Library-Based Placement and Routing in FPGAs with Support of Partial Reconfiguration

Mao, Fubing; Chen, Yi-Chung; Zhang, Wei; Li, Hai (Helen); He, Bingsheng

2015

Mao, F., Chen, Y.-C., Zhang, W., Li, H., & He, B. (2016). Library-Based Placement and Routing in FPGAs with Support of Partial Reconfiguration. ACM Transactions on Design Automation of Electronic Systems, 21(4), 1-26.

https://hdl.handle.net/10356/82303

https://doi.org/10.1145/2901295

Library-based Placement and Routing in FPGAs with Support of Partial Reconfiguration

FUBING MAO, Nanyang Technological University
YI-CHUNG CHEN, University of Manchester
WEI ZHANG, Hong Kong University of Science and Technology
HAI (HELEN) LI, University of Pittsburgh
BINGSHENG HE, Nanyang Technological University

While traditional FPGA design flow usually employs fine-grained tile-based placement, modular placement is increasingly required to speed up the large-scale placement and save the synthesis time. Moreover, the commonly used modules can be pre-synthesized and stored in the library for design reuse to significantly save the design, verification time and development cost. Previous work mainly focuses on modular floorplanning without module placement information. In this paper, we propose a library-based placement and routing flow, which best utilizes the pre-placed and routed modules from the library to significantly save the execution time while achieving the minimal area-delay product. The flow supports the static and reconfigurable modules at the same time. The modular information is represented in B*-Tree structure, and the B*-Tree operations are amended together with Simulated Annealing to enable a fast search of the placement space. Different width-height ratios of the modules are exploited to achieve area-delay product optimization. Partial reconfiguration-aware routing using pin-to-wire abutment is proposed to connect the modules after placement. Our placer can reduce the compilation time by 65% on average with 17% area and 8.2% delay overhead compared with fine-grained results of VPR through the reuse of module information in the library for the base architecture. For other architectures, the area increase ranges from 8.32% to 25.79%, the delay varies from -13.66% to 19.79%, and the running time improves by 43.31% to 77.2%.

CCS Concepts:•**Hardware** → **Placement;** *Wire routing; Software tools for EDA;*

Additional Key Words and Phrases: FPGA, Partial Reconfiguration, B*-tree, Placement, Routing

ACM Reference Format:
Fubing Mao, Yi-chung Chen, Wei Zhang, Hai (Helen) Li and Bingsheng He, 2015. Library-based Placement and Routing in FPGAs with Support of Partial Reconfiguration. *ACM Trans. Des. Autom. Electron. Syst.* xxx, xxx, Article XXXX (June 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

With the technology advancement, both semiconductor industry and design community face the great challenge that the design effort is continuously increasing [Castro-lpez et al. 2006]. The rapidly growing design complexity and short time-to-market pressure make the problem even worse [Castro-lpez et al. 2006]. Library-based design is a promising solution to address this challenge as it facilitates the design reuse. Design reuse is an approach that utilizes a previously successful design in a new design

Author's addresses: F. Mao and B. He, School of Computer Engineering, Nanyang Technological University, Singapore; email: fmao001@e.ntu.edu.sg, bshe@ntu.edu.sg; W. Zhang, Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong; email: wei.zhang@ust.hk; Y.-C. Chen, School of Computer Science, University of Manchester, UK; email: yichung.chen@manchester.ac.uk; H. Li, Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania, USA; email: hal66@pitt.edu.

DOI: http://dx.doi.org/10.1145/0000000.0000000

project to reduce the design, verification cycle and risk [Castro-lpez et al. 2006; Xiu 2007]. It also reduces design cost since the reused components have been prepared and verified [Castro-lpez et al. 2006; Xiu 2007]. Library based design has been studied in previous work. The work in [Wang and Leeser 2010] presents a floating-point library to support floating-point computation in general formats and enable a higher level of parallelism. In [Sklyarov et al. 2003], they reuse hardware components of the library in *Field-Programmable Gate Array* (FPGA) design for improving the verification of the digital circuits implemented on FPGA. In [Lavin et al. 2011], hard macros (already placed and routed) are reused to reduce the compilation time. IP reuse is one type of design reuse and it helps to address the gap between the capacity for design complex system and productivity [Gajski 1999]. IP modules are used for improving productivity in [Zergainoh et al. 2005]. In [Hekmatpour et al. 2005], a methodology is described for IP design and integration. The above mentioned studies mainly consider the library based design for design of application-specific circuits or reduce compilation time for the FPGA design flow based on hard macros consisting of synthesized, placed and routed circuits.

In recent years, FPGA has attracted attention as accelerators in the computing systems due to its high performance, low power, high design flexibility and low cost [Telle et al. 2004; Stitt et al. 2004; Vereen, L. 2004]. FPGAs are widely used in various fields to accelerate the intensively critical computations [Cong et al. 2011]. To implement a design in FPGA, placement and routing are important stages which significantly affect the performance of the design. In traditional design flow, fine-grained tile-based placement is performed to achieve the optimal solution. However, it usually requires a long searching time. Especially for a large-scale design, the running time of tile-based placement can be hours or days [Sankar and Rose 1999; Gort and Anderson 2014]. Hierarchical or modular floorplanning is hence proposed to speed up the process [Areibi et al. 2007; Samaranayake et al. 2009]. The requirement for modular floorplanning and placement also arises from the support for dynamic *partial reconfiguration* (PR) in FPGA. This operation requires the separation of PR logic, *i.e.,* the logic reconfigured during operation, and static logic, *i.e.,* the logic kept unchanged during operation. Thus, decomposing the circuit into logic modules is a well adopted method to differentiate the reconfigurable logic and static logic [Xilinx 2012; Altera 2010]. The final and most important advantage of the module-based placement is that it can enable module reuse and the commonly used modules with pre-placement and routing can be stored in a library for later reuse to save significant development efforts, like IP cores.

In this paper, in order to take the advantage of design reuse in FPGA, we propose a library-based flow to improve the efficiency of placement and routing. The pre-placed modules in the library are used to significantly save the placement time. Moreover, with the accurate pre-placement module information, it can better direct the global placement of the modules in the whole design. A *B*tree-based placer* (BMP) is introduced to perform a fast modular placement on both fine-grained and coarse-grained FPGA resources considering both module sizes and aspect ratios. Reconfigurable modules with multiple contexts are well supported during the placement. Finally, we propose a pin-to-wire abutment routing interface to connect the modules in the reconfigurable and static regions which does not require the macro or proxy logic which are usually needed in current tools [Xilinx. 2011b; Carver et al. 2009; Athanas and et al. 2007].

The proposed flow has been integrated into the *Versatile Place and Route* (VPR) [Rose and et al. 2012] to substitute the original placer and router. Experiments demonstrate improvement on execution time with acceptable area and delay overhead compared with the results of tile-based VPR. The results show that BMP has around 17% overhead in area and 8.2% overhead in delay. The execution time is significantly

improved by 65% for the basic architecture. For other architectures, the area increase ranges from 8.32% to 25.79%, the delay varies from -13.66% to 19.79%, and the running time improves by 43.31% to 77.2%. We also perform thorough design space explorations to analyze and optimize the parameters of BMP. Our main contributions can be summarized as follows:

— We propose a library-based placement and routing flow to facilitate design reuse and improve the placement quality. The multi-context reconfigurable module is well supported during the placement and routing.
— We utilize B*-tree representation to enable a fast modular placement on both fine-grained and coarse-grained fabric considering different module ratios.
— We introduce the detailed pin-to-wire routing interface to support the PR-aware routing.

The rest of the paper is organized as follows. Section 2 reviews the previous work related to module-based design flow. Section 3 introduces the overview of our proposed placement and routing flow. Section 4 discusses the methodology for module library construction prepared for the later placement and routing. The detailed placement steps are introduced in Section 5, and Section 6 describes the PR-aware routing interface design. Section 7 describes the local placement and local routing for multiple contexts in the PR region. Section 8 discusses the experimental results and parameter optimizations through design space exploration. Finally, we conclude with our remarks in Section 9.

2. RELATED WORK

2.1. Tile-based placement

Conventional place and route tools for FPGA are based on fine-grained homogeneous units to map the logic functions. *Verilog-to-Routing* (VTR) [Rose and et al. 2012] gave a complete flow from *hardware description language* (HDL) to physical mapping on FPGAs of various hardware architectures, where VPR was used inside to perform the tile-based placement and routing. Later work extended the fine-grained placement to heterogeneous FPGAs. For example, [Jamieson et al. 2013] introduced the genetic algorithm for solving the heterogeneous FPGA placement. [Selvakkumaran and et al. 2004] proposed a multilevel multi-resource partitioning algorithm for heterogeneous FPGA placement. [Hu 2006] employed a multi-layer density system for the heterogeneous FPGA placement. [Gort and Anderson 2012] proposed an analytical placer for the heterogeneous FPGAs. However, as we mentioned before that the tile-based placement faces the challenges of a long running time for large-scale designs.

2.2. Module-based floorplanning and placement

Module-based floorplanning has been proposed in recent years to ease the mapping of large-scale designs in modern FPGAs. In [Cheng and Wong 2006], they proposed a floorplanning algorithm for heterogeneous resource. They firstly found a position for the specific slicing structure and then used simulated annealing to tune towards a better solution. [Yuan et al. 2005] proposed an approach named “less flexibility first” algorithm to find locations of different modules and used a metric to estimate the solution priority. [Banerjee et al. 2009] used a three-phase deterministic approach to get a unified floorplan topology and then used the bipartitioning method to find the final positions for the heterogeneous blocks. [Liu et al. 2011] proposed a high utilization method for heterogeneous FPGAs. They used the non-slicing structure to optimize the wirelength first and then used min-cost-max-flow algorithm to assign the positions to *Configurable Logic Blocks* (CLBs). Finally they assigned positions to the RAMs and

DSPs. [Chen et al. 2014] proposed a packing and analytical placement flow for heterogeneous FPGAs from floorplanning to detailed placement. They used look-ahead legalization to allocate positions to different resources.

