



**DESIGN AUTOMATION FLOW FOR PARTIAL
RUN-TIME RECONFIGURATION ON FPGAS**

MAO FUBING

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A THESIS SUBMITTED TO THE NANYANG TECHNOLOGICAL
UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY

2017

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Mao Fubing

Acknowledgments

There are many people that I would like to thank for their help and support throughout this thesis.

I am greatly indebted to my supervisors Professor Siew-Kei Lam at Nanyang Technological University, Professor Wei Zhang at Hong Kong University of Science and Technology and Professor Bingsheng He at National University of Singapore for their tremendous support, encouragement, and guidance during my graduate study. Their advice has made a profound impact on my study and life. They have helped me with a better way to approach the problems and solve them independently.

I thank the people who collaborate with me during my study including Dr. Yi-chung Chen at University of Pittsburgh, Professor Hai (Helen) Li at Duke University, Professor Yuchun Ma, Mr. Bo Feng at Tsinghua University and Professor Weichen Liu at NTU for their effort and help on my research.

I thank my friends and colleagues in the Hardware and Embedded Systems Lab (HESL) of NTU and Xtra Computing Group of NUS for their friendship and help during my study. In particular, thanks go to Dr. Jin Cui, Li Zhang, Hao Liang, Hanhua Qian, Yingnan Cui, Zhaojie Niu, Anran Wang, Xiaoning Wang, Yupeng Chen, Liwei Yang, Amelie Chi Zhou, Tao Luo, Xiaozhe Gu, Xiangwei Li, Liangliang Jiang, Sanjeev Kumar Das, Zeke Wang, Peng Sun, Qing Xu, Deheng Ye, Hao Chen, Xuntao Cheng. I am thankful to Chua Ngee Tat for his technical assistance in HESL. I also thank Professor Yang Liu and Gaoxi Xiao at Nanyang Nanyang Technological University for their partially financial support. I am grateful to all those who have helped and supported me throughout my PhD life.

My deepest appreciation goes to my family for their unending love and support.

Abstract

Field-Programmable Gate Array (FPGA) is a programmable hardware that allows post-manufacturing configuration to meet application-specific functionality and requirement. Partial Reconfiguration (PR) is an advanced feature in modern FPGAs that enables the configuration of the FPGA to be altered at runtime. This provides the means to maximize the utilization of the limited FPGA resources to support more functions and shorten the time to market of the product. However, there is a lack of efficient computer-aided design (CAD) tools for placement and routing that support partial reconfiguration on FPGAs. Traditional approaches usually rely on manual partitioning and placement, which is an error-prone and tedious process, and requires huge efforts and long development cycle due to the large design space. 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 reduce the synthesis time. Moreover, the commonly used modules can be pre-synthesized and stored in the library for design reuse to significantly lower the design time, verification time and development cost. To address the problems mentioned above, this research attempts to fill the gap by proposing an automatic mapping flow for efficient PR. The thesis has three major contributions.

Firstly, 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 considering area-delay products of each module with different ratios and optimizing area and delay of the final design. The flow supports both the static and reconfigurable modules. The modular information is represented in a B*-Tree structure, and the B*-Tree operations are amended

and used with Simulated Annealing (SA) to enable rapid exploration of the placement space. Different width-height ratios of the modules are exploited to achieve area and delay 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 on average with acceptable area and delay overhead compared to tile-based results from the Versatile Place and Route (VPR) tool through the reuse of module information in the library for the target architecture.

Secondly, we propose a dynamic module partitioning approach for the library based design flow to dynamically generate the appropriate shape of modules based on single-ratio modules in the library while efficiently utilizing the pre-placement module information. A set of rules are developed to select the most suitable module and determine the partition to minimize the area and delay of the placement without increasing much of the synthesis time. The proposed approach can adapt to different architectures and also address the fixed-outline constraint. Experiment results show that our approach can reduce the area by up to 10% with marginal increase in delay and acceptable runtime.

Finally, we explore the automatic workflow mapping in the interposer based multi-FPGA system. We propose a two-stage modular placement flow for interposer based multiple FPGAs aiming for delay optimization with the incorporation of a detailed interposer routing model for wirelength and delay estimation. We adopt the force-directed method for its global property to obtain an efficient solution as a starting point of the placement. Next, we employ simulated annealing (SA) for its efficiency and effectiveness in refining the initial solution. In order to speed up the refinement process, the hierarchical B*-tree (HB*-tree) is employed to enable a fast search and convergence. The experiment results demonstrate that our flow can achieve an efficient solution in reasonable time and can scale well for different design sizes.

Contents

Acknowledgements	vi
Abstract	viii
List of Figures	xiii
List of Tables	xv
List of Algorithms	xvii
List of Abbreviations	xix
1 Introduction	1
1.1 Field-Programmable Gate Arrays	1
1.2 Scope of Usage and Application Scenarios of Partial Reconfiguration	4
1.3 Motivation and Challenges	6
1.4 Research Objectives	9
1.5 Organization of the Thesis	11
2 Background and Literature Review	13
2.1 Background	13
2.1.1 A General Mapping Flow for the FPGA Design	13
2.1.2 Verilog-to-Routing (VTR) Flow for the FPGA Design	14
2.1.3 Partial Reconfiguration	16
2.2 Literature Review	19
2.2.1 Tile-based Placement	19
2.2.2 Clustering based FPGA Placement	20
2.2.3 Module-based Floorplanning and Placement	20
2.2.4 Modular Placement for 2D Single FPGA	23
2.2.5 Placement for 2D Multi-chip and 3D FPGA	23
2.2.6 Placement for 2.5D IC and FPGA with Interposer	24

