence pair representation. The method has some improvements in local search and violation constraints to speed up the process. The approach does take the connections between PRRs and static modules into account if needed. However, the method handles PRRs and static modules in the same manner which imposes unnecessary strict PR constraints to the regions dedicated to static modules. For instance, in their approach, the PRRs have to span the whole clock regions. However, this condition is not necessary for static modules. As a result, the floorplanner will be more likely to fail to place the PRRs when the resource utilization is high.

The floorplanner suggested in [24] uses a greedy approach called Columnar Kernel Tessellation. For each PRR, all the kernels are generated from the available resources of the FPGA. The kernels are then replicated in the columnar fashion to satisfy the requirements of the PRR. The smallest kernel is chosen for that PRR and the next PRR is processed. The procedure is repeated several times with different initial kernels to find the best floorplan. The post process procedure is executed after the above operation to further optimize the wirelength by moving or swapping the PRRs vertically. Similarly, Duhem et al. [6] suggest an exhaustive method to find all possible placements for each PRR. The resulting regions are then sorted with respect to the cost of the shapes and resource wastages. After that, the regions are greedily selected for each PRR considering non-overlapping constraint and the threshold for the physical distances between *only* PRRs. Again, these approaches do not consider the static modules at all.

The only approach that takes into account both static and PR modules (PRMs) is proposed in [2]. They suggested a three-stage process. First, it creates a dummy module which is the union of all PRMs that are mapped to one PRR. Then, the PnR is run for the new non-PR design consisting of this dummy module and the other static modules. The center of mass of this dummy module is identified. Finally, a placement which is closest to that center is selected among all possible placements of the PRR. For more than one PRR, the authors suggest executing the above process iteratively for each PRR but it is left for future work.

The most recent floorplanner is proposed in [20]. It is based on Mixed-Integer Linear Programming (MILP). The analytical model formulated only for the set of PRRs with respect to their resource requirements and connectivities. Since the MILP takes into account the global search space, it is proven to provide better floorplans compared with the previous works [3,24]. Nevertheless, this method overlooks the static modules as illustrated in Fig. 2.

As can be seen from the above literature review, except the work [2], there is no floorplanner that considers all modules in the PR system therefore they all suffer from the problems discussed in Section 1.

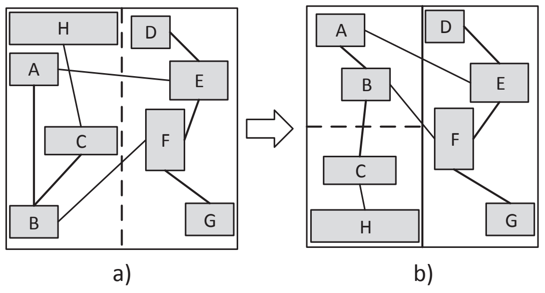

Figure 4: The cut-size driven recursive bipartitioning. The connections between modules are illustrated by the solid lines. In *a*)**, the** ***vertical cut*** **(dashed line) is done to vertically separate the region into two halves. Similarly, in** *b*)**, the next** ***horizontal cut*** **(dashed line) is performed for the newly created partition on the left.**

3. PROPOSED APPROACH

3.1 NLP-based Bipartitioner

3.1.1 *Recursive bipartitioning*

Recursive cut-size driven multilevel netlist bipartitioning is used widely in VLSI floorplanning [5,13,26]. The basic idea of the cut-size driven recursive bipartitioning is shown in Fig. 4. In general, the cuts are recursively performed to divide the circuit into two partitions such that the cut line crosses the least amount of wires while balancing the weights of modules in two new partitions. The order of the cuts, i.e *vertical* and *horizontal cuts*, is not known a priori and it must be decided by the algorithm based on the analysis of the circuit. Nonetheless, further details of this problem are beyond the scope of this paper; the reader can refer to [5,13,26] for more information.

3.1.2 *The proposed bipartitioner*

As discussed in Section 1, the prior approaches cannot be applied directly to FPGA because the FPGA resources are predefined and placed in specific locations. Additionally, none of the current multi-resource aware partitioners [11,16,21] can provide a solution for the resource occupation issue. That is the resources taken by the possible placements of one module in two partitions can be different. Therefore, we propose a novel bipartitioner that is capable of **(1)** minimizing the number of nets crossing two partitions and **(2)** supporting multi-resource bipartitioning in which:

- The available resources, type and quantity, in two partitions can be different.
- The resources occupied by the possible placements of **one** module in **two** partitions can be different.

- The resources occupied by the modules in two partitions can be balanced individually with respect to each type of resource.

The partitioning algorithms presented in [16, 21] support the hypergraph representation of the connections between modules. Our bipartitioner and [11] work on the graph representation instead. Our method transforms a group of nodes connected by one particular hyperedge to a *fully connected* graph. Even though using hypergraph is more straight forward to represent the connections between modules in a circuit, in practice, converting them to graph has minor effect on the quality of the bipartitioning solutions. The correctness of the bipartitioning solutions is not affected. The only concern is the extra memory used to store the graph. However, using graph makes it easier to construct the NLP model for our bipartitioner.

In our bipartitioner, the partitioning problem is modeled as an NLP optimization problem [12]. The binary variable m_i, of which value is either 0 or 1, represents the partition assignment for module i. The objective function is the total number of nets between modules that cross two partitions (this number is *cut-size*). Equations 1 and 2 show how to calculate the cut-size $nets_{ij}$ between two modules i and j which are connected by n_{ij} wires. Intuitively, $nets_{ij} = n_{ij}$ when $m_i \neq m_j$.

$$\begin{aligned} nets_{ij} &= n_{ij} * (1 - m_i) * m_j + n_{ij} * m_i * (1 - m_j) && (1)\\ &= n_{ij} * (m_i + m_j - 2 * m_i * m_j) && (2) \end{aligned}$$

The objective function of the NLP program is the summation of all $nets_{ij}$. This function (derived from Equation 2) is presented in Equation 3 followed by the constraints on the resources occupation in Equations 4, 5, 6 and 7.