In order to support partial reconfiguration, commercial FPGA mapping flow has included PR-aware placement. However, it needed manual specification which was an error-prone and tedious process. Xilinx Early-Access (EA) PR design flow [Xilinx 2012; He and et al. 2012; Xilinx. 2011a] was commonly used in the PR designs, which required that PR regions were manually defined in terms of shape, size, and physical location. In order to reduce the manual efforts for searching in a large design space, various work proposed automatic floorplanning for PR modules. In earlier work [Bazargan et al. 2000], each PR module was modeled as a fixed-size block and the PR floorplanning was formulated as a three-dimensional template placement problem. However, this assumption was difficult to apply in practical applications. Later studies like [Yousuf and Gordon-Ross 2010; Beckhoff and et al. 2013] developed automatic flow for PR floorplanning based on Xilinx process and special bus was needed to connect modules and support run-time reconfiguration. [Carver et al. 2009] developed an automated simulated annealing-based bus macro placement tool and evaluated the tool using timing results generated by Xilinx PAR (place and route) utility. The dimensions of partial reconfigurable region were fixed and the PR region was manually placed, then the bus macro was automatically placed around the reconfigurable region. [Banerjee and et al. 2011] extended the floorplanning algorithm in [Cheng and Wong 2006] to consider the PR floorplanning in heterogeneous resources. In that work, a global floorplan generation approach was introduced to obtain shared positions for common modules across sub-task instances. [Singhal and Bozorgzadeh 2006] proposed a multi-layer floorplanner which combined the multiple reconfigurable design's floorplanning and maximized the reuse of common components to reduce the reconfiguration overhead. These two work focused more on the maximization of resource reuse among multiple designs. [Vipin and Fahmy 2011] proposed an efficient mapping method which performed the floorplanning optimization of reconfigurable modules from the high-level estimation.

There was also work proposed to perform simultaneously floorplanning and placement or a direct global placement. [Montone et al. 2010] introduced a mapping flow, which partitioned the scheduled task graph into reconfigurable regions and then performed floorplanning and placement of reconfigurable regions in heterogeneous reconfigurable FPGAs targeting the wirelength minimization. Most recently, [He and et al. 2012] proposed a fine-grained placement for PR FPGA. Compared with these studies, our work focused more on the efficient reuse of pre-placed modules in the library to significantly save the execution time. In addition, our placement introduced B*-tree representation to represent module information and speed up the searching speed while considering different ratios of modules during placement to achieve the area-delay product optimization.

2.3. PR-aware routing

The routing interface design is a major step for supporting the PR operation and used for connecting the static logic and reconfiguration logic. In Xilinx FPGAs, bus macros were predefined and used for connecting the static and reconfigurable modules [Shah and Rose 2012; Claus and et al. 2007]. The bus macros were double-lines or hex-lines in the early generation of Virtex II or Virtex II-Pro devices, while a Look-Up-Table (LUT) based bus macro was used in recent devices, such as Virtex-4, Virtex-5 and Spartan-III. [Claus and et al. 2007] used a bus macro generated based on Xilinx Design Language (XDL) for connecting the static and reconfigurable parts. [Koch et al. 2008] proposed a tool called 'ReCoBus-Builder' to enable the communication between

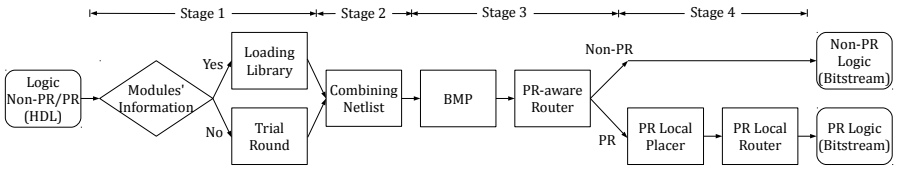


Fig. 1. Mapping flow for library based design.

the static and reconfigurable modules through a fixed bus infrastructure or dedicated point-to-point links with other parts of the system. [Athanas and et al. 2007] used a wrapper structure to connect different regions and each reconfigurable module was encased in a wrapper structure before placement and routing. The wrapper structure had anchor points which existed at pre-defined locations for the module's ports. Xilinx PR design flow [Xilinx. 2011b] used 'proxy' logic to solve the boundary crossing connection problem. The 'proxy' logic was a one-input LUT which had a fixed placement in a reconfigurable region. It must be the same for every reconfigurable module. It involved logic overhead for implementing the 'proxy' logic and extra delay for passing the 'proxy' logic. [Koch et al. 2010] used the pre-assigned route for connecting different regions and eliminating 'proxy' logic. However, it imposed constraint on the communication between static systems and PR regions which may increase the design complexity and narrow its range of usage. [Shah and Rose 2012] explored the pin-to-wire connections and measure the difficulty to form such type connections. However, it was not used in the PR system and it did not describe the interface design when there were more than two PR regions. We adopted the pin-to-wire concept and proposed a detailed pin-to-wire interface to support PR-aware routing without requirement of the bus macro or 'proxy' logic.

3. LIBRARY-BASED MAPPING FLOW

The mapping flow we propose to solve the library-based placement is shown in Fig. 1. The flow is composed of four main stages: information collection of individual sub-function modules, netlist combination, module-based placement, PR-aware routing targeting delay and area optimization, and local placement and local routing for PR regions. First, we assume that logic modules have been decomposed from the main logic function by designers. The first stage gathers the information on area, delay, connection ports, netlist, *etc.* of the modules from the library storing the information of modules. If the module is not available in the library yet, our flow can run a trial round to generate the corresponding information from the HDL files of the modules. Since different ratios may be suitable for different designs, the trial round will generate the module placement and routing for k different ratios which result in minimal area-delay product. If the module is a reconfigurable module, the area and delay for multiple contexts are considered where each context is a configuration. Then the logic modules are combined to form the netlist for the whole function in the second stage. In the third step, a *B*tree-based placer* (BMP) is introduced to place the static and reconfigurable modules simultaneously in heterogeneous FPGA resources. BMP introduces B*-tree representation [Chen and Chang 2006] to model a floorplan, and enables corresponding operations for fast search of the optimal solution. The cost function of total delay and area is used to guide the simulated annealing based search algorithm [Chen and Chang 2006]. Note that different ratios may occupy different placement resources which affect the placement result significantly, especially considering the special resource and position constraint, *i.e.*, modules with memory blocks must be placed in special positions. During the placement, not only the module size, delay, but

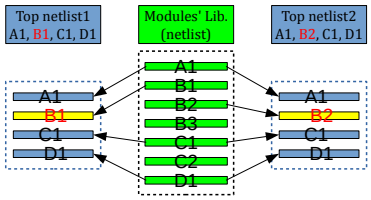


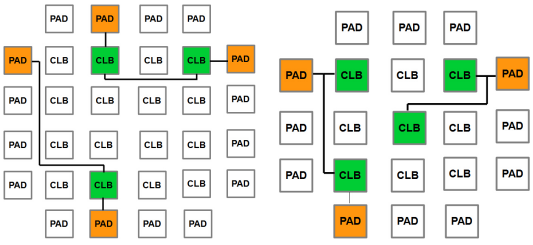
Fig. 2. Library based structure.

also the aspect ratio (width/height) is considered to achieve the area-delay product optimization, which creates a large solution space. After placement, the PR-aware routing is performed to connect the modules with restricted routing outside of PR region to prevent resource interference during PR procedure. The last stage of the flow is specifically for PR modules. It performs local placement and routing for each context inside the module.

For designing a new function, the designer only needs to list modules in use and provide connection port name and top module IO information. Then to finish the design, only module-based placement and routing are needed. Fig. 2 shows an example for the library based design. There are totally four functions (modules) A, B, C and D, where B and C are reconfigurable modules with multiple contexts. The designer can specify the module list, such as A1, B1, C1 and D1 as initial modules and connections in the top netlist. Then modules are fetched from the library and combined into a complete function netlist. After the modular placement and routing, the netlist is implemented. For reconfigurable module B and C, the implementation supports other contexts to substitute B1 and C1 during operation.

4. MODULE LIBRARY CONSTRUCTION

As shown in the overall mapping flow, we need to prepare the modules and store the information of all the modules in the module library when needed. VTR [Rose and et al. 2012] can be used to synthesize each module from verilog file to layout. The synthesized modules stored in a library may contain both fine-grained and coarse-grained resources. We need to consider the following several aspects for constructing a module library. Firstly, we need to determine the delay, area and pinlist for the reconfigurable modules with multiple contexts. Secondly, we need to choose the k ratios of modules to achieve a good area-delay trade-off. Thirdly, we need to support the delay estimation for the module with a specific ratio, *i.e.*, an user-specific ratio which is not available in the current library, since the modules are only synthesized with the proper k ratios. In the next subsections, we discuss each of the aspect in detail.



(a) 4x4 clbs FPGA. (b) 3x3 clbs FPGA.

Fig. 3. Two contexts of a reconfigurable module

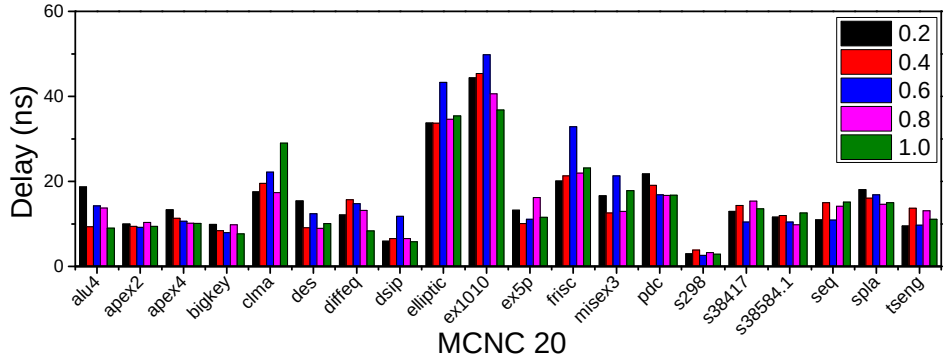


Fig. 4. Relation between shapes and delay for MCNC20 benchmark.