2.2.7 PR-aware Routing	24
2.3 Summary	25
3 Library-based Placement and Routing in FPGAs with Support of Partial Reconfiguration	27
3.1 Introduction	27
3.2 Library-Based Mapping Flow	30
3.3 Module Library Construction	32
3.3.1 Synthesis of Reconfigurable Modules	32
3.3.2 Module Ratio Selection	33
3.3.3 Delay Estimation Model for the Modules with Specific Ratios . . .	36
3.4 Library-Based Placement	37
3.4.1 Problem Formulation	37
3.4.2 B*-tree Representation	38
3.4.3 B*-tree based Module Placement	39
3.5 PR-aware Routing	42
3.5.1 Pin-to-wire Routing Interface Design	45
3.6 Local Placement and Routing in PR Region	47
3.7 Experimental Results and Discussion	49
3.7.1 Benchmarks for Library-based Placement	50
3.7.2 Library-based Placement Results and Discussion	51
3.7.3 PR-aware Global Routing Results	56
3.7.4 Design Space Exploration	57
3.7.5 Running in Different Architectures	61
3.8 Summary	63
4 Dynamic Module Partitioning for Library based Placement on Heterogeneous FPGAs	65
4.1 Introduction	65
4.2 Problem Formulation	67
4.3 Our Proposed Dynamic Partitioning Flow for Library based Placement . .	68

4.3.1	B*-tree Based Module Placement with Dynamic Partitioning . . .	68
4.3.2	Partition Strategy	72
4.4	Experimental Results	76
4.4.1	Experimental Setup	76
4.4.2	Area, Delay and Runtime Comparison Considering Modules with One Ratio	77
4.4.3	Area, Delay and Runtime Comparison Considering Modules with Multiple Ratios	78
4.4.4	Area, Delay and Runtime Comparison for Different Architectures .	78
4.5	Summary	79
5	Modular Placement for Interposer based Multi-FPGA Systems	81
5.1	Introduction	81
5.2	Problem Formulation	83
5.3	Proposed Modular Placement Flow	84
5.3.1	Routing Model for Interposer	85
5.3.2	Force-directed Placement	86
5.3.3	Hierarchical B*-tree Based Module Placement	89
5.4	Experimental Results	93
5.4.1	Impact of Interposer	94
5.4.2	Impact of Different Module Ratios	95
5.4.3	Speed Up of the Placement Flow	97
5.5	Summary	98
6	Conclusions and Future Work	99
6.1	Conclusions	99
6.2	Future Work	101
6.2.1	Further Performance Optimization	101
6.2.2	Extension of Current Work	103
	List of publications	105

List of Figures

1.1	Basic FPGA structure without heterogeneous blocks.	2
1.2	Basic FPGA structure with heterogeneous blocks.	3
2.1	A general mapping flow for FPGAs.	14
2.2	Verilog-to-Routing (VTR) flow for FPGAs.	15
2.3	An example for Partial Reconfiguration.	18
3.1	Our proposed mapping flow for library based design.	30
3.2	Library based structure.	31
3.3	Two contexts (R1 and R2) of a reconfigurable module	32
3.4	Relation between shapes and delay for MCNC20 benchmark.	33
3.5	Relation between shapes and area for MCNC20 benchmark.	33
3.6	Various shapes and delay in two modules with the same logic.	34
3.7	Coarse-grained module with different ratios	35
3.8	Module delay estimation for a specific ratio.	36
3.9	A placement and corresponding B*-tree representation.	38
3.10	Module rotation and wirelength estimation.(a) Rotation of a module. (b) Red line shows wirelength estimation from two modules.	40
3.11	Coarse-grained module placement results	41
3.12	Normal routing in VPR and PR-aware routing.	43
3.13	Four types of connections of nets.	44
3.14	Defining interface according to different types of nets.	46
3.15	Local placement in a PR region.	48
3.16	Floorplan and placement results for the case boundtop.	51
3.17	Area comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.	53

3.18	Delay comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.	54
3.19	Execution time comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases. (The y axis is in log10 scale.)	55
3.20	Boundtop routing result without/with PR-aware.	56
3.21	Design space exploration for definition of small module size. (The x axis is in log10 scale.)	58
3.22	Design space exploration for parameters in the cost function.	59
3.23	Design space exploration for rate of moving operation.	61
4.1	Vertically Partition a module A into A1 and A2	70
4.2	Dynamic partitioning node n1 into two subnodes n11 and n12	71
4.3	Vertically Partition a module K into K1 and K2 based on RAM and DSP columns	73
4.4	Vertically Partition a module A into submodules A11 and A12	75
5.1	A general 2.5D FPGA with interposer.	82
5.2	Difference of wirelength estimation when using interposer compared traditional bounding box calculation.	85
5.3	Example for attraction force considering interposer.	88
5.4	An Hierarchical B*-tree. In the interposer level, it has 9 chips. Chip n5 has 4 modules in the chip level.	90
5.5	Chip-level packing: a sub-B*-tree of the chip n5 and its corresponding placement.	91
5.6	Interposer-level packing: a sub-B*-tree for chips and its corresponding placement	91

List of Tables

3.1	Architectures and Parameters in VPR Suite	49
3.2	Benchmarks in VPR Suite	50
3.3	Resource Utilization for all the Cases in the Final Design	52
3.4	BMP Comparison Between Modules with Different Ratios and Modules with Only One Ratio	57
3.5	Comparison between VPR and Our Proposed Flow in Different architectures.	62
4.1	Architectures Used in the Experiment	76
4.2	Benchmarks in VPR Suite	76
4.3	Comparison between Module Based Placement Flow and Our Proposed Ap- proach Considering Modules with only One Ratio	77
4.4	Comparison between Module Based Placement Flow Considering Multiple Ratios and Our Proposed Approach	78
4.5	Comparison between Module Based Placement Flow and Our Proposed Ap- proach for Different Architecture	79
5.1	Benchmarks from VPR Suite	93
5.2	Comparison between Constant Delay Model used in [HPB14] and Our Proposed Model in Enhanced Force-Directed Placement	94
5.3	Comparison between Constant Delay Model used in [HPB14] and Our Proposed Model in Our Proposed Flow	94
5.4	Comparison between Modules with only one ratio and different ratios in Enhanced Force-directed Placement	95
5.5	Comparison between Modules with only one ratio and different ratios in Our Proposed Flow	96
5.6	Comparison between Min-cut + SA and Our Proposed approach	97