Objective:

$$\sum_i m_i * \sum_{j \neq i} n_{ij} - 2 * \sum_i m_i * \sum_{j \neq i} (n_{ij} * m_j) \quad (3)$$

Subject to:

$$Total0_{CLB} = \sum_i ((1 - m_i) * CLB0_i) <= MAX0_{CLB} \quad (4)$$

$$Total1_{CLB} = \sum_i (m_i * CLB1_i) <= MAX1_{CLB} \quad (5)$$

$$ub_{lower} * Total0_{CLB} <= ub_{upper} * Total1_{CLB} \quad (6)$$

$$ub_{lower} * Total1_{CLB} <= ub_{upper} * Total0_{CLB} \quad (7)$$

The objective is to minimize Equation 3 while satisfying the maximum resource constraints. That is the total resources of the modules assigned to one partition should not exceed the available resources of that partition. The constraints on the total number of CLBs occupied by the modules assigned to partition 0, $Total0_{CLB}$, and partition 1, $Total1_{CLB}$, are shown in Equations 4 and 5. In the equations, $MAX0_{CLB}$ and $MAX1_{CLB}$ represent the numbers of CLBs available in partition 0 and 1 respectively. The possible numbers of CLBs occupied by a module i in either partition are $CLB0_i$ and $CLB1_i$.

Another constraint is to balance the resources in two partitions by the unbalance factor, ub, similar to [11]. This factor controls the ratio of the resources occupied by the modules in two partitions. This ratio must be within the range $(0.5 - ub, 0.5 + ub)$. Let $ub_{lower} = 0.5 - ub$ and $ub_{upper} = 0.5 + ub$. The unbalance constraints for the CLB resource are illustrated in Equations 6 and 7.

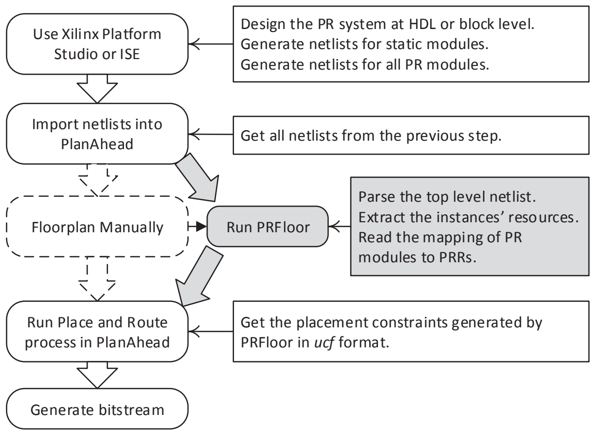
Use Xilinx Platform Studio or ISE
Design the PR system at HDL or block level. Generate netlists for static modules. Generate netlists for all PR modules.
Import netlists into PlanAhead
Get all netlists from the previous step.
Floorplan Manually
Run PRFloor
Parse the top level netlist. Extract the instances' resources. Read the mapping of PR modules to PRRs.
Run Place and Route process in PlanAhead
Get the placement constraints generated by PRFloor in ucf format.
Generate bitstream

Figure 5: The new design flow with PRFloor.

Similar constructs of the above constraints are done for the remaining resources such as DSP and BRAM. The NLP program is then solved by the Gurobi Optimization tool [9]. The experiment results of the proposed bipartitioner is provided in Section 4.1.

3.2 The PRFloor

3.2.1 The Design Flow

Fig. 5 shows the flow of designing a PR system as suggested by [25]. However, the manual floorplanning step is replaced by the separate execution of the PRFloor outside of PlanAhead. The PRFloor reads the top level netlist of the design to get the information of all modules in the system and the connections between them. Thereafter, the resources of each module are parsed from its netlist. The *total resources required for one PRR* is the **union** of the resources of the PRMs that are mapped to that PRR. In this work, it is assumed that the mappings of PRMs to PRRs are already carried out. In addition, the PRMs which are mapped to one particular PRR are also implemented with a common interface wrapper [25]. Finally, PRFloor executes the *recursive pseudo-bipartitioning heuristic* to floorplan all modules in the design. The placement constraints for PRRs is written in the *ucf* format. Hereafter, the PRR is considered as generic module with implied PR constraints.

3.2.2 Overview of the PRFloor

All the steps executed inside the *PRFloor* are summarized in Algorithm 1. It starts by building the model of the FPGA device used in the design. The empty ROOT partition which is an entire FPGA is created. A list of static and PR modules is parsed from the netlist. All possible placements for each module are computed. After that, the recursive pseudo-bipartitioning process is executed to determine the preferred position for each module. The placements are then heuristically filtered and sorted. Finally, the feasible combination of them is found using the recursive trial-and-error algorithm. The details of the aforementioned major steps are discussed in the subsequent sections.

3.2.3 The FPGA Model

In this section, the architecture of the FPGA is analyzed and the representations of the FPGA resources are simplified to ease the search process in the algorithm. The FPGA

Algorithm 1 PRFloor

```
1: Build the FPGA model {Section 3.2.3}
2: Create ROOT partition containing all the modules and the
   current bounding box is the entire FPGA
3: Find all possible placements {Section 3.2.4}
4: while NOT SUCCESS do
5:     Do recursive vertical cut for ROOT {Section 3.2.5}
6:     Do recursive horizontal cut for ROOT {Section 3.2.5}
7:     Calculate the normalized wastage and distances to anchor
       points for all placements
8:     Select the placement candidates {Section 3.2.6}
9:     Sort the placements of each module in the increasing order
       of their $OBJ_{placement}$ values {Section 3.2.6}
10:    Sort the modules in the decreasing order of the resource
11:    Find the feasible floorplan {Section 3.2.7}
12:    if NOT SUCCESS then
13:        Shift the first vertical cut line to the right
14:    end if
15: end while
16: return final placements
```

considered in this paper is Xilinx Virtex-6 XC6VLX240T (hereafter, called Virtex-6). However, the method is general enough that it can be extended to support other devices such as Virtex-4, Virtex-5 and Virtex-7.