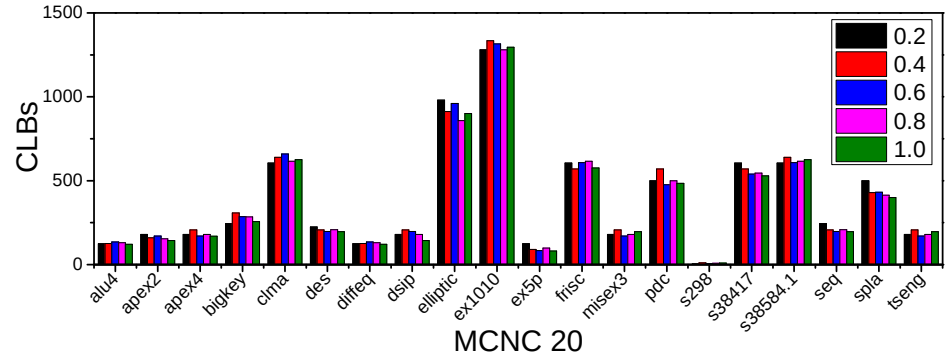


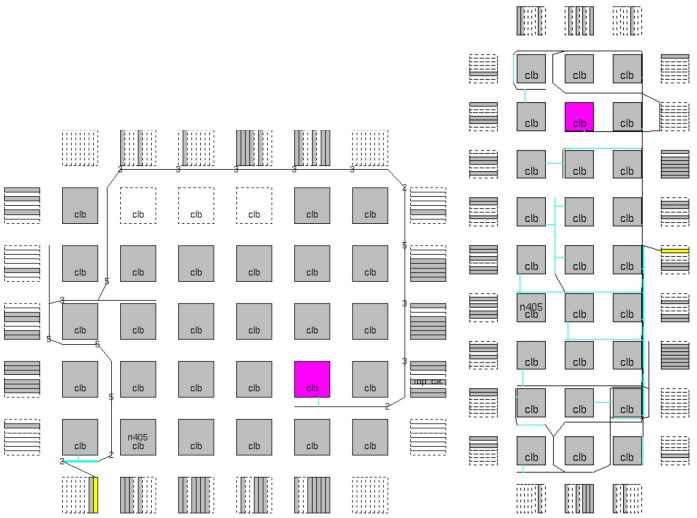
Fig. 5. Relation between shapes and area for MCNC20 benchmark.

4.1. Synthesis of Reconfigurable Modules

Since the reconfigurable modules have multiple contexts for different functions, the synthesis flow needs to run for each context of the reconfigurable function. The area and delay of the module are determined by the maximum one across all the contexts. However, since the routing between the modules may go through the static region and cannot be reconfigured, the pinlist of the module needs to satisfy the input/output requirement of all the contexts. Hence, the pinlist of the module is the combination of IOs of all the contexts. For example, Fig. 3 shows the two contexts of a reconfigurable module. Since the context in (a) has a larger area than the context in (b), the module size and delay are determined according to the context in (a). However, for the pinlist of the module, we need to find out all the unique IO nets. For example, assume that two contexts, context R1 and context R2, share the IO nets of netA and netB, but context R2 has a unique IO net of netC. In order to enable the context switch between R1 and R2, all the required IO nets netA, netB, netC need to be routed and connected. Hence, the total pinlist of the reconfigurable module is the combination of unique IOs of the two contexts, which is {PadA, PadB, PadC}. Then the local placement and routing of the contexts inside the module are based on the pin arrangement of the module.

4.2. Module Ratio Selection

We observe that the shape of a module shows impact on delay and area. Fig. 4 and Fig. 5 demonstrate simulation results of delay and area of MCNC benchmarks [Minkovich K. 2007] for various aspect ratios, respectively. Legends shown in the



(a) Square shape. (b) Rectangular shape.

Fig. 6. Module of various shapes and delay.

figures are aspect ratios, width over height, of the shapes. Note that here we assume the basic reconfigurable unit is one CLB without lost of generality. The parameter for the aspect ratio can be set to other number without affecting the placement method. It can be seen from the figure that among different ratios, the delay has at most 38% difference for the same benchmark. Fig. 6(a) and Fig. 6(b) demonstrate placement results of an example module *rng5* in the shapes of square and rectangle, respectively. The corresponding delays are $2.56ns$ and $2.05ns$. The module of the square shape has extra 25% delay compared to the rectangular one. The module *rng5* is a simple logic which only needs fewer tiles for the logic function, thus, a shape with low aspect ratio makes input signal pass through logic to output in a shorter path incurring smaller delay. Area of the benchmarks has less variation than delay for different aspect ratios because each benchmark has fixed numbers of tiles in usage. Area differences among various ratios are from the wasted area of empty tiles included in the rectangular shape.

There are several types of resources in FPGA and different resources in a module must be placed in their corresponding positions. Fig. 7 shows the impact of ratio change to a module with heterogeneous resources. For example, a logic has three memory blocks and six CLBs. When the ratio is 1/4, we can place it in the region A. When we change the ratio to 4/1, the new space with the same area is invalid, since it does not have enough resource (RAMs) for fitting the logic. Thus empty region B cannot be used for the module. Hence, for heterogeneous modules, we not only need to check the space size, but also need to check the type of resource. In other words, we must place the module in a valid position which makes use of the resource efficiently and results in a shorter delay.

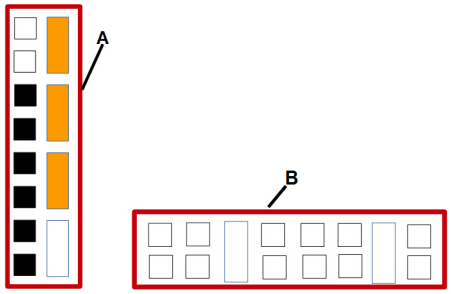
To indicate the delay and area for each module with different aspect ratios, we introduce a parameter τ which is $area \times delay$ of a shape of the module. Shapes of each module are sorted in the ascending order of τ in a list for reference in the later placement stage.

4.3. Delay Estimation Model for the Modules with Specific Ratios

Constructing an efficient module library is a challenging problem due to the reason that it is difficult and impractical to collect the implementations of a PR logic with all the possible ratios in the module library. We currently store the implementations of the modules with the best k τ in the module library. However, sometimes the user may want to try other ratios during the module placement without pre-synthesized module layout. To enable this flexibility, we propose an approach for estimating the module delay based on the available module information for these specific ratios which do not exist in the current library. We use piecewise linear interpolation to estimate module delay and give an effective and fast feedback to the user for the search of optimal placement. We first sort the module in the ascending order of its aspect ratios. Then we create a linear interpolation function between each adjacent points. Thus each ratio can use a corresponding function to get its estimated delay value. The more the aspect ratio points exist in the current library, the more accuracy the approximate approach can achieve. We adopt this estimation approach because modules for two adjacent points usually have similar structure which may reflect the delay information more accurately than the wirelength estimation due to the routing impact. Moreover, it saves the efforts for identifying the critical path. An example of module delay estimation is shown in Fig. 8. We assume that we have four ratios for a module in the library. The ratios are 0.2, 0.4, 0.6 and 0.8. We need to estimate the module delay with ratio A = 0.5. Thus we can use the yellow line to approximate its delay. The function for yellow line is $y = 5x$ and the delay is 2.5 for the point A. We also apply this approach in our flow. Assume $k = 5$ so that we have five ratios for each module stored in the library and the ratios are 0.2, 0.6, 1.0, 1.6 and 5.0. We need to estimate the module delay with ratios 0.4, 0.8, 1.2 and 2.5. During placement, we use the delay estimation approach to estimate these specific ratios. The experiments show that our estimation method can achieve 93% accuracy on average on the critical path of the final routing results. Some previous studies [Nayak et al. 2002; Mak et al. 2007; Hung et al. 2009] report that their delay estimation approaches for FPGA have approximate error rate of 13% which shows our approach is feasible.

5. LIBRARY-BASED PLACEMENT

With the combined netlist, the modules picked up from the library are placed into the underlying reconfigurable fabric in the placement stage. In this section, we first



(a) Coarse-grained module with ratio 1/4. (b) Coarse-grained region with ratio 4/1.

Fig. 7. Coarse-grained module with different ratios

introduce the problem formulation for the library-based placement and then focus on the discussion of the detailed placement steps.

5.1. Problem Formulation

The problem of library-based placement can be formed as the follows. Given a set of n rectangular modules $B = \{b_1, b_2, ..., b_n\}$ stored in the library, where each module has a width and height denoted by w_i and h_i, $1 \leq i \leq n$ respectively. The aspect ratio of module b_i is defined by w_i/h_i. A placement $p = \{(x_i, y_i)\}(1 \leq i \leq n)$ with modules is an assignment of the rectangular modules $b_i's$ such that no two rectangular modules overlap and the bottom-left corner coordinate of module b_i is assigned to (x_i, y_i). The objective is to optimize the area utilization, wirelength and delay.

5.2. B*-tree Representation

In order to enable a fast search of optimal placement in the solution space, we use B*-Tree to represent the module placement. Here we introduce the background for the B*-Tree representation. B*-tree is an ordered binary tree structure proposed in modern floorplanning of ASIC designs [Chen and Chang 2006]. Compared to other data structures, B*-Tree provides faster searching and area estimation, convenient handling of constraints, linear time transformation between the tree and placement. The solution space of B*-tree algorithm is $O(n!2^{2n}/n^{1.5})$ [Chen and Chang 2006], where n is the number of modules.

To build a B*-Tree, the bottom-left corner of the placement is taken as the root. B*-tree representation of the placement is built in a recursive fashion from the root. The subtree is first constructed at the left-hand side and is then built in the same manner at the right-hand side. Each node n_i in a B*-tree denotes a module and the root of a B*-tree corresponds to the module on the bottom-left corner. The left child n_e of the node n_i denotes the module b_e which is the lowest adjacent unvisited module on the right-hand side of b_i, *i.e.*, $x_e = x_i + w_i$. The right child of n_r of the node n_i denotes module b_r which is the lowest unvisited module above and adjacent to module b_i and its x-coordinate equal to that of b_i, *i.e.*, $x_r = x_i$. And also $b_r's$ y-coordinate is smaller than that of the top boundary of the module on the left-hand side and adjacent to b_i, if any. An example of the mapping graph for B*-tree and placement is shown in the Fig. 9.