List of Algorithms

4.1	The Rule for Choosing Modules for Partitioning	73
4.2	The Rule for Partitioning a Module	74

List of Abbreviations

FPGA	Field-Programmable Gate Array
PR	Partial Reconfiguration
CAD	Computer-Aided Design
SA	Simulated Annealing
VPR	Versatile Place and Route
HB*-tree	Hierarchical B*-tree
HDL	Hardware Description Languages
BRAM	Block Random Access Memory
DSP	Digital Signal Processing Block
SB	Switch Box
CB	Connection Box
CLB	Configurable Logic Block
ASIC	Application Specific Integrated Circuit
SOC	System-on-Chip
RTL	Register Transfer Level
VTR	Verilog-to-Routing
LUT	Look-up Table
FF	Flip-Flop
2D	Two Dimensional
2.5D	Two-and-a-Half-Dimensional
3D	Three Dimensional
IP	Intellectual Property
TSV	Through-Silicon Via
SLL	Super Long Line
SVM	Support Vector Machine

CHAPTER 1

Introduction

In the thesis, we propose a library-based placement and routing flow for Partial Run-time Reconfiguration on FPGAs. We introduce the concepts of the FPGA, Partial Run-time Reconfiguration, advantages of FPGAs and PR. We also introduce the problems, motivation, challenges and our research objectives. In this chapter, we first introduce the Field-Programmable Gate Array (FPGA) by describing its architecture and the advantages of dynamic partial reconfiguration in section 1.1. Next, we describe the scope of usage and application scenarios of partial run-time reconfiguration in section 1.2. Then, we present the motivation and challenges of our study in section 1.3. Finally, we summarize our research objectives in section 1.4. The organization of the thesis is given in section 1.5.

1.1 Field-Programmable Gate Arrays

FPGA is an integrated circuit [FMM12]. Customers and designers can configure or program it after it has been manufactured. It is a semiconductor device and has high flexibility for changing functions according to different requirements of applications. It uses the pre-fabricated logic blocks and programmable routing resource to implement different hardware functions. We use the hardware description languages (HDL) like VHDL and verilog to specify the FPGA configuration [Yas11, Aga15, FMM12]. The basic architecture of the FPGA mainly consists of three components: programmable logic blocks, programmable interconnects and IO (input/output) blocks. Programmable logic blocks implement logic functions and programmable interconnects connect different

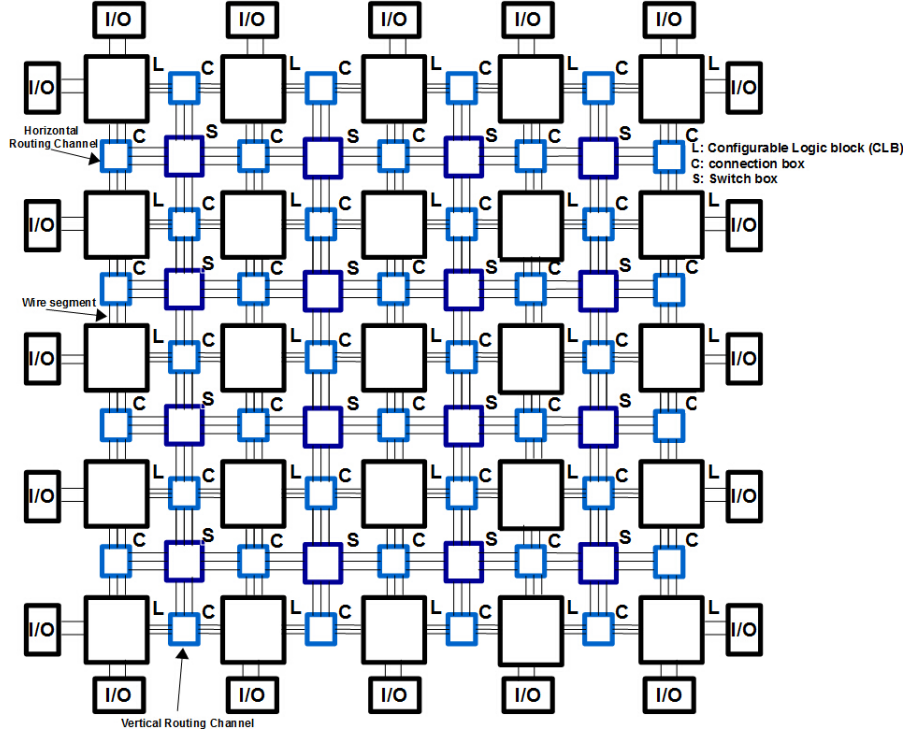


Figure 1.1: Basic FPGA structure without heterogeneous blocks.

components to implement the functions [Yas11, Aga15, FMM12]. IO blocks provide the connections between external components and internal logic of the FPGA. An illustration of an island-style FPGA architecture is shown in Fig. 1.1 and Fig. 1.2. Fig. 1.1 shows a basic FPGA structure without heterogeneous blocks, e.g, Block Random Access Memory (BRAM), Digital Signal Processing block (DSP) and Fig. 1.2 shows a basic structure that contains heterogeneous blocks. As shown in Fig. 1.1, the configurable logic blocks (CLBs) are arranged on a 2D array and they are connected by the programmable routing network. The IO blocks surround the array which is connected by the routing network [KTR08]. The routing network consists of pre-fabricated wire segments and programmable switches. The switch boxes (SB) are used to connect the horizontal and vertical routing tracks. The connection boxes (CB) are used to connect the logic blocks and routing network [Yas11, Aga15, FMM12].