The Virtex-6 is divided into 6-by-2 clock regions. The clock region is further divided into roughly 50 columns (depending on the region on the left or the right side of the FPGA), each of them can be CLB, BRAM, DSP or special resources such as IOB, PCIe, GTX or even empty. The resources in each clock region do not need to be the same across the entire FPGA. Also noted that there is a blank space in the middle of the clock region, as can be seen in Fig. 1. It is reserved for the regional clock spine, separating the columns into two halves. Each half of CLB, BRAM and DSP column consists of 20 CLBs, 4 BRAMs (if configured as 18-bit-width RAM, there are 8 blocks) and 8 DSP blocks respectively. The smallest addressable *configuration frame* spans the entire column, which is equivalent to the 40-CLB height. This is the reason why PRR-2 and PRR-3 violate the overlapping constraint since one frame cannot contain configuration bits for more than one PRR.

In our approach, instead of looking at the fine-grain granularity of resources, the *half-column* resource is preferred. This half-column granularity balances the trade-off between the number of configuration frames and the occupied resources. For instance, the PRR-1 in Fig. 1 requires 8 DSP blocks. If the full-column granularity is used as suggested by [20, 24], we have to assign the entire DSP column containing 16 DSP blocks to it. The costs saved and the configuration time overhead by using half-column granularity are discussed in Section 4.2. Our method uses an accurate half-column model of the Virtex-6, including the PCIe, GTX and other resources. The wastage cost of each placement is also calculated similar to the related works [20, 24]. Each resource is given a weight based on its scarcity in the FPGA and the cost is the *weighted sum* of those resources. The connections of the modules to the external IO pins are not considered in this paper. However, it does not affect the general idea of the algorithm because the IO pins can be handled as generic fixed-location modules.

3.2.4 Find the possible placements

Since the actual occupied resources as well as the position of the placements of the module cannot be predicted as discussed in Fig. 3, all possible rectangle-shape placements

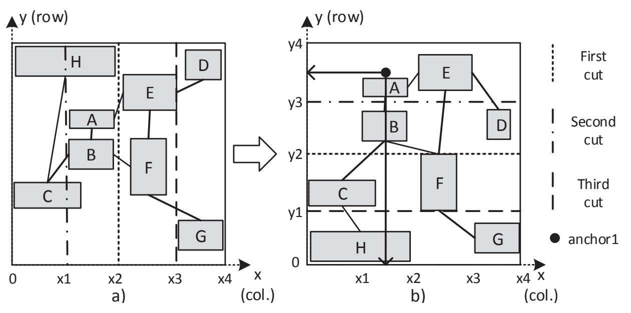

Figure 6: The pseudo vertical cuts (a) and horizontal cuts (b) used to estimate the preferred anchor points of the modules. The vertical cuts are performed first to scatter the modules horizontally across the FPGA. In a, the first cut separates the entire FPGA into two halves, the second cut is applied to the partition on the left (from 0 to $x2$), and so on. Likewise, in b, the horizontal cuts are done to spread the modules vertically.

of each module (subjected to PR constraints described in Section 1 if it is PRR) across the aforementioned simplified FPGA model are generated. It is possible to generate L-shape placements. However, these shapes may hinder the place and route process from satisfying the timing constraints of the whole system. Therefore, rectangle-shape is preferred. All placements that overlap with user-defined or hard-macro regions are not entertained.

3.2.5 The Recursive Pseudo-bipartitioning Heuristic

The difficulty in using the cut-size driven recursive bipartitioning for FPGA is that the resources are limited and their locations are predefined. It is not possible to strictly determine a region for a module before having knowledge of the resources under that region. Therefore, we propose the *recursive pseudo-bipartitioning heuristic* as the solution for this problem. That is, except the *first vertical cut* which is actually used to partition the circuit, the *subsequent cuts* are *pseudo*. These vertical and horizontal pseudo cuts are done independently without back-tracing. The objective is to **(1)** scatter the modules across the FPGA device as evenly as possible. **(2)** The cut-size at every cut is minimized. **(3)** The newly created partitions must have sufficient resources to accommodate the modules. These cuts are performed to have the global view of the preferred positions, or *anchors*, of all modules with respect to the FPGA fabric. The cuts are illustrated in Fig. 6. The anchor point is represented by the (x, y) pair corresponding to $(column, row)$ axes of the FPGA. It is calculated based on the final partition where the corresponding module is assigned to as shown in Fig. 6. Equations 8 is used to compute the (x, y) coordinate of the anchor $anchor1$ of Module A. The *recursive pseudo-bipartitioning heuristic* is presented in the Algorithm 2.

$$anchorA_x = (x1 + x2)/2; \; anchorA_y = (y3 + y4)/2 \quad (8)$$

The modules may not fit entirely in the assigned partition (Module C in Fig. 6a). In FPGA floorplanning, the shape of a placement cannot be freely adjusted by changing the ratio of its edges to fit in the partition because it depends on the locations of the FPGA resources. Therefore, the pseudo cuts are relaxed in such a way that a placement only belongs to one partition if *at least* 90% of its area is inside that