5.3. B*-tree based Module Placement

As discussed in the last section, given a non-overlapping placement, modules are represented as the nodes of a B*-Tree, which provides a fast searching and area estimation with linear time transformation between the tree and placement. Simulated Annealing (SA) is performed to search for the efficient placement based on the B*-Tree. Without loss of generality, we assume that SA algorithm begins with a randomly

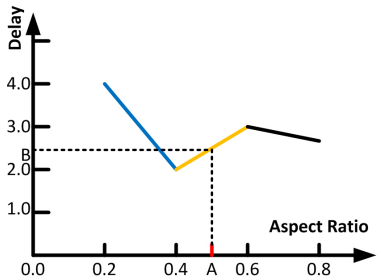


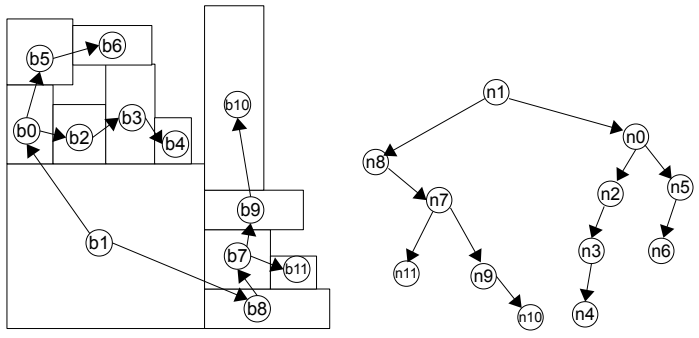
Fig. 8. Module delay estimation for a specific ratio.

generated placement of modules. Then in each iteration, the B*-tree based operation is performed to explore the solution space as follows:

— OP1 loads module with shape of 1/(aspect ratio).
  It is similar to rotate node/block by 90 degrees in the B*-Tree as shown in Fig. 10(a). Theoretically, the delay and the area of individual module with aspect ratio and (aspect ratio)$^{-1}$ are the same. However, it affects the total placement results.
— OP2 moves module to an empty place.
  It is to delete a node and move it to another empty place.
— OP3 swaps two modules.
  It swaps two nodes in the B*-Tree.
— OP4 loads a different aspect ratio of the module.
  It is modified from the original B*-Tree operation of resizing a soft block. For placement on FPGA, resizing of modules cannot be arbitrary size, but is equivalent to apply different aspect ratios to the module.

When placing the modules with heterogeneous resources, position constraints need to be imposed on the operations. For example, the module with memory blocks must be placed in a valid position. Moreover, for the module with heterogeneous resources, the hardware fabric can have different positions inside the module for the same module ratio. For example, the column of memory blocks can be on the left or right of the CLB column. To simplify the design, currently we assume that the shift of the resource position inside the module will not change the module delay. The assumption is similar to the previous studies [Xilinx. 2011a; Beckhoff and et al. 2013; Xu et al. 2014]. Fig. 11 illustrates an example for the placement of heterogeneous modules. Here each module represents a static region or reconfigurable region. Fig. 11 (a) is the coarse-grained module placement generated by BMP. After we obtain this placement result, we then map it to the FPGA architecture and meet the position constraint. The Fig. 11 (b) is the final placement result. Note that the validation of a heterogeneous module has been discussed when creating the module.

In the implementation, to simplify the operation, we combine OP1 and OP4 together. The proposed BMP tool chooses a portion of the modules with smaller τ from the list to optimize the placement and the portion represents the percentage of the number of ratios for each module. Currently, we set the portion to be 0.5 to save the running time while still obtain good placement quality. The portion can be set to any value between 0 and 1 according to the system requirement. Simulated annealing is performed together with B*-Tree operation to explore the solution space. In each iteration, the options



(a) A non-overlap placement. (b) The (horizontal) B*-tree.

Fig. 9. A placement and corresponding B*-tree representation.

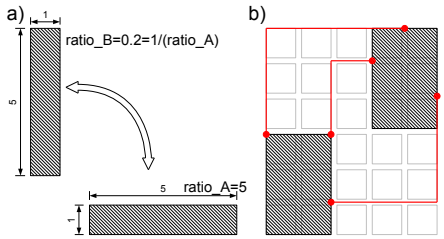
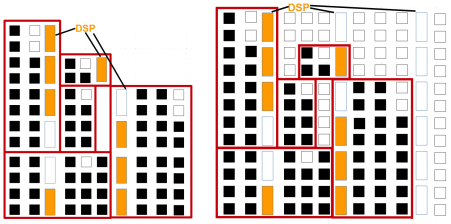


Fig. 10. Module rotation and wirelength estimation.(a) Rotation of a module. (b) Red line shows wirelength estimation from two modules.



(a) Coarse-grained module placement generated by BMP.

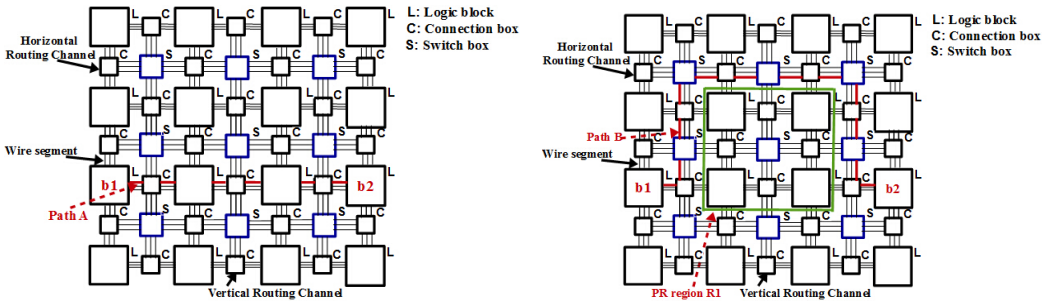
(b) Coarse-grained module placement after mapping to the FPGA architecture.

Fig. 11. Coarse-grained module placement results

OP2, OP3 and OP1+OP4, are randomly selected by the tool. Different weights are given to the operations to control the probability of choosing each operation, which are explored in the experiments.

$$Cost_{system} = \alpha \frac{A}{A^*} + (1-\alpha)((\beta)\frac{D_T}{D_T^*} + (1-\beta)\frac{D_M}{D_M^*}) \quad (1)$$

After each operation, we map the placement result generated by BMP to the FPGA architecture and evaluate its quality. The cost function [Wang and Wong 1991] is calculated as shown in Eq. 1. Here A and A^* represent the current total and average area, respectively. D_M and D_T are current total module delay and total track delay (wirelength). D_M^* and D_T^* are the average module delay and track delay. We divide the current results by the average value to normalize the results. The α is a parameter to bias the cost function to area or delay. The β is a proposed value to balance the weight between the inter and intra delay. Both parameters are explored in Section 8. The delay of each module is already obtained from the library. Thus, we only need to calculate the wirelength for the nets among the modules. The track delay (wirelength) is calculated by port-to-port wirelength between two modules [Kennings and Markov 2000] as shown in Fig. 10(b). With the pre-placement modules, we can get the exact location of the IOs of the modules and hence, more accurate wirelength can be obtained compared to the previous simple Manhattan distance between the center points of the modules. If the module needs to connect with the IO pads of the circuit, the delay is similarly estimated, assuming the module connects with the available IO pads determined by the minimum total wirelength.



(a) Normal routing in VPR. (b) PR-aware routing.

Fig. 12. Normal routing in VPR and PR-aware routing.

6. PR-AWARE ROUTING

After getting the placement results, routing is performed to connect the modules. We propose a PR-aware router to consider the routing restriction for PR regions without adding extra logic. An example of the PR-aware routing and normal routing in VPR is shown in the Fig. 12. Fig. 12 (a) shows that the path A from b1 to b2 connects directly as the red path shown. However, path B can not go through the PR region R1 (green bounding box) shown in the Fig. 12(b) and it finds another way as the red path shown. Next, we investigate different routing cases and propose a corresponding interface design in this section.

Firstly, we divide nets into four types based on the source and sink locations of nets since the source and sink locations of a net determine whether the net can go through the PR region. Fig. 13 shows an example for different types of connections existing in the PR-aware routing. The FPGA contains 6×6 CLBs. We assume that it has two PR regions and some static regions. The regions with green CLBs (left-top region R1) and yellow CLBs (right-bottom region R2) are the only two PR regions while the orange CLBs are the static region. The CLBs with white color are not used and other colored CLBs indicate that they are currently used. There are four types of nets connecting the static or reconfigurable modules. We describe their routing rules as follows.

— Type 1: If the source and sink are in two PR regions A and B respectively, the net routing can go through region A, B and the static regions, but not other PR regions.

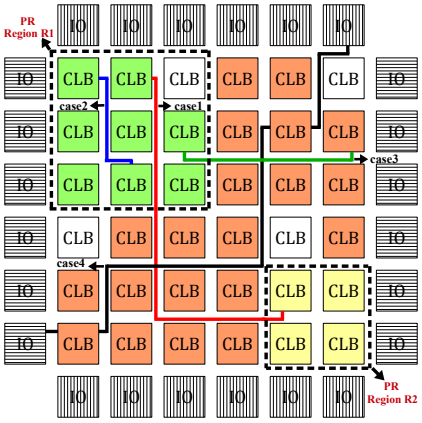


Fig. 13. Four types of connections of nets.

— Type 2: If the source and sink are all in the same PR region A, then the net routing is limited within the region A.
— Type 3: If the source in a PR region A and the sink in a static region C, the net routing can go through A and all the static regions.
— Type 4: If the source and sink are all in the static regions, the net routing can go through all static regions.

The PR-aware global routing also determines the virtual pins (tracks) needed for connecting the PR modules and static region for supporting PR and we call these pin-to-wire connections [Shah and Rose 2012]. We describe the detailed pin-to-wire interface design in the next section.