FPGA market is popular and growing rapidly with different requirements of applications for different industries in the last decades. Compared to application specific integrated

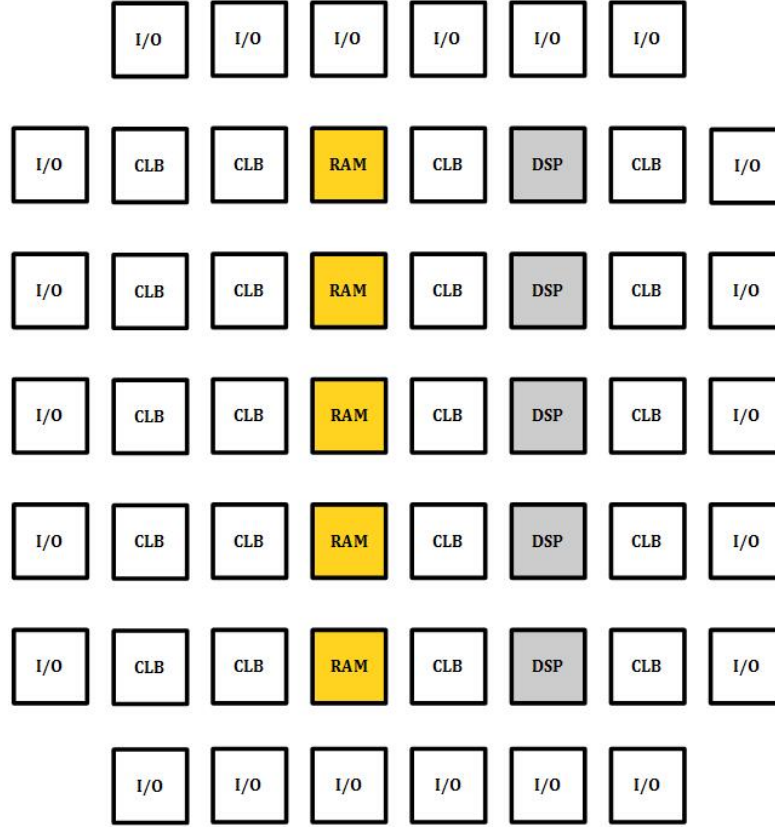


Figure 1.2: Basic FPGA structure with heterogeneous blocks.

circuits (ASICs), it has the advantage of flexibility due to their programmable feature (reconfiguration feature). Dynamic partial reconfiguration is a new concept in the development of the FPGA industry. Many application fields use this feature to meet different requirements of the applications. To meet the space requirements in small portable mobile systems, to create a flexible system-on-chip (SOC) and to dynamically change the hardware algorithms according to different requirements during runtime, we make use of dynamic partial reconfiguration. Dynamic partial reconfiguration enables the configuration of the FPGA to be altered at runtime. This provides the means to maximize the utilization of the limited FPGA resources to support more functions and shorten the time-to-market of the product [LFy09].

We summarize the advantages of the dynamic partial reconfiguration over full reconfiguration of FPGAs as follows.

- It allows the FPGA to change its hardware functionality during runtime.
- It improves fault tolerance. This is very important and valuable for mission-critical applications since devices running in such an environment cannot be interrupted. Some parts of the device can be configured while other parts are kept unchanged.
- It improves the resource utilization, the performance and the power consumption.
- It saves compilation time and reduce the configuration time since the system does not need to be reconfigured from the beginning and the partial reconfiguration files are much smaller than the full reconfiguration files.

1.2 Scope of Usage and Application Scenarios of Partial Reconfiguration

As we have described above, partial run-time reconfiguration has many benefits. However, not all systems benefit from partial runtime reconfiguration (e.g. all the parts of a system are active at any time) and not all FPGA vendors support partial run-time reconfiguration in their devices. Partial run-time reconfiguration needs to be supported by the FPGA and by the corresponding design tools. Parts of the FPGA fabric can be reconfigured without affecting other parts during runtime [KT10, Xil16]. The basic premise of Partial Reconfiguration is that the hardware resource of the device has the ability to time-multiplexed during runtime [Xil16]. We describe several different application scenarios below.

- Networked Multiport Interface [Xil16]

We can identify, isolate and implement time-independent functions as Reconfigurable Modules and swap them in and out of a single device according to the requirements. A 40G OTN muxponder application is a typical example. The ports of the client side of the muxponder supports multiple interface protocols. However, it is impossible for the system to predict which protocol will be used before the FPGA is configured. They implement every possible interface protocol for every port. However, it is an

inefficient design since each port only take only one type of the standards at any point in time. Partial Reconfiguration provides a more efficient design and it makes each of the port interface a Reconfigurable Module which helps to eliminate the MUX elements required to connect multiple protocol engines to one port [Xil16].

- Configuration by Means of Standard Bus Interface [Xil16]

Partial Reconfiguration can use an interface standard that is more compatible with the system architecture to create a new configuration port. For example, a system host connects to a FPGA through the PCIe and FPGA is a peripheral on a PCIe bus. The system host can configure the FPGA through PCIe connection. When the power is on, the FPGA must be configured with a full BIT file. However, the PCIe interface and connection to the internal configuration access port (ICAP) may be the only content in the full BIT file. We can use the bitstream compression to reduce the size and reconfiguration time of the initial device load. It can help FPGA configuration meet PCIe enumeration. The system host could use the partial BIT file downloaded through the PCIe port to configure the majority of the FPGA functionality. The PCIe standard requires the FPGA to acknowledge any request. If we reconfigure the entire FPGA, it would violate the requirement since PCIe interface is part of static logic in FPGA and it is active during PR which can make sure that FPGA can respond to PCIe commands in time [Xil16].

- Dynamically Reconfigurable Packet Processor [Xil16]

A packet processor can change its processing functions quickly by Partial Reconfiguration according to the packet types received. For example, a packet has a header which contains the partial BIT file. When the partial BIT file is processed, it can be used to configure a coprocessor in the FPGA [Xil16].