Algorithm 2 Recursive Pseudo-bipartitioning Heuristic

```
Require: $parent\_partition \neq \emptyset$ and valid $cut\_line$ and cut_type

1: create $partition0 \leftarrow \emptyset$ from $cut\_line$ and $parent\_partition$
2: create $partition1 \leftarrow \emptyset$ from $cut\_line$ and $parent\_partition$
3: $list\_mod \leftarrow \emptyset$ {list of modules used for bipartition process}
4: $area\_constraint \leftarrow 90\%$
5: if cut_type = vertical cut and first cut then
6:     $area\_constraint \leftarrow 100\%$
7: end if
8: for all module $\in$ parent_partition do
9:     $list\_placement0 \leftarrow \emptyset$; $list\_placement1 \leftarrow \emptyset$
10:    for all placement of module $\in$ parent_partition do
11:        $area\_ratio0 \leftarrow area\_of\_placement \cap partition0$
12:        $area\_ratio1 \leftarrow area\_of\_placement \cap partition1$
13:        if $area\_ratio0 \geq area\_constraint$ then
14:            add placement to $list\_placement0$
15:        else if $area\_ratio1 \geq area\_constraint$ then
16:            add placement to $list\_placement1$
17:        end if
18:    end for
19:    if $list\_placement0 = \emptyset$ and $list\_placement1 = \emptyset$ then
20:        deduct the resource of $module$ from the total resource
           of $partition0$ and $partition1$
21:    else
22:        $resource0 \leftarrow estimate\_resources(list\_placement0)$
23:        $resource1 \leftarrow estimate\_resources(list\_placement1)$
24:        add $module \rightarrow list\_mod$
25:    end if
26: end for
27: run $NLP\_Bipartitioner(list\_mod, partition0, partition1)$
28: if SUCCESS then
29:     update anchor points of modules
30:     execute Recursive Pseudo-bipartitioning for partition0
31:     execute Recursive Pseudo-bipartitioning for partition1
32: end if
```

partition (this requirement is drawn empirically from our experiments). If all possible placements of a module do not belong to any partition due to that area constraint, that module will not be considered for the subsequent cuts as the case of Module H in Fig. 6a. Its resources will be deducted from the corresponding partitions. This process is described between lines 8 and 20 in Algorithm 2. The anchor point is still determined based on the partition that it is previously assigned to as shown in Equation 9.

$$anchorH_x = (0 + x2)/2;\ anchorH_y = (0 + y1)/2 \qquad (9)$$

During the recursive bipartitioning process, all possible placements of each module in each interim partition are filtered based on the 90%-area constraint. The resources occupied by each module in two partitions are then estimated (lines $22 - 23$ in Algorithm 2). For each of two sets of these placements, the *arithmetic mean* $\bar{x}$, median $\tilde{x}$ and *standard deviation* σ_x of each type of the occupied resources (CLB, DSP, BRAM) are calculated. Algorithm 3 shows how the resource is estimated. The *minimum_occupied* is the smallest possible number of resources occupied by the module. This calculation makes sure that the estimated resources will not be skewed by the unusually large placements.

However, this 90% area constraint is not applicable for the *first vertical cut* as mentioned before (lines $4 - 7$ in Algorithm 2). The placements must be completely inside either of the partitions. This is to reduce the problem size without compromising the quality of the floorplan. The reason is that the initial *ROOT* partition, which is the entire FPGA, is bigger than most modules; dividing it by half should not cause any difficulty in finding good placements for modules.

Algorithm 3 Estimate Occupied Resources

```
1: $a = \bar{x}$
2: if $a > \tilde{x}$ then
3:     $a = \tilde{x}$
4: end if
5: $result = a - 1.5 * \sigma_x$
6: if $result < minimum\_occupied$ then
7:     $result = minimum\_occupied$
8: end if
9: return $result$
```

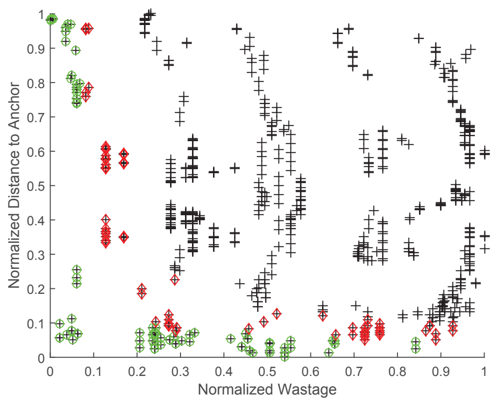

Figure 7: The Pareto-ranking selection technique for the placement candidates of one module. This is the real data obtained from the experiments. The ones in green circle are the first-rank points. The other points in red diamond are the second-rank.

Besides, our algorithm will shift the first vertical cut line along the x-axis to enlarge one of the partitions if some of the modules are too big to fit in neither half of the FPGA.

3.2.6 Select the placement candidates

At this stage, the *anchor points* of all modules are identified. The *geometric* distances from them to the center points of the placements of the corresponding modules are calculated. These distances are then normalized to the largest distance from the placements of the module. Similarly, the wastage cost of these placements are normalized. Then for each module, the Pareto-ranking selection technique [18] is performed for all placements. Only the first and second-rank points are selected for the next steps as presented in Fig. 7.

In PRFloor, the objective of the recursive bipartitioning process is to minimize the cut-size and the connected modules tend to stay closer to each other. The wirelength between connected modules is optimized indirectly via the distance to the anchor point metric. Therefore, when the designer specifies the preferences between the *total wirelength* and the *resource wastage* using the 2-tuple weight (α, β), it can be applied directly to the distance to the anchor point and the resource wastage. Then, the selected placements of each module are sorted in the increasing order of the objective values calculated using the Equation 10.

$$OBJ_{placement} = \alpha * wastage + \beta * dist\_to\_anchor \qquad (10)$$

3.2.7 Find the feasible floorplan

At this step, the number of possible placements for each module is reduced significantly thanks to the selections done in Section 3.2.6. The modules are then sorted in the de-