6.1. Pin-to-wire Routing Interface Design

We adopt the pin-to-wire connection framework and define the detailed interface to support PR. The pin-to-wire connections mean creating connections from output pins of logic blocks to specific wire segments in the network, or from specific wire segments to input pins [Shah and Rose 2012]. We define virtual pins (tracks or wire segments) to connect the PR regions and static regions. The PR regions are connected to tracks as its virtual IOs and the virtual IOs are fixed during the PR operation to avoid the interference with the static routing. By this way, the PR regions are connected to the static regions without using the proxy logic.

We first setup a PR-aware global routing for the whole design which contains all the modules (static and PR modules). After the initial PR-aware global routing, we can identify the tracks which connect one PR region to other regions. The first track in the static region that the IO port of the PR region connects to is defined as the virtual pin of the PR region. We discuss different cases as illustrated in the Fig 14. Here we assume that region A and B are the only two PR regions and other parts are static regions.

— Case 1: The source port and sink port are in different PR regions A and B, and their connection is through a switch box. Since there is no track in the static region along the path, we define the virtual pin to be the track A1 (red line) in the region A and track B1 (blue line) in the region B.

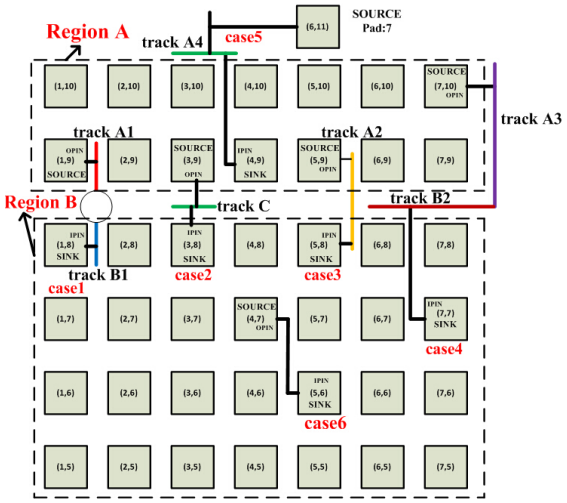


Fig. 14. Defining interface according to different types of nets.

— Case 2: The source port and sink port are in different PR regions A and B, and the ports are connected through a track C between the two regions. Then the track C (green line) is defined as the virtual pin for both regions.
— Case 3: The source port and sink port are in different PR regions A and B. It is similar as case 1, but a track A2 across the two regions can directly connect the ports. Thus we define A2 (yellow line) track as the virtual pin for both regions.
— Case 4: The source port and sink port are in different PR regions A and B, and there are more than one tracks between them. Thus we define track A3 (purple line) as the virtual pin for track A, and track B2 (red line) as the virtual pin for region B.
— Case 5: The source port and sink port are in one PR region, such as region A, and the static region, such as the pad. We only need to define the virtual pin the PR region. Following the rule, the track A4 (green line) can be defined as the virtual pin for region A.
— Case 6: The source port and the sink port are all in the same region (all are in the static region or in the PR region). We do not need to define interface for this net. Since all the content in the PR region can be reconfigured and re-routed.

We use these defined virtual pins or tracks for later PR operation. If a PR region needs PR operation, the context in the specific region is modified while other parts keep unchanged. During PR operation, the new context needs to connect to the virtual pin and through it connects to the static region or other PR region. Local placement and routing are needed to connect the PR context to the virtual pins as discussed in the next section.

7. LOCAL PLACEMENT AND ROUTING IN PR REGION

As we have discussed in Section 4, the size of the reconfigurable module or region is determined by the largest context of the module. To place the other different contexts in the PR region, we need to perform local placement and routing to fit the context in the region and connect the context IOs with the virtual pins which are fixed during the global routing.

We divide local placement into two types. In the first case, some ratio of the pre-synthesized module of the new context can be directly fitted in the PR region. Since the contexts stored in the library are already placed and routed well, we take it as a reference and place the module as a whole inside the region using the wirelength or delay as the optimized objectives. In the second case, the new context cannot be fit in the PR region directly. In this case, we need to place the logic of the new context using tile-based placement in the PR region and route the whole region. Fig. 15 shows an

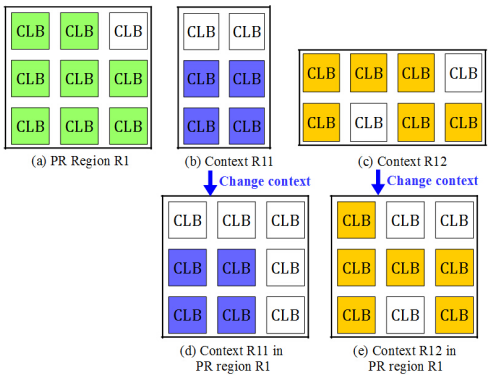


Fig. 15. Local placement in a PR region.

Table I. Benchmarks in VPR Suite

Arch Name	BLEs	RAM Startcol	RAM Gap	RAM Height	DSP Startcol	DSP Gap	DSP Height
Architecture 1	10	4	8	6	2	8	4
Architecture 2	4	3	5	2	5	5	4

Table II. Benchmarks in VPR Suite

circuit name	modules	CLB	RAM	DSP	circuit name	modules	CLB	RAM	DSP
raygentop	15	427	7	1	stereovision1	21	4001	0	0
bgm	32	8290	0	0	stereovision2	14	7618	0	0
LU32PEEng	4	17897	32	150	boundtop	13	558	0	0
mkPktMerge	4	34	0	15	mcml	5	15504	30	38
mkSMAdapter4B	4	393	0	5	mkDelayWorker32B	8	1117	0	41
LU8PEEng	4	5102	8	45					

example for the local placement in a PR region. We use the PR region R1 from the Fig. 13 as shown in (a). Fig. 15 (b) and (c) are two contexts: R11 and R12 which need to be loaded in the PR region R1. Fig. 15 (d) shows that the context R11 can be placed in the PR region R1 directly, and Fig. 15 (e) shows that the context R12 can not be placed in the PR region directly and thus we need to do tile-based local placement for this context in the PR region.

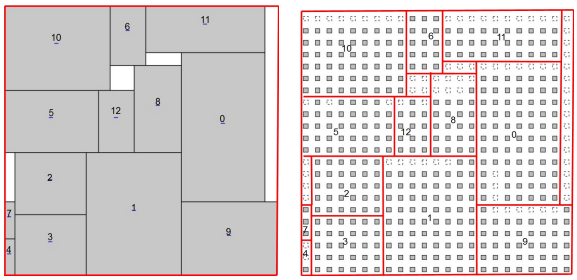
As for the local routing, note that we have considered the IOs needed for all the contexts of a reconfigurable module as discussed in Section 4. Hence, the virtual pins needed for this context have been fixed during the global routing. The local routing only needs to connect the context with its corresponding virtual pins through the normal routing. If the new context has no path to make connections to the previously chosen virtual pin interfaces. We can adopt the following two ways. 1. We do re-placement of the logic in the modules which are to connect to the virtual pin interface. 2. We change the framework from scratch which is to redo the flow for the whole design. It is similar to the VLSI and FPGA mapping flow that when the routing failed, the flow can reroute (*i.e.*, re-placement and re-routing).

8. EXPERIMENTAL RESULTS AND DISCUSSION

To demonstrate the performance of the proposed mapping flow and explore the design space of placer BMP, benchmarks for library-based placement and routing flow are first created to test the tools. Then the results of the library-based mapping flow is compared with the tile-based flow to demonstrate the trade-off in area, delay and execution time. Finally, the design space exploration is performed to optimize the parameters of the BMP placer. The simulations are run on IBM server x3650 with Intel Xeon(R) CPU and 42 GB DDR2 RAM. In general, we are using the architecture file which is similar to k6_N10_memDepth16384_memData64_40nm_timing, with the VTR project. The experiments are performed on this base architecture and its homogeneous variation which has the same architecture parameters except without the RAMs and DSPs. The two architectures are denoted as Architecture 2 and 4. The channel width is set to be 200 and the percentages for length-4, length-2 and length-1 wires are 60%, 20% and 20%, respectively. In order to further evaluate our flow in different architectures, we changed the parameters of Architecture 2 and created a new architecture file, denoted as Architecture 1, whose homogeneous variation is Architecture 3 correspondingly. The parameters for the Architecture 1 and 2 are summarized in Table I.

8.1. Benchmarks for Library-based Placement

Due to the unavailability of the benchmarks for module-based mapping, we select and modify the cases in the VPR suite for demonstrating the performance and functionality of the proposed library-based placement. Information of the cases are shown in Table II. The number of modules ranges from 4 to 32 for current case set. We also



(a) Boundtop floorplan. (b) Boundtop placement.

Fig. 16. Floorplan and placement results for the case boundtop.

Table III. Resource Utilization for all the cases

circuit name	CLBU	DSPU	RAMU	circuit name	CLBU	DSPU	RAMU
LU32PEEng	68%	2%	4%	LU8PEEng	80%	2%	4%
mcml	80%	2%	1%	mkDelayWorker32B	63%	0	14%
mkPktMerge	27%	0	68%	mkSMAdapter4B	88%	0	8%
raygentop	79%	22%	1%	bgm	83%	0	0
boundtop	81%	0	0	stereovision2	89%	0	0
stereovision1	89%	0	0				

show the number of tiles contained to compare the difference in the size among these cases. The absolute tile values are for reference only since it varies in different architectures. We decompose each of the verilog file into multiple verilog files according to subfunctions, which are known as modules. For each reconfigurable module, multiple contexts are created correspondingly for pre-defined functions. For the module-based applications, identifying connections between each module relies on the name of each inputs and outputs. Ports of the same name would be connected together to form the whole function in the netlist. Since depending on the benchmarks, different modules may have their own best ratios. We evaluated some typical ratios within the range of 0.1 to 1.0 with step 0.1 and also their inversions. We observed that except the extreme cases of very wide (ratio 10) or tall modules (ratio 0.1), the area-delay results of other ratios do not differ very much. In order to provide the tool a good flexibility to choose different ratio of modules according to different benchmark requirements and at the same time limit the number of total ratios to a small number, we choose 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.6, 2.5, 5.0 for the case of $K = 9$. In the experiments, we assume all the modules are stored in the library.