- Asymmetric Key Encryption [Xil16]

Some applications are not possible without Partial Reconfiguration. When Partial Reconfiguration and asymmetric cryptography are combined, a very secure method for protecting the FPGA configuration file can be designed [Xil16]. For example, a

host connects to a FPGA and a group of functions can be implemented within the physical package of the FPGA. The private key always in a well-protected container in the FPGA. When we implement a design, the initial BIT file is an unencrypted design and only contains the algorithm to generate interface connections between the FPGA, ICAP and host and public-private key pair. When we configure the FPGA with the initial BIT file, the FPGA generates public-private key pair. The host use the public key to encrypt a partial BIT file. The encrypted partial BIT file is downloaded to the FPGA and decrypted and sent to ICAP to partially configure the FPGA [Xil16].

1.3 Motivation and Challenges

Due to the slowing down of Moore's Law, FPGAs have become an attractive means to accelerate computing systems due to its high performance, low power, high design flexibility and low cost [TLC04, SVN04, Ver04]. For example, FPGAs are widely used in domain-specific computing to accelerate the critically intensive computation and implement different functions [CSRB11]. Some adaptive systems even require to modify the function based on the run-time requirement. To support such requirement, modern FPGAs have enabled dynamic partial reconfiguration. Partial Reconfiguration (PR) is an advanced reconfiguration feature for FPGAs and it has the capability to reconfigure specific regions of FPGAs while the other parts are still running. It provides many benefits for industry, *e.g.* changing different hardware algorithms according to different requirements and sharing the same hardware resource for different applications. It can also save reconfiguration time and resources. One of the most important and impressive advantages for partially reconfigurable systems is that it has the ability to create the fine-grained reconfigurations to maximize the utilization of the limited FPGA resources compared to the fully reconfigurable systems. Generally we partition an application into a set of fine-grained instances or tasks based on the function of the application to fulfill the partially reconfigurable designs [HW95]. Each task is regarded as a distinct configuration (module) during the running of the application. We will then download

these configurations according to different requirements of the application during runtime. Xilinx provides a floorplanning tool, PlanAhead [Xil12], to support the partial reconfiguration flow. Designer can manually use this tool to define the reconfigurable areas's position and shape. It also helps to validate the constraints. However, the tool does not offer the automatic flow for floorplanning. The designers need to manually perform the floorplan. This requires the designers to have strong expertise in floorplanning and knowledge in the module information which makes the tool difficult to use. Also the widely used approaches for VLSI flow cannot be directly adopted in the automatic flow for the placement of modern FPGAs involving partial reconfiguration. The reason is that a single block is an object in the VLSI flow and each module is an object in the module based FPGA flow. However, a module may consist of different types of blocks *e.g.* CLBs, RAMs and DSPs and these blocks not only need to be placed within a continual region to form a single entity, but also need to be placed in their valid positions which complicates the problem. Some previous studies have been proposed to address the problems [CW06, BSKB09a, YDHW05a, SB07]. However, none of them simultaneously considers different shapes (aspect ratios) and delay for each PR logic (module) during placement. Considering different shapes and delay for each PR during placement is very important for preparing an efficient initial configuration for the partial reconfiguration flow since different shapes may be suitable for different applications. Thus, an efficient CAD tool considering different shapes of modules for supporting partial reconfiguration on FPGAs is urgently needed. It can speed up the FPGA mapping flow involving partial reconfiguration. Further, it shortens the time to market of the product and reduces the cost for FPGA design.

Reconfigurable FPGAs are increasingly used in the industry because of its parallel computing characteristic, flexibility, high density and low cost compared to the ASIC and general purpose processors (GPU). However, this reprogrammable nature of FPGAs introduces new challenges in the development of the FPGA systems and it requires novel CAD tools to support the designers to implement the partially reconfigurable systems. One of the most important and challenging steps is the floorplanning of the reconfigurable areas. FPGAs involve two directions. On one hand, we consider the device reconfiguration

capability. On the other hand, we consider the heterogeneous and nonuniform distribution of the different resources of the FPGAs. These aspects introduce the new constraints for the FPGA placement. We need to find a placement with high performance (*e.g.*, high frequency) that can place the reconfigurable modules in the suitable reconfigurable regions that does not overlap with other modules [MM11, BMS11].

It is very important and challenging to prepare an efficient initial configuration as a reference for different applications and to manage the static region and PR region resources in order to avoid signal interruption incurred by PR [SBA⁺05]. Static logic means the logic will not be reconfigured during PR operation and PR logic means the logic can be reconfigured during PR operation [Xil12, Alt10]. The initial configuration solution is taken as a reference for later PR operation and different reconfigurations of each PR module can have various properties *e.g.* area, aspect ratio and delay. Hence, it is difficult to take them into consideration simultaneously during placement. On the other hand, it is challenging to efficiently place static logics and PR logics in the FPGAs.

We summarize the main shortcomings of existing studies for PR as follows.

- Traditional approaches usually rely on manual partitioning and placement, which is an error-prone and tedious process, and requires huge efforts and long development cycle.
- Existing studies do not explicitly consider different shapes and delay for each PR during placement. However, this is very important for preparing an efficient initial configuration for the partial reconfiguration flow.
- Existing studies are mostly based on traditional tile-based placement which is slow for the placement of large design on multiple FPGAs and do not consider the placement involving PR on multiple FPGAs.

This thesis tries to propose efficient solutions to address the problems mentioned above.

1.4 Research Objectives

There is a lack of efficient CAD tools for placement and routing that support partial reconfiguration on FPGAs. Traditional approaches usually rely on manual partitioning and placement, which is an error-prone and tedious process, and requires huge efforts and long development cycle due to the large design space. To address the above issues, our research attempts to fill this gap and proposes an automatic mapping flow for efficient PR. Particularly, the thesis consists of the following three pieces of work.

- 1) We propose a library-based placement and routing flow which supports the PR operation.
- 2) We propose a dynamic module partitioning approach for library based design to better guide the placement of the modules.
- 3) We propose a fast two-stage modular placement flow for interposer based multiple FPGAs.

In the first work, we propose a library-based placement and routing flow and integrate our flow into the Versatile Place and Route (VPR) [RLY⁺12].

Our main contributions of the first work 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 [CC06] 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.