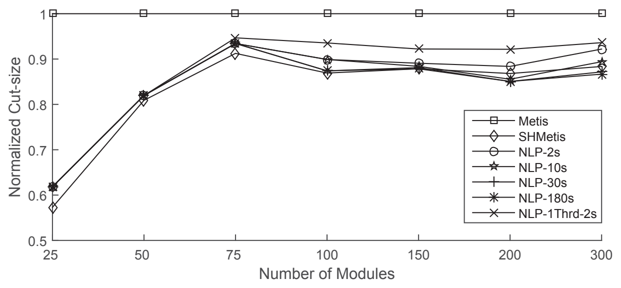

Figure 8: Comparison between the proposed NLP bipartitioner, Metis [11] and SHMetis [10]. The Gurobi solver [9] used for the NLP bipartitioner is configured to stop after 2, 10, 30 and 180 seconds to compare how good the result is for a certain time constraint. The line NLP-1Thrd-2s represents the results obtained from Gurobi solver when it is executed with one thread. All the results are normalized to the ones provided by Metis.

creasing order of the resources requirement and the number of wires in its input/output interface. The recursive trial-and-error algorithm is executed to obtain the final floorplan. The possible combinations of the placement candidates (with separate constraints for PRRs) are considered. Nevertheless, the algorithm is optimized such that it does not back-trace all the possible combinations of small modules. These modules usually have large number of possible placements to choose from. Therefore it is easier to find suitable placements for them. The algorithm stops immediately when the first feasible floorplan is found. If the algorithm cannot find any, it shifts the first vertical cut gradually closer to the right edge to enlarge one of two partitions to facilitate the placement of big modules (lines $12-14$ in Algorithm 1). The heuristic is restarted with the new cut line.

At first sight, our algorithm (generating all placements then finding the feasible combination of them) may look similar to [6,24]. However, the PRFloor is better than those because it has the sophisticated bipartitioning process to estimate the possible locations of modules and the selection of the placement candidates to filter out bad placements. These processes play an important role in making the recursive algorithm finish very fast as reported in Section 4.3.

4. EXPERIMENTS

4.1 The NLP-based Bipartitioner

Fig. 8 shows the quality of the results obtained from the proposed NLP bipartitioner in comparison with Metis [11] and SHMetis [10]. SHMetis is the single-resource hypergraph partitioning tool created by the same group which develops Metis. In the experiments, the resources requirement for each module in two partitions are set to be the same; the available resources in two partitions are also equal. We only perform *one cut* to separate the modules into two halves. Since SHMetis does not support multi-resource bipartitioning, the multiple resources are converted into one weighted sum function. We use SHMetis in the experiments for reference purposes only because it cannot be directly compared with the multi-resource bipartitioners. The NLP program is solved by the Gurobi Optimization tool [9] with default configurations and various run time constraints.

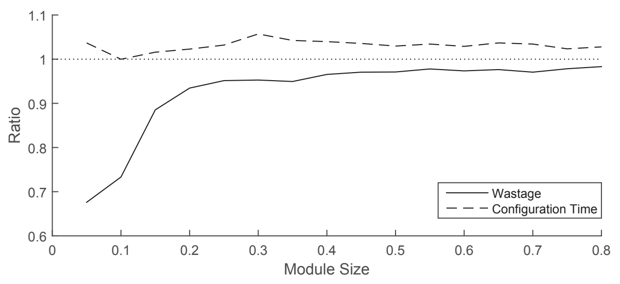

Figure 9: The ratio of the wastage and configuration time between the FPGA modeled in half-column and full-column granularity. The x-axis represents the maximum sizes of the randomly generated modules compared to the size of Virtex-6.

As can be seen in Fig. 8, the NLP bipartitioner provides better cut-size than Metis in all configurations of number of modules. The cut-size given by our method is up to 38% smaller than Metis [11]. The average cut-size is *15%* smaller. The results are becoming better when the run time constraint for the solver is increased. This is the advantage of using NLP bipartitioner compared to the heuristic one. We can increase the run time constraint to cope with the situations in which it is difficult to solve the NLP program. These situations happen when the number of modules is high or the sizes of the modules are too large.

Given that the number of modules is not expected to be very large because we only floorplan the top-level modules in the system, the features supported by the NLP bipartitioner and the quality of the results, our method is the viable solution for the problem mentioned in Section 1.

4.2 The costs saved by using the half-column model compared to the full-column model

In this section, the resource cost saved by using the half-column granularity model compared to the full-column model is examined. The half-column model is used in our proposed approach, while the full-column model is used by [20, 24].

In the experiments, the resource requirements of the modules are randomly generated. They do not exceed a predefined maximum proportion of the total resources of the Virtex-6. The modules can have any of the CLB, BRAM and DSP resources. For each range up to the maximum proportion, 500 modules are uniformly randomly generated. The smallest possible placements for each module in half-column and full-column model are recorded. The final results are the average of those placements of 500 modules. The average configuration times of the modules are also reported.

The experiments are carried out with various maximum size proportions. The resource wastage and configuration time ratios between the half-column and full-column models are reported in Fig. 9. As shown, the resource wastages of the placements of modules found by using half-column model are always smaller than the ones found in full-column model. The saving ranges from 2% to 32%.

The average configuration time of the modules in half-column model is 6% higher than the full-column model. In half-column model, the placements for modules tend to be wider to cover the required resources. They require more configuration frames because the minimum addressable Xilinx configuration frame spans the entire column [25]. Nevertheless, the controller for the Internal Configuration Access

Table 1: The set of experimental systems with the actual utilization values. For each resource type, the first number is the utilization of PRRs, the second one is of the whole system.