8.2. Library-based Placement Results and Discussion

First we demonstrate a placement result of benchmark *Boundtop* in Fig. 16(a). The red rectangle (the outside boundary) is the area of the benchmark. The grey area is the floorplan of each module of the logic function. White area is not placed with any modules and is taken as the waste area due to the library-based placement. The IOs are not shown here which is around the red area. We index each module with a specific number starting from 0. Hence, there are totally 13 modules in the benchmark. Fig. 16(b) shows the placement results on VPR. The red area are the corresponding mapping area of each module showed in Fig. 16(a). The proposed placement also supports heterogeneous components as discussed, *e.g.* internal memory, multiplier. Currently, the library-based flow deals with the low and high utilization designs in the same manner. We take each module as a function module and place them in the chip. The utilization rates of the cases are listed in Table III. CLBU, DSPU and RAMU represent the utilization of CLB, DSP and RAM. The experiments show that our flow

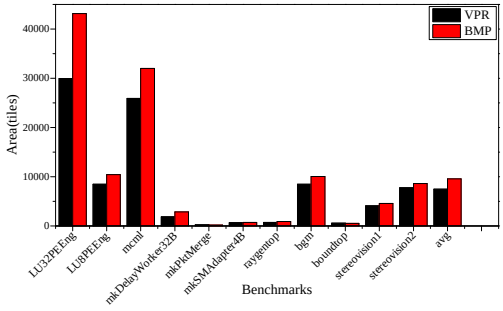


Fig. 17. Area comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.

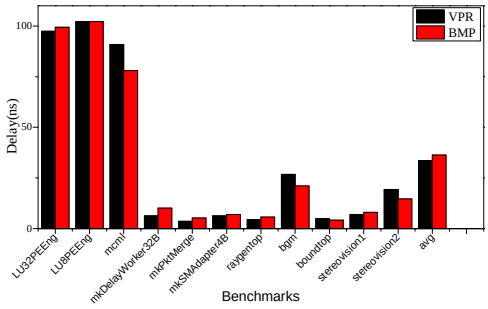


Fig. 18. Delay comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.

succeeds for the placement and routing of all the cases, even when the utilization rate of some case is high. Next, we compare the mapping results between the tile-based flow and our proposed flow to discuss the area and delay overhead incurred by the library-based approach, and also on the other hand, the saving on the execution time. Note that since VPR routing does not support PR, in order to fairly compare with the tile-based flow, we assume all the modules are static.

8.2.1. Area. Fig. 17 shows the area comparison of each benchmark and average area comparison of all the cases. The black (left, VPR) bars introduce the tile-based results generated by the original VPR flow and the red (right, BMP) bars demonstrate the library-based placement results using the proposed flow. Theoretically, the library-based placement has extra area cost because of the non-placed area demonstrated in Fig. 16. Meanwhile, the amount of the logic tiles in usage for each benchmark should be the same. Thus, the differences between the black bars and red bars mainly result from the extra cost. As the boundary (red) line of the bounding box showed in the Fig. 16, the extra cost of each benchmark varies from -22% to 55%. The bars for average area comparison in the last column shows that our approach has 17% extra area cost more than tile-based results. Allowing more iterations in the searching may reduce the wasted area between the modules. Moreover, the blank area inside the module depends on the benchmark, however, it can also be minimized through properly selecting the aspect ratio. Note that reducing the waste area may also reduce the delay, since the track delay is proportional to the routing distance in the FPGAs.

8.2.2. Delay. We demonstrate the delay comparison in Fig. 18. The black (left, VPR) bars and red (right, BMP) bars show that the delay of the critical path of each bench-

mark generated by the original VPR and by BMP, respectively. We can see that our flow gives better delay for 5 cases and worse delay for 6 cases. The average delay comparison in the last column shows that the average delay is slightly worse than the VPR around 8.2%. The cases of LU8PEEng, mcml, bgm, boundtop and stereovision2 demonstrate delay is improved by 0.03%-24% as showed in Fig. 18. It is because that the modules selected from the candidate list are with smaller τ. It gives the tool a good initial placement to ease the optimization. Furthermore, each module can be regarded as a cluster (the term is commonly used in FPGA field), for some benchmarks, it may reduce the critical path. Thus, later it may ease the routing too and enable better performance. The tile-based VPR flow needs to search in larger solution space in placement stage which may result in less optimized results in the same amount of time. Moreover, module with different ratios may provide better design for that module, it enables better final placement results. For the benchmarks with poor delay, the situation is the opposite. The cluster does not catch the critical path well and the critical path may across several modules and hence, trigger long delay. Tile-based placement managed to optimize across the modules and place the critical path together while the library-based could not achieve it. One thing to be noticed is that the preparation of different ratios of a module is similar to compiling the module for multiple rounds with different seeds. However, the efforts can be amortized through module reuse in later designs. If we allow the VPR to run same number of rounds with different seeds and select the best results, the delay of VPR flow can be improved by 3%.

8.2.3. Execution Time. The total execution time of the placement and routing of benchmarks for the original VPR and BMP are shown in Fig. 19 with black (left, VPR) and red (right, BMP) bars, respectively using log10 scale. Execution time includes placement and routing stage. Assuming the module information is available in the library, the library-based mapping flow has significantly better execution time in all benchmarks. For benchmark with a large number of tiles, the improvement can achieve 96%. For all the benchmarks, it has 65% improvement in average. The reason for the improvement is obviously from the reduction of searching space for solution. Compared to the original VPR flow, the number of modules are largely reduced for several orders of magnitude. At the same time, the reduction in the module number also leads to a simple routing. All connections between modules are relatively close and thus the tool has no need to trace and optimize for a long path.

The above discussion is for the situation that we have the modules' information available. If there is no module library or pre-synthesis results, the tool has to run a trial round for all the modules. The module based placer (BMP) needs modules with various aspect ratios, which take more trial rounds. However, since it is a part of floor-

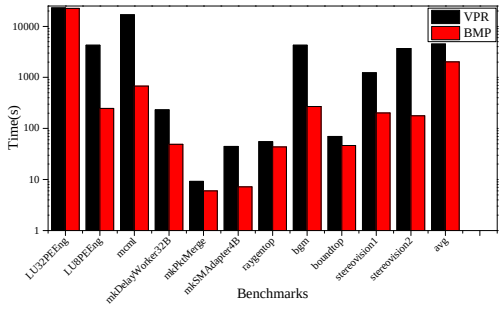
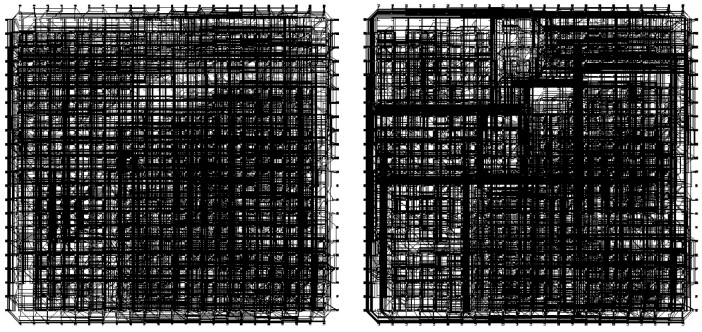


Fig. 19. Execution time comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.



(a) General. (b) PR-aware.

Fig. 20. Boundtop routing result without/with PR-aware.

Table IV. BMP Comparison Between Modules with Different Ratios and Modules with Only One Ratio

circuit name	one ratio		multi-ratio			
	Area	Delay	Area	AreaIncrease	Delay	DelayImprove
LU32PEEng	40610	9.89E-08	43130	6.21%	9.73E-08	-1.66%
LU8PEEng	9600	1.05E-07	10437	8.72%	1.00E-07	-4.77%
mcml	31439	7.76E-08	32016	1.84%	7.63E-08	-1.72%
mkDelayWorker32B	2856	9.34E-09	2860	0.14%	9.43E-09	0.94%
stereovision2	8533	1.43E-08	8625	1.08%	1.30E-08	-9.54%
Average	18607.6	6.11E-08	19413.6	3.60%	5.93E-08	-3.35%

planning search, the total running time considering module generation is similar with VPR flow to achieve the same placement results. Moreover, note that the trial rounds are all independent, we can use multi-threaded processing to speed up the execution.

8.3. PR-aware Global Routing Results

The difference between PR-aware router and the router of VPR is that PR-aware router cannot use the routing resource of all PR regions. Fig 20 shows the routing result without PR-aware and with PR-aware. Fig 20(a) shows the routing result without PR-aware for the case boundtop. Assuming that all the modules are PR modules, the routing resource for the static regions is limited to the tracks between modules and the tracks between the modules and IO pads. Fig. 20(b) shows the routing result with PR-aware. Compared to the routing result without PR-aware in the Fig 20(a), routing result with PR-aware in Fig. 20(b) shows more congestion between modules. Due to the resource limitations of PR-aware routing, it can be expected that more routing efforts have to be made to find a feasible routing. The study of pin-to-wire routing [Shah and Rose 2012] has observed a more than twice increase of routing efforts assuming using 30% more routing tracks than normal routing.