In our second work, we propose a dynamic module partitioning approach for library based design which efficiently utilizes the pre-placement module information to better guide the placement of the modules. In order to adapt to different architectures and reduce the waste area between modules as a result of module size variations, we propose a set

of effective rules to select the candidate module and perform the dynamic partition of the module into several submodules to best fit the device. In addition, the approach can also improve the module based placement flexibility to address the fixed-outline constraint. Experiment results show that our approach can reduce the area by up to 10% with marginal increase in delay and acceptable runtime.

The main contributions of the second work are as follows:

- We propose a new dynamic partitioning approach to facilitate design reuse and reduce final placement area. Our approach improves the flexibility of modules for further design and try to eliminate the waste area inside and between modules.
- Our approach provides an opportunity to address the fixed-outline constraint which may not be solved in the traditional module-based design.
- Our approach can reduce the area up to 10% with marginal increase in delay and acceptable runtime.

Our third work involves placement for the interposer based multi-FPGA system. We propose a flow to address the placement for multi-FPGA system with interposer targeting delay optimization. Our approach consists of two steps. Firstly a force-directed placement is performed to obtain an efficient initial solution. Secondly, the initial solution is represented by the hierarchical B*-tree (HB*-tree) [CC06] and based on it, simulated annealing is performed to obtain the refined solution. A detailed routing model for interposer is built and integrated in the flow for addressing the interposer impact on the placement. The B*-tree representation is introduced to speed up the search and convergence of the solution [CC06]. The HB*-tree matches the multi-FPGA system well where the multi-FPGA can correspond to a B*-tree and each chip can correspond to a sub B*-tree to enable the swap of modules in different chips. In addition, we also exploited the impact of different width-height ratios of the modules for delay optimization.

Our results demonstrate that 1) Interposer impacts the placement result significantly and compared with the traditional model, our proposed model can reduce the wirelength, total delay, delay from nets crossing the interposer by 8%, 7%, 19%, respectively; 2)

The ratios of the module impact the placement results. When considering the different ratios of modules, the wirelength, total delay and delay from nets crossing interposer are improved by 8%, 4%, 9% compared to the results considering modules with only one ratio; 3) Our proposed flow can speed up the placement and achieve higher quality compared to the state-of-the-art min-cut based flow derived from [KAKS97].

The main contributions of the third work are as follows:

- We develop a wirelength estimation model involving interposer according to the commercial FPGA with interposer [HPB14, Inc13].
- We optimize timing and consider the interposer simultaneously.
- We propose a HB*-tree based modular placer for the multi-FPGA system with interposer.

1.5 Organization of the Thesis

The thesis is organized as follows.

Chapter 1 introduces the background and motivation related to our research. It presents the major research objectives of our study which is to propose an automatic mapping flow for efficient PR.

Chapter 2 describes a general mapping flow for FPGA design and gives a brief introduction to each step. It also describes the popular academic Verilog-to-Routing (VTR) flow for FPGA design that our work is based on and partial reconfiguration (PR) on FPGAs, and reviews the previous studies related to PR on FPGAs.

The main research contributions, highlighted above, are presented in Chapters 3, 4 and 5.

Chapter 3 presents our first work about the placement and routing tool which supports partial reconfiguration on FPGAs. We propose a library-based placement and routing flow to facilitate design reuse and improve the placement quality.

Chapter 4 presents our second work related to library based design and IP reuse. We

propose a dynamic module partitioning approach for the library based design flow to reduce the waste area while efficiently utilizing the pre-placement module information.

Chapter 5 presents our third work involving placement for the interposer based multi-FPGA system. We develop a wirelength estimation model for the interpose structure and propose a hierarchical B*-tree (HB*-tree) based modular placer for the placement of interposer-based multi-FPGA system.

Finally, Chapter 6 presents concluding remarks and suggestions for future work.

CHAPTER 2

Background and Literature Review

We introduce the problems, motivation and objectives in Chapter 1. Next, we introduce the basic FPGA design flow on placement and routing and provide an overview of previous studies related to FPGA placement. In this chapter, we introduce the background knowledge about Field-Programmable Gate Array (FPGA) design in section 2.1. We first describe a general mapping flow for FPGA design in section 2.1.1 and give a brief introduction to each step. Next, we describe the popular academic Verilog-to-Routing (VTR) flow for FPGA design in section 2.1.2 that our work is based on. Partial reconfiguration (PR) is briefly presented in section 2.1.3. Finally, we review the existing research work related to PR mapping flow in section 2.2.

2.1 Background

2.1.1 A General Mapping Flow for the FPGA Design

FPGA design contains several steps: design entry, register transfer level (RTL) synthesis, technology mapping, placement, routing and mapping the bitstream file to a FPGA board. Fig. 2.1 shows a general mapping flow for FPGA designs and each step is shown in the figure. We briefly introduce each step and will describe them more in the Verilog-to-Routing (VTR) mapping flow [RLY⁺12] of section 2.1.2 that our work is based on. Design entry usually contains a circuit description file. RTL synthesis performs technology logic optimization. The technology mapping maps the logic gates of the optimized design to the look-up tables (LUTs) and flip-flops(FFs) [RLY⁺12] of the FPGAs. Placement is

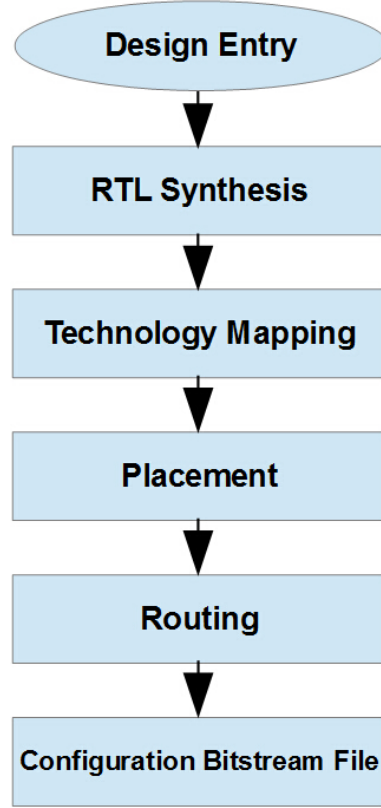


Figure 2.1: A general mapping flow for FPGAs.

used to place the logic in the FPGA and find a location for each block. Routing is used to connect different blocks according to the circuit netlist to implement the functionality of the circuit. The last step is to generate the bitstream file and map the bitstream file to the FPGA board [CJA08, CCP06].