No. PRRs	No. Mod.	%CLB *PRR*	%CLB *All*	%BRAM *PRR*	%BRAM *All*	%DSP *PRR*	%DSP *All*
3 (65%)	99	40.8	66.2	8.9	42.3	3.8	6.4
3 (70%)	99	45.8	71.2	27.2	60.6	13.5	16.1
3 (75%)	99	50.1	75.5	21.9	55.3	10.0	12.6
3 (80%)	99	55.5	80.9	24.5	57.9	10.7	13.3
3 (85%)	99	60.1	85.5	26.2	59.6	10.7	13.3
8 (65%)	116	35.6	65.3	16.1	27.9	11.1	14.5
8 (70%)	116	40.7	70.4	17.3	29.1	11.2	14.6
8 (75%)	116	45.8	75.6	18.3	30.0	11.7	15.1
8 (80%)	116	50.7	80.5	15.9	27.6	11.2	14.5
8 (85%)	116	56.3	86.1	19.2	31.0	11.7	15.1
15 (65%)	130	34.4	65.1	28.1	46.6	22.1	25.5
15 (70%)	130	39.8	70.5	29.6	48.1	23.2	26.6
15 (75%)	130	44.9	75.6	34.6	53.1	23.4	26.8
15 (80%)	130	49.8	80.5	31.3	49.8	24.7	28.1
15 (85%)	130	57.1	87.8	27.4	45.9	25.1	28.5
24 (65%)	126	33.0	66.0	31.7	55.5	31.3	31.9
24 (70%)	126	37.5	70.5	33.2	57.0	26.6	27.2
24 (75%)	126	42.3	75.3	21.6	45.4	23.4	24.1
24 (80%)	126	47.1	80.1	31.7	55.5	28.1	28.8
24 (85%)	126	52.3	85.3	36.1	59.9	22.7	23.3

Port proposed in [23] can reach the maximum theoretical speed of *3.2 Gbps* which is fast enough to handle the increased configuration time. Besides, our approach can easily discard the placements for modules with high reconfiguration time overhead if the designer wants to do so with the expense of possible higher wastage.

4.3 The PRFloor

4.3.1 Experiment with synthetic systems

To evaluate the PRFloor, the PR heterogeneous multiprocessor system-on-chip (PR-HMPSoC) [19] is used with different number of PRRs (or Tiles as described in that work). In the original systems, there is no direct connection between PRRs because they are connected indirectly by the Network-on-Chip (NoC). In this experiment, the systems are modified such that the PRRs are also connected with each other via PLB buses to observe the wirelength between them in the final floorplan. For each system configuration, the different sets of PRMs mapped to PRRs are semi-randomly chosen such that the overall CLB utilization ranges from 65% to 85%. Additionally, each PRRs are required to have at least 1 BRAM block and 1 DSP block to make it harder for the floorplanner to find a feasible floorplan. The details of the experimental systems are shown in Table 1. The data in the rows is used to show how big the PRRs are compared to the whole system, the average size of each PRR and the heterogeneity of the PRRs (CLB, BRAM, DSP). For example, the first row indicates that there are 99 modules in the system including 3 PRRs. The whole system utilizes 66.2% CLB of the FPGA while three PRRs utilize 40.8% CLB of the FPGA. The similar interpretation can be drawn for the BRAM and DSP columns.

The average execution times of the PRFloor for each system configuration in Table 1 are presented in Fig. 10. As seen, the largest runtime of PRFloor is about 530 seconds for the system with 24 PRRs and the CLB utilization is 70%. The execution time is increasing almost linearly with the number of modules in the system and the number of PRRs. It can also be noticed that for the same system architecture,

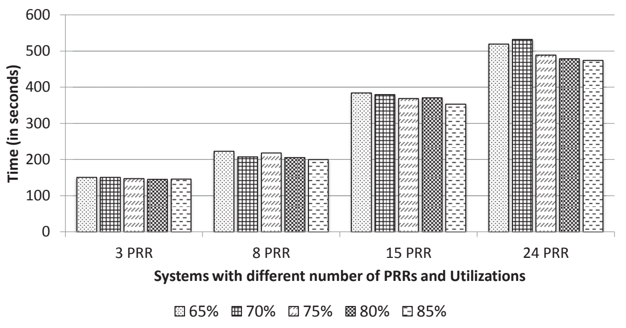

Figure 10: The execution time of PRFloor in different system configurations presented in Table 1.

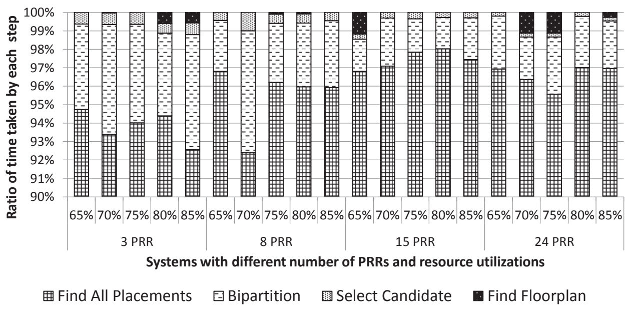

Figure 11: The ratios of time taken by the major steps in PRFloor. They are: finding all placements for modules, recursive bipartitioning, selecting placement candidates and finding final floorplan.

the execution time slightly decreases as the utilization increases. The reason is that when the resources requirement of the PRRs are larger, the number of placement candidates becomes smaller because there is lesser freedom to move the PRRs around. Thus, the trial-and-error process described in Section 3.2.7 takes lesser time to complete.

Fig. 11 provides the ratio of time taken by the major steps in PRFloor which are: finding all placements for modules, recursive bipartitioning, selecting placement candidates and finding final floorplan. As stated earlier, the recursive process used to find the final floorplan finishes very fast. It takes only at most 1.2% of the total runtime and in most cases, the percentage is almost 0. The most time-consuming process is finding all placements for modules. It constitutes more than 92% of the total runtime. This process can be further optimized in future work by having a smarter way to restrict the search space for each module instead of the whole FPGA as implemented in this work.