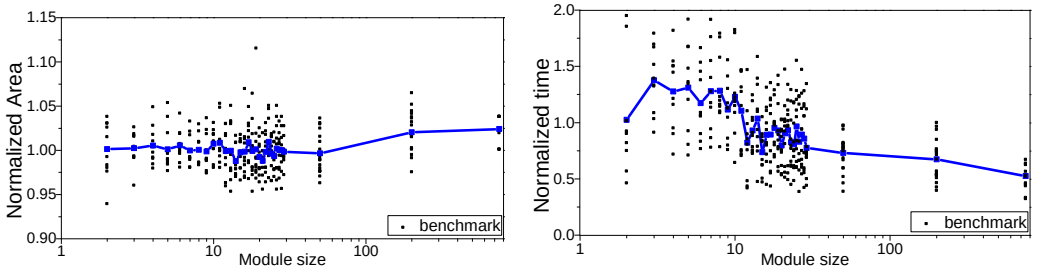
8.4. Design Space Exploration

The proposed mapping flow searches solution randomly at beginning which may result in widely various results. Thus, it is set to run 10 rounds for each benchmark of each group of parameters in the exploration. It may give results of distinctive deviation, but multi-round simulation helps to reduce impact of the deviations. Then, we average across benchmarks and normalize the results of all benchmarks to eliminate differences in logic complexity among benchmarks. Without loss of generality, we take the case bgm for exploration and it totally has 32 modules. The distribution of module sizes is as follows: there are 12 cases with CLB number less than 10, 9 cases with CLB number less than 200 and 11 cases with CLB number bigger than 600.

8.4.1. Reducing Searching Space. Searching space of a solution is the main factor for the execution time. To reduce the searching space, we propose a reduction method to lower down the search space for selecting the modules. We set the smaller modules to a fixed aspect ratio, since they can easily fit in an empty slot and do not have big impact on the total area. It indicates that original searching space should be $modules \times ratios$, now it becomes $modules_{large} \times ratios + modules_{small}$. Here we need to discuss what is the proper threshold value for defining small module. We compare the experiment results between modules with different ratios and modules with only one ratio to show the effects of the module ratios for area and delay, and the comparison results are listed in the Table IV. From the table, we can see that we can get better delay when we consider different ratios of modules compared with module with only one ratio. The average delay improvement can achieve 3.35% with area increase slightly compared to the case of using one ratio. The reason for delay improvement is that it can choose the best of several different ratios of each module. We also show a design space exploration result of the size threshold in Fig. 21. From the figure, we can see that with the threshold increase, the area of the placement increases. It is because when the threshold increases, more modules are placed with a fixed ratio, which reduces the placement flexibility and leads to a suboptimal placement. If we look into the details of the curve, we can see that there are fluctuations around the mean value. The fluctuations are from the simulated annealing algorithm and the differences in the benchmark set. Fig. 21(b) shows a trend that the larger module size can reduce execution time. Especially, when all the modules have only one ratio, its runtime reduce largely. In summary, from the normalized area graph Fig. 21(a) and time graph Fig. 21(b), we can see that blue line (the average line) shows that the number from 13 to 16 is a proper number for defining the small module since the threshold larger than this has negative impacts to the overall optimization results. It is because when larger module is set to fixed aspect ratio, the solution space is smaller and the algorithm can quickly find the optimal point, but the area and the runtime generally are conflicting objectives. We can see from the exploration that according to characteristics of the targeting benchmark set, threshold can be set to the value which optimizes the area and delay together. For our benchmarks, the value can be around 14.

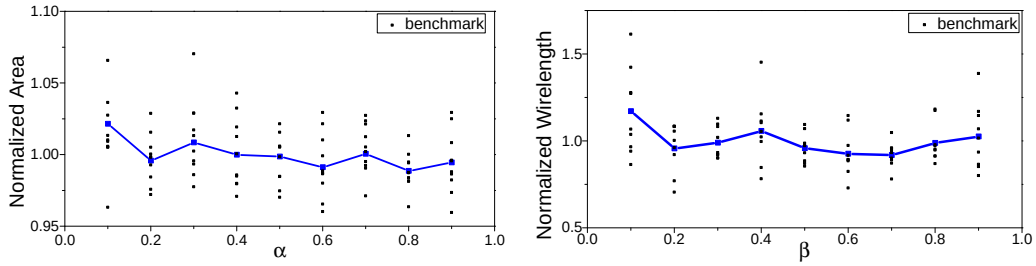
8.4.2. Parameters α and β in Eq. 1. Parameter α and β of the cost function are two important parameters to guide CAD tools in placement. Quality of placement and execution time are strongly related to these two parameters. We demonstrate design space explorations for optimization of the two parameters here.

We explore α from 0.1 to 0.9 with step of 0.1, and also β is set from 0.1 to 0.9 with step of 0.1. Fig. 22(a) shows the impact of variation of α on the normalized area and deviation for the benchmarks. The area is related to α with around -4% to 7% deviation



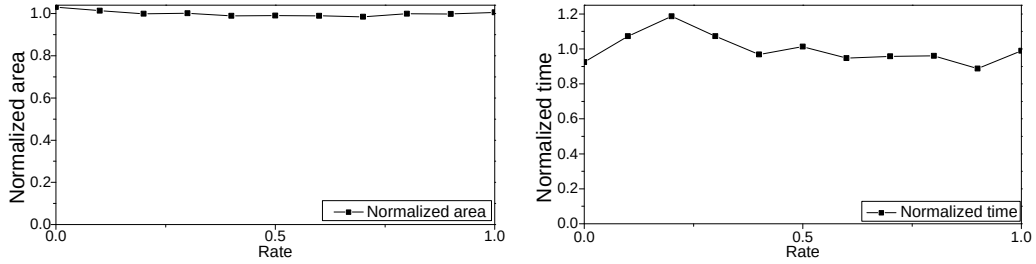
(a) Module size vs. area. (b) Module size vs. computation time.

Fig. 21. Design space exploration for definition of small module size.



(a) Parameters α (b) Parameters β

Fig. 22. Design space exploration for parameters in the cost function.



(a) Move rate vs. area. (b) Move rate vs. computation time.

Fig. 23. Design space exploration for rate of moving operation.

from normalized average area for different values of α when β is stepping from 0.1 to 0.9. We can see that when α ranges from 0.5 to 0.9, the algorithm give best area results. This is because that when $\alpha > 0.5$, the cost function emphasizes area more. The best point can be 0.6 and 0.8.

The β values are percentage between wirelength (track delay) and module delay in the placement. To discuss impact of β on the wirelength (track delay), we first preset α from 0.1 to 0.9 and β is varied from 0.1 to 0.9 with step of 0.1. The results of average wirelength (track delay) and deviation to mean value with various β are showed in Fig. 22(b). We can see that when when β ranges from 0.5 to 0.7, the algorithm give best wirelength (track delay) results. The best point can be 0.6 and 0.7.

8.4.3. Probability of Operations in SA. Theoretically, there are four operations in the proposed modified B*-tree representation. We merge OP1 and OP4 and then load all modules with various aspect ratio into computation. Hence, the weight for loading operation is 1. We only discuss the weight for the rest operations including moving module to other empty place (OP2) and swapping two modules (OP3). Their weights sum up to be 1. Fig. 23(a) demonstrates impact of moving rate on the area. We simulate the benchmarks with various moving rates in many rounds and then average and normalize the result. We can see that the moving rate has less impact on the area result. Area decreases slightly when moving rate ranges from 0 to 0.6 and it increases slightly when moving rate ranges from 0.6 to 0.8. The moving rate is zero, which indicates there is no moving operation. Fig. 23(b) shows moving rate impact on the execution time. However, we can see the average time varied a lot with the moving rate change. From the exploration, the moving rates ranging from 0.6 to 0.9 can improve the execution time and it impacts the area slightly.

Table V. Comparison between VPR and Our Proposed Flow in Different architectures.

circuit name	ChannelWidth	VPR			BMP		
		Area	Delay	Time	Area	Delay	Time
Architecture 1							
LU32PEEng	200	10000	1.53E-07	18864.55	14694	1.55836E-07	1896.91
LU8PEEng	200	2916	1.52543E-07	966.47	4186	1.51457E-07	209.23
mcml	200	8464	1.11432E-07	5271.82	10212	1.10583E-07	514
mkDelayWorker32B	200	1764	7.77673E-09	135.23	1849	9.12278E-09	82.2
mkPktMerge	200	676	4.02695E-09	17.57	624	5.94131E-09	26.49
mkSMAdapter4B	200	324	7.73036E-09	23.91	418	8.67654E-09	6.18
raygentop	200	289	6.66207E-09	27.46	414	7.63094E-09	32.4
Average		1	1	1	25.79%	13.15%	-43.31%
Architecture 2							
LU32PEEng	250	29929	9.71165E-08	20057.85	43130	9.88503E-08	1962.22
LU8PEEng	250	8464	1.01784E-07	4288.78	10437	1.01821E-07	198.84
mcml	250	25921	9.07892E-08	16882.24	32016	7.79289E-08	866.83
mkDelayWorker32B	250	1849	6.21091E-09	237.99	2860	1.01263E-08	59.2
mkPktMerge	250	225	3.60412E-09	10.11	176	5.31364E-09	6.48
mkSMAdapter4B	250	676	6.15022E-09	46.21	704	7.048E-09	8.83
raygentop	250	729	4.51367E-09	56.12	888	5.67742E-09	45.63
Average		1	1	1	21.40%	19.79%	-70.15%
Architecture 3							
bgm	300	3481	3.40905E-08	2068.89	3922	3.2529E-08	202.19
boundtop	300	256	7.82643E-09	47.76	266	7.35228E-09	47.22
stereovision1	300	1681	9.57075E-09	505.17	2002	1.02107E-08	143.56
stereovision2	300	3136	1.61955E-08	1644.56	3312	1.74035E-08	111.36
Average		1	1	1	10.32%	0.88%	-64.04%
Architecture 4							
bgm	150	8464	2.67565E-08	4287.94	10064	2.10551E-08	245.61
boundtop	150	576	5.04455E-09	68.35	525	4.25568E-09	44.71
stereovision1	150	4096	6.85571E-09	1217.05	4582	7.8015E-09	193.06
stereovision2	150	7744	1.94235E-08	3691.04	8625	1.33106E-08	155.31
Average		1	1	1	8.32%	-13.66%	-77.20%

8.5. Running in Different Architectures

In order to evaluate the efficiency of our flow in different architectures, we perform the experiments in four architectures as shown in Table I. Moreover, we also set channel width to be different values to evaluate the impact of different channel widths. The experiments show that basically the area increases all the time. The homogeneous architectures tend to have less area overhead. It is because that the fixed position of DSPs and RAMs increases the challenge of module placement and more easily incurs area waste. Similarly the delay in homogeneous architecture is better than that of heterogeneous architecture, especially for Architecture 4. The possible reason should be that our proposed BMP can get compact solution and there is no position constraint. The Architecture 4 achieves better delay than Architecture 3 because except the benchmark stereovision1, other three benchmarks all favor the small CLBs in Architecture 4 and achieve delay improvement. Table V shows that the area increase ranges from 8.32% to 25.79%, delay varies from -13.66% to 19.79%, and running time improves by 43.31% to 77.2%. It shows the efficiency of the proposed flow in various architectures.