2.1.2 Verilog-to-Routing (VTR) Flow for the FPGA Design

Our work is based on the Verilog-to-Routing (VTR) [LGW⁺14] flow and we describe it below. VTR is a collection of tools that performs the full FPGA CAD flow from verilog to routing. The stages of the flow are shown in Fig. 2.2 [RLY⁺12]. The flow mainly consists of ODIN II (synthesis), ABC (optimization and technology mapping), and VPR (pack, place and route) [LGW⁺14]. Odin II [LGW⁺14] makes the Hardware Description Language (HDL) design be a flattened netlist. Next, the ABC [LGW⁺14] implements logic optimization and maps the logic gates into LUTs and FFs. A .blif format netlist

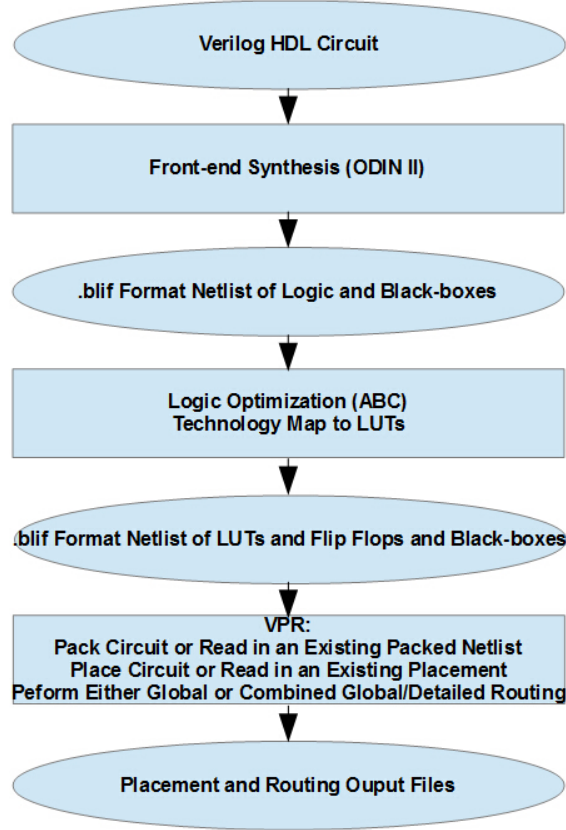


Figure 2.2: Verilog-to-Routing (VTR) flow for FPGAs.

generated contains LUTs, FFs and blackboxes. VPR [LGW⁺14] packs the netlist through clustering approach and places the logic blocks into the FPGA. Finally, VPR routes the nets for the circuit [CJA08, CCP06]. More detailed description about the placement and routing is described below.

FPGA placement and routing

The placement stage requires a netlist consisting of a set of logic elements and their connections. Since the netlist is created in the synthesis step [RLY⁺12] which involves technology mapping, we can directly map these logic elements to the logic resources on the FPGA. Given a netlist consisting of logic elements and their connections, FPGA placement can be modeled as follows. FPGA placement places all the logic elements on the available logic elements sites of FPGAs and meet the timing constraints. It also makes

sure that all the nets including their connections can be routed successfully using the available routing fabric of FPGAs. This is similar to the ASIC placement problem [Jay01].

FPGA routing problem can be defined as follows. We use a routing-resource graph [KTR08, BR97] to represent the FPGA. The nodes of the graph are the wires and the logic block pins. The edges are the switches. We need to use the routing resources to finish netlist connections and optimize the timing.

2.1.3 Partial Reconfiguration

In this subsection, we provide an introduction to partial reconfiguration of FPGAs. Partial Reconfiguration (PR) is an advanced reconfiguration feature for FPGAs and it has the capability to reconfigure specific regions of FPGAs while the other parts are still active or are inactive in a shutdown mode after its initial configuration. Not all FPGAs support dynamic partial reconfiguration. Some of the Xilinx FPGA devices (i.e. Virtex series) on the market have the ability to allow dynamic partial reconfiguration.

Two basic types of dynamic reconfiguration on a single FPGA are mentioned by the Xilinx Inc [Xil12]. One is the difference-based partial reconfiguration and the other is module-based partial reconfiguration [BMS11].

When only a small change is needed in the design, we can use the difference-based partial reconfiguration. For instance, if we only need to change Look-Up Table (LUT) equations or dedicated memory blocks content, it is better to use the difference-based partial reconfiguration. The partial reconfiguration files (bitstreams) contain only the information about the differences among the new content of the FPGA and the current content that resides in the FPGA. In the difference-based partial reconfiguration mode, since the partial bitstreams only contain the difference information, it can be extremely smaller than the full reconfiguration files (bitstreams) which makes the switching activity that the configuration is changed from one implementation to another very fast.

Module-based partial reconfiguration is used to reconfigure large blocks of logic by using modular design concepts. We define each distinct portion of the design that can be

reconfigured as a reconfigurable module. Each reconfigurable module has its own specific properties and specific layout criterion which must be met (i.e. timing constraint). If we want to use the partial reconfiguration, we must make sure that the reconfigurable module meets all the requirement of the designs.

The dynamic partial reconfiguration can support two types of applications. On the one hand, it supports the temporal locality of applications. On the other hand, it supports the functional locality of the applications [BMS11].