The experiment on the effect of varying the 2-tuple weight (α, β) in Equation 10 is also carried out. Fig. 12 plots the wirelength (the Manhattan distance) between the centroids of the PRRs and the total wastage of the PRRs. The α runs from 0 to 1 and $\alpha + \beta = 1$. The data is normalized to the results when $\alpha = \beta = 0.5$. As shown, the resource wastage decreases gradually with the increment of α. However, the wirelength does not react the same to the changes of β. In most cases, the differences are very small. It may be the cause of the Manhattan distance metric used in our current measurement. The larger the α, the smaller the placements, therefore the shorter the distance between the centroids of the placements. Another possible explanation

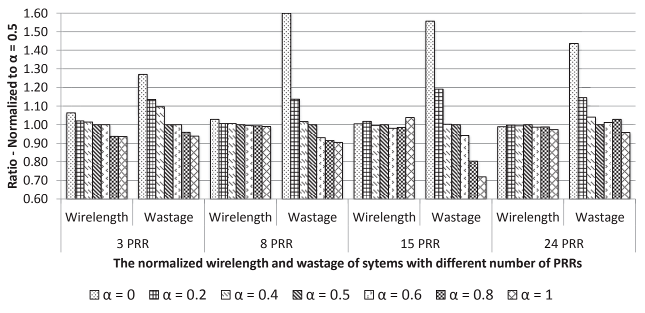

Figure 12: The effect of varying weight of the wastage in different system configurations.

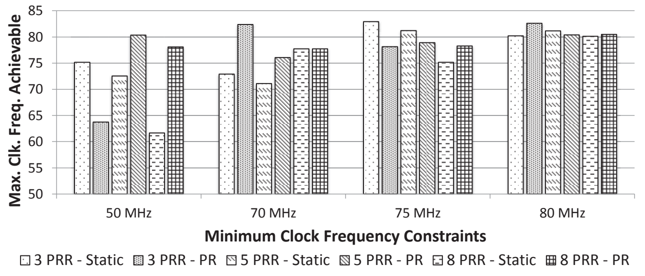

Figure 13: The maximum clock frequency achievable with different minimum clock constraints.

is the computation of the anchor points. They are simply the center points of the partitions in which they reside. In future works, we would use another metric such as the half perimeter wirelength to have a more accurate observation on the distances between PRRs. The calculation of the anchor points can also be improved by shifting the anchor point of a module toward the other modules connected to it.

4.3.2 *Experiment with actual systems*

The quality of the floorplans produced by PRFloor is further verified with the actual synthesizable systems. The maximum clock frequency achievable for PR systems is compared against the ones without PR capability. In this experiment, the original PR-HMPSoCs described in [19] are used in which there is no direct connection between Tiles. All Tiles are Xilinx Microblaze processor configured with default settings. In the static systems, these Tiles are static. In PR systems, they are partially reconfigurable. We use the Xilinx XPS 14.4 with default settings to implement the static systems. The PR systems are implemented by the Xilinx PlanAhead 14.4 with similar settings. The placement constrains for PRRs are generated by the PRFloor. The minimum clock constraint is varied from 50 MHz up to 80 MHz. The achievable clock frequencies for these systems are shown in Fig. 13. As can be seen from the chart, in all cases, the minimum clock frequency constraints are satisfied. Moreover, in most cases, the PR systems even have higher maximum clock frequency than the static systems. On average, it is 3% higher.

4.3.3 *Comparison with the previous work*

For comparison with the related work, it is not possible to compare our work directly with them. The problem that our method is dealing with is different from all the existing works discussed in Section 2. Moreover, there is no standard

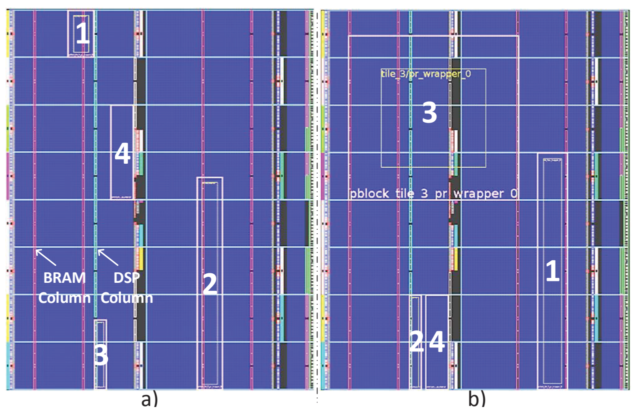

Figure 14: The floorplans provided by PRFloor for the same systems used in the examples in Fig. 2. The floorplans in *a)* and *b)* correspond to Fig. 2a and Fig. 2b respectively.

Table 2: The resource requirements of the PRRs used in the experiments in Section 4.3.3.

PR	Fig. 14a			Fig. 14b		
Regions	*CLB*	*BRAM*	*DSP*	*CLB*	*BRAM*	*DSP*
PRR 1	100	4	0	100	20	0
PRR 2	50	18	0	50	0	14
PRR 3	50	0	10	50	40	20
PRR 4	200	0	0	200	0	0

benchmark for FPGA floorplanning. Intuitively, in terms of the complexity of the experimental systems, we have much larger number of modules (including PRRs). The communication architecture between static modules and PRRs is also more complicated. However, we do have two use-cases to compare the PRFloor with [20]. The same systems given in Fig. 2 are fed into PRFloor with the identical FPGA, Xilinx Virtex-5 XC5VLX110T. The final floorplans are illustrated in Fig. 14. In these systems, the PRRs are connected indirectly by NoC and the PRFloor is run with $\alpha = \beta = 0.5$. The weight costs for CLB, BRAM and DSP are 1, 12 and 60 respectively. The resource requirements of the PRRs used in the experiments are provided in detail in Table 2.

Regarding the system in Fig. 14a, the total wastage cost given by [20] is 652 while ours is 19% lower, 530, thanks to the half-column granularity described in Section 3.2.3. The PRRs in our floorplan are also placed closer to each other. The total Manhattan distances (with weight 1) between four PRRs in our floorplan is 35% lower than [20].