9. CONCLUSIONS

Our work proposes a library-based mapping flow which supports the partial run-time reconfiguration and replacement for multi-context functions in a PR region. Furthermore it can reuse the resource in the module library to reduce the compilation time and to build large circuits. The proposed module-based (BMP) placer uses the modified B*-Tree representation to optimize the floorplanning and placement of the modules with the consideration of flexible module ratio. The corresponding parameters for cost functions and searching algorithms are explored in the experiments. Compared to the original tile-based flow, the delay of the proposed module based flow is slightly worse

than the VPR around 8.2% in a basic architecture, but with significant running time reduction around 65% with acceptable area cost due to the empty space which shows the efficiency of the proposed flow. We also develop the pin-to-wire interface to support the PR-aware routing without adding extra cost.

ACKNOWLEDGMENTS

This work is partly supported by a MoE AcRF Tier 2 grant (MOE2012-T2-1-126) in Singapore and by Grant R9336 from Hong Kong SAR.

REFERENCES

Altera. 2010. Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs. (2010). http://www.altera.com.

S. Areibi, G. Grewal, D. Banerji, and P. Du. 2007. Hierarchical FPGA placement. *Electrical and Computer Engineering, Canadian Journal of* 32, 1 (Winter 2007), 53–64.

P. Athanas and et al. 2007. Wires on Demand: Run-Time Communication Synthesis for Reconfigurable Computing. In *FPL, 2007*. 513–516.

Pritha Banerjee and et al. 2011. Floorplanning for Partially Reconfigurable FPGAs. *TCAD* 30, 1 (2011).

P. Banerjee, S. Sur-Kolay, and A. Bishnu. 2009. Fast Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs. *TCAD* 28, 5 (May 2009), 651–661.

Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. 2000. Fast Template Placement for Reconfigurable Computing Systems. *IEEE Design & Test of Computers* 17, 1 (2000), 68–83.

Christian Beckhoff and et al. 2013. Automatic Floorplanning and Interface Synthesis of Island Style Reconfigurable Systems with GOAHEAD. In *Architecture of Computing Systems (ARCS)*. Springer, 303–316.

Jeffrey M Carver, Richard Neil Pittman, and Alessandro Forin. 2009. Automatic Bus Macro Placement for Partially Reconfigurable FPGA Designs. In *FPGA*. 269–272.

R. Castro-lpez, F.V. Fernndez, O. Guerra-vinuesa, and . Rodrguez-vzquez. 2006. A Reuse-based Design Framework for Analog ICs. In *Reuse-Based Methodologies and Tools in the Design of Analog and Mixed-Signal Integrated Circuits*. Springer Netherlands, 27–62.

T.C. Chen and Y.W. Chang. 2006. Modern Floorplanning Based on B*-Tree and Fast Simulated Annealing. *TCAD* 25, 4 (2006), 637–650.

Yu-Chen Chen, Sheng-Yen Chen, and Yao-Wen Chang. 2014. Efficient and Effective Packing and Analytical Placement for Large-scale Heterogeneous FPGAs. In *ICCAD*. 647–654.

Lei Cheng and Martin DF Wong. 2006. Floorplan Design for Multimillion Gate FPGAs. *TCAD* 25, 12 (2006).

C. Claus and et al. 2007. An XDL-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems. In *Works. on Reconfigurable Computing Education*.

Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. 2011. Customizable Domain-specific Computing. *IEEE Design & Test of Computers* 28, 2 (2011), 6–15.

D.D. Gajski. 1999. IP-based design methodology. In *Design Automation Conference, 1999*. 43.

M. Gort and J.H. Anderson. 2012. Analytical placement for heterogeneous FPGAs. In *FPL, 2012*. 143–150.

Marcel Gort and Jason Anderson. 2014. Design Re-use for Compile Time Reduction in FPGA High-level Synthesis Flows. In *FPT*. 4–11.

Ruining He and et al. 2012. PDPR: Fine-Grained Placement for Dynamic Partially Reconfigurable FPGAs. In *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 350–356.

A. Hekmatpour, K. Goodnow, and H. Shah. 2005. Standards-compliant IP-based ASIC and SoC design. In *SOC Conference, 2005. Proceedings. IEEE International*. 322–323.

Bo Hu. 2006. Timing-Driven Placement for Heterogeneous Field Programmable Gate Array. In *ICCAD, 2006*. 383–388.

E. Hung, S.J.E. Wilton, Haile Yu, T.C.P. Chau, and P.H.W. Leong. 2009. A detailed delay path model for FPGAs. In *FPT 2009*. 96–103.

P. Jamieson, F. Gharibian, and L. Shannon. 2013. Supergenes in a genetic algorithm for heterogeneous FPGA placement. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*. 253–260.

Andrew A Kennings and Igor L Markov. 2000. Analytical Minimization of Half-perimeter Wirelength. In *ASPDAC*. 179–184.

D. Koch, C. Beckhoff, and J. Teich. 2008. ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS. In *FPL, 2008*. 119–124.

Dirk Koch, Christian Beckhoff, and Jim Torresen. 2010. Zero Logic Overhead Integration of Partially Reconfigurable Modules. In *SBCCI (SBCCI '10)*. 103–108.

C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. 2011. HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping. In *FCCM, 2011*. 117–124.

Nan Liu, Song Chen, and T. Yoshimura. 2011. Floorplanning for high utilization of heterogeneous FPGAs. In *ISQED, 2011*. 1–6.

T.S.T. Mak, P. Sedcole, P.Y.K. Cheung, and W. Luk. 2007. Average interconnection delay estimation for on-FPGA communication links. *Electronics Letters* 43, 17 (August 2007), 918–920.

Minkovich K. 2007. MCNC benchmarks. (2007). http://cadlab.cs.ucla.edu/~kirill/.

Alessio Montone, Marco D Santambrogio, Donatella Sciuto, and Seda Ogrenci Memik. 2010. Placement and Floorplanning in Dynamically Reconfigurable FPGAs. *TRETS* 3, 4 (2010), 24.

A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. 2002. Accurate area and delay estimators for FPGAs. In *DATE, 2002*. 862–869.

Jonathan Rose and et al. 2012. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *FPGA*. 77–86.

M. Samaranayake, H. Ji, and J. Ainscough. 2009. Module placement based on hierarchical force directed approach. In *Signals, Circuits and Systems (SCS), 2009 3rd International Conference on*. 1–6.

Yaska Sankar and Jonathan Rose. 1999. Trading Quality for Compile Time: Ultra-fast Placement for FPGAs. In *FPGA (FPGA '99)*. ACM, 157–166.

Navaratnasothie Selvakkumaran and et al. 2004. Multi-resource Aware Partitioning Algorithms for FPGAs with Heterogeneous Resources. In *FPGA (FPGA '04)*. 253–253.

N. Shah and J. Rose. 2012. On the difficulty of pin-to-wire routing in FPGAs. In *FPL, 2012*. 83–90.

Love Singhal and Elaheh Bozorgzadeh. 2006. Multi-layer Floorplanning on a Sequence of Reconfigurable Designs. In *FPL*. 1–8.

V. Sklyarov, I. Skliarova, P. Almeida, and M. Almeida. 2003. Design tools and reusable libraries for FPGA-based digital circuits. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*. 255–263.

Greg Stitt, Frank Vahid, and Shawn Nematbakhsh. 2004. Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems. *TECS* 3, 1 (2004), 218–232.

Nicolas Telle, Wayne Luk, and Ray CC Cheung. 2004. Customising Hardware Designs for Elliptic Curve Cryptography. In *Computer Systems: Architectures, Modeling, and Simulation*. Springer, 274–283.

Vereen, L. 2004. Soft FPGA Cores Attract Embedded Developers. (April 2004). http://www.embedded.com//showArticle.jhtml?articleID=19200183.

K. Vipin and S.A. Fahmy. 2011. Efficient Region Allocation for Adaptive Partial Reconfiguration. In *FPT*. 1–6.

Ting-Chi Wang and DF Wong. 1991. An Optimal Algorithm for Floorplan Area Optimization. In *DAC*. 180–186.

Xiaojun Wang and Miriam Leeser. September 2010. VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. *ACM Trans. Reconfigurable Technol. Syst.* (September 2010).

Xilinx. 2011a. Early Access PR User Guide. (2011). http://www.xilinx.com.

Xilinx. 2011b. Xilinx Partial Reconfiguration User Guide. (2011). http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ug702.pdf.

Xilinx. 2012. Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite. (2012). http://www.xilinx.com.

Liming Xiu. 2007. VLSI Circuit Design Methodology Demystified : A Conceptual Taxonomy. *IEEE Press.* (2007), 200.

Chang Xu, Wentai Zhang, and Guojie Luo. 2014. Analyzing the impact of heterogeneous blocks on FPGA placement quality. In *FPT, 2014*. 36–43.

Shaon Yousuf and Ann Gordon-Ross. 2010. DAPR: Design Automation for Partially Reconfigurable FPGAs. In *Engineering of Reconfigurable Systems and Algorithms (ERSA)*. 97–103.

Jun Yuan, Sheqin Dong, Xianlong Hong, and Yuliang Wu. 2005. LFF algorithm for heterogeneous FPGA floorplanning. In *ASP-DAC, 2005*, Vol. 2. 1123–1126 Vol. 2.

N.-E. Zergainoh, K. Popovici, A. Jerraya, and P. Urard. 2005. IP-block-based design environment for high-throughput VLSI dedicated digital signal processing systems. In *ASP-DAC, 2005*, Vol. 1. 612–618.

Received June 2015; revised November 2015; accepted March 2016