If there are idle or underutilized operations within a reconfigurable application, we can take this case as the temporal locality of an application. Sometimes some operations are not needed at a specific time or they are not needed for computation immediately, these operations will remain idle. For instance, there are data dependencies in an algorithm. It will make the operation take some time to wait for results of the other operations. There can also be an application-specific operation that was not used frequently in the computation. Dynamic reconfiguration is an impressive way and a good choice to remove such idle operations from the systems. It helps to replace them with other more useful operations during the runtime. It offers a chance to save the FPGA resources that we dynamically change hardware function according to different requirements compared to the possible static systems. Thus, if a design is too big to be placed in the hardware resources on the FPGAs, we can use dynamic reconfiguration. In other words, it provides a new design methodology for producing large designs. Since idle operations within a application do not consume any valuable resources, the application has the chance to consume less resources than possible in a static reconfigurable system. Fundamentally, it means that we can use the feature of partial reconfiguration of FPGAs to dynamically adjust the configuration of reconfigurable area to match the requirements of the temporal locality of the application. Dynamic partial reconfigurability offers the ability to maximize the utilization of the limited FPGA resources. With the increasing of capacity of the individual FPGAs, we have more chance to deploy the dynamic reconfiguration. We also may integrate the complex systems into a single FPGA more easily since the larger dynamic partially reconfigurable FPGAs become available [BMS11, ZN00, Wir97].

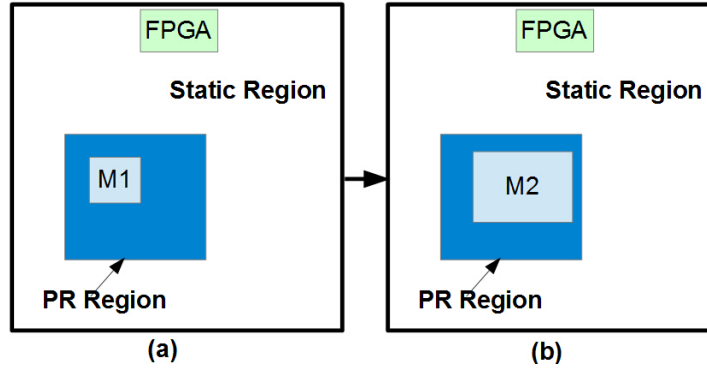


Figure 2.3: An example for Partial Reconfiguration.

The application-specific computing nature of an application is regarded as the functional locality of operations [BMS11, ZN00]. In other words, the functional locality of operations means the application uses the operations specifically within an architecture. In a static system, if a large application cannot fit within a finite resource of the system, we need to partition the application and schedule them on static resources. However, if we want to execute different partition within a sequential execution schedule in a single static architecture, the architecture must be general-purpose enough to provide the ability to compute all the functions of the application. The amount of specialization is restricted by the reuse of hardware for various partitions which results in less efficient structures for supporting functional locality of operations. However, dynamic reconfiguration can be used well for large special-purpose systems that need to be partitioned. Compared to the static system that generally supports all computation variations within an application, the specialized operations of the application are reconfigured in the dynamic system during runtime. Hardware resources can be tailored to be executed in the dynamic systems for the application by using its flexible nature which makes the architecture more powerful than the static architecture [BMS11, ZN00].

Fig. 2.3 shows an example for the partial reconfiguration on a FPGA. We assume that there is one PR region with the blue color and the left part is the static region with the white color. M1 and M2 are two logic modules. At the beginning, the M1 is placed in the PR region shown in Fig. 2.3(a). We can replace logic module M1 with logic module M2 in the PR region later shown in Fig. 2.3(b).

2.2 Literature Review

This section describes previous research work related to PR mapping flow on FPGAs. We first give an overview of previous work related to FPGA placement. Tile, cluster and module based placement are different strategies. Floorplanning and placement are two different problems in FPGA design. In the Floorplanning problem, each module only contains resource requirements for different types of blocks and each module does not have the ratio and size. The aim of Floorplanning is to find a region for each module such that the region has enough resources for the module. In other words, Floorplanning determines the shape and area of a set of blocks. But each block's position is unknown and is determined in the placement step. Partition is an approach that can be used in the placement. 2D, 2.5D and 3D with interposer are different types of architectures. In the thesis, we propose a library-based placement and routing flow which belongs to module based placement. Each module has been pre-placed, which means that each module has a size (width and height) and each block inside the module has its relative position which is different from the Floorplanning problem. Firstly, we classify the placement according to different strategies.

2.2.1 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) [RLY⁺12] gave a complete flow from hardware description language (HDL) to physical mapping on FPGAs of various hardware architectures, where Versatile Place and Route (VPR) was used inside to perform the tile-based placement and routing. Later work extended the fine-grained placement to heterogeneous FPGAs. For example, [JGS13] introduced the genetic algorithm for solving the heterogeneous FPGA placement. [SRRK04] proposed a multilevel multi-resource partitioning algorithm for heterogeneous FPGA placement. [Hu06] employed a multi-layer density system for the heterogeneous FPGA placement. [CCC14, GA12] proposed an analytical placer for the heterogeneous FPGAs. However, the tile-based placement faces the challenges of a long running time for large-scale designs and does not support PR.

2.2.2 Clustering based FPGA Placement

Clustering based placement is proposed to cluster the fine-grained tiles and speed up the large-scale FPGA placement design through hierarchical placement. Clustering algorithm based on connectivity of logic blocks has been proposed to reduce the size of the large-scale placement problem [HMS04]. Clusters could also be obtained by partition. A multilevel multi-resource partitioning algorithm has been proposed that guarantees to produce solutions which balance the utilization of different resources across the partitions [SRRK04]. There were different clustering strategies adopted. Clustering algorithm could perform iterative local refinement on fine-grained clusters [YVC14]. Some clustering algorithms considered the level move that a Basic Logic Element (BLE) node in a cluster could move to another cluster or swap with a BLE node in another cluster [CC04]. In [CW04], clustering algorithm adopted irreducible realization lists to store the possible positions for each cluster. They determined the cluster position based on the wirelength. However, there is no module defined for functions or sub-circuits in the cluster based placement, and hence the flow does not support module reuse.

2.2.3 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. The work