For the system in Fig. 14b, one of the static modules requires 10 DSP blocks; therefore the PRFloor reserves that resource for it between PRRs 2 and 3. On the other hand, in Fig. 2b, the floorplanner [20] does not take into account the static modules. Hence, the remaining number of DSP blocks after floorplanning is just 8 which is not sufficient to implement the static module.

5. CONCLUSIONS AND FUTURE WORKS

This paper presents an automatic floorplanner for PR systems using the proposed NLP-bipartitioner to address the differences between the FPGA and VLSI floorplanning problem. Unlike other previous works, PRFloor takes into account the complex connections between both static modules and PRRs rather than just between PRRs. The experiments with various system setups show that PRFloor can provide results in a couple of minutes. The quality of the floorplan is demonstrated via the higher maximum clock frequency achievable than the comparable static systems.

The forthcoming works are to improve the performance and the quality of the algorithm. It would be extended to support bitstream relocation at runtime with tighter constraints for the placements of the PRRs.

6. 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.

7. REFERENCES

[1] P. Banerjee, M. Sangtani, and S. Sur-Kolay. Floorplanning for partially reconfigurable FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 8–17, 2011.

[2] C. Beckhoff, D. Koch, and J. Torreson. Automatic floorplanning and interface synthesis of island style reconfigurable systems with GOAHEAD. In *Architecture of Computing Systems–ARCS 2013*, pages 303–316. Springer, 2013.

[3] C. Bolchini, A. Miele, and C. Sandionigi. Automated resource-aware floorplanning of reconfigurable areas in partially-reconfigurable FPGA systems. In *Field Programmable Logic and Applications International Conference on*, pages 532–538. IEEE, 2011.

[4] L. Cheng and M. D. Wong. Floorplan design for multimillion gate FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(12):2795–2805, 2006.

[5] J. Cong, M. Romesis, and J. R. Shinnerl. Fast floorplanning by look-ahead enabled recursive bipartitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(9):1719–1732, 2006.

[6] F. Duhem, F. Muller, W. Aubry, B. Le Gal, D. Négru, and P. Lorenzini. Design space exploration for partially reconfigurable architectures in real-time systems. *Journal of Systems Architecture*, 59(8):571–581, 2013.

[7] Y. Feng and D. P. Mehta. Heterogeneous floorplanning for FPGAs. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pages 6–pp. IEEE, 2006.

[8] D. Göhringer, M. Hübner, E. N. Zeutebouo, and J. Becker. Operating system for runtime reconfigurable multiprocessor systems. *International Journal of Reconfigurable Computing*, 2011.

[9] Gurobi. Gurobi Optimization version 6.0.2. http://www.gurobi.com, April, 2015.

[10] G. Karypis and V. Kumar. hMETIS 1.5: A hypergraph partitioning package, 1998.

[11] G. Karypis and V. Kumar. Metis - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 5.1.0. *Technical report*, 2013.

[12] D. Li and X. Sun. *Nonlinear integer programming*, volume 84. Springer Science & Business Media, 2006.

[13] S. K. Lim. *Practical problems in VLSI physical design automation*. Springer, 2008.

[14] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik. A reconfiguration-aware floorplacer for FPGAs. In *Reconfigurable Computing and FPGAs, 2008. ReConFig'08. International Conference on*, pages 109–114. IEEE, 2008.

[15] A. Montone, M. D. Santambrogio, F. Redaelli, and D. Sciuto. Floorplacement for partial reconfigurable FPGA-based systems. *International Journal of Reconfigurable Computing*, 2011.

[16] S. Mukhopadhyay, P. Banerjee, and S. Sur-Kolay. Balanced bipartitioning of a multi-weighted hypergraph for heterogeneous FPGAS. In *Programmable Logic (SPL), 2011 VII Southern Conference on*, pages 91–96. IEEE, 2011.

[17] C. E. Neely, G. Brebner, and W. Shang. ReShape: Towards a High-Level Approach to Design and Operation of Modular Reconfigurable Systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 6(1):5, 2013.

[18] P. Ngatchou, A. Zarei, and M. El-Sharkawi. Pareto multi objective optimization. In *Intelligent Systems Application to Power Systems. Proceedings of the International Conference on*, pages 84–91. IEEE, 2005.

[19] T. D. Nguyen and A. Kumar. PR-HMPSoC: A versatile partially reconfigurable heterogeneous Multiprocessor System-on-Chip for dynamic FPGA-based embedded systems. In *Field Programmable Logic and Applications (FPL), 24th International Conference on*, pages 1–6. IEEE, 2014.

[20] M. Rabozzi, J. Lillis, and M. D. Santambrogio. Floorplanning for Partially-Reconfigurable FPGA Systems via Mixed-Integer Linear Programming. In *Field-Programmable Custom Computing Machines (FCCM), Annual International Symposium on*, pages 186–193. IEEE, 2014.

[21] N. Selvakkumaran, A. Ranjan, S. Raje, and G. Karypis. Multi-resource aware partitioning algorithms for FPGAs with heterogeneous resources. In *Proceedings of the 41st annual Design Automation Conference (DAC)*, pages 741–746. ACM, 2004.

[22] L. Singhal and E. Bozorgzadeh. Multi-layer floorplanning on a sequence of reconfigurable designs. In *Field Programmable Logic and Applications (FPL). International Conference on*, pages 1–8. IEEE, 2006.

[23] K. Vipin and S. Fahmy. A high speed open source controller for fpga partial reconfiguration. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 61–66, Dec 2012.

[24] K. Vipin and S. A. Fahmy. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 13–25. Springer, 2012.

[25] Xilinx. *Xilinx Partial Reconfiguration User Guide UG702 (v14.5)*, 2013.

[26] J. Z. Yan and C. Chu. DeFer: deferred decision making enabled fixed-outline floorplanning algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):367–381, 2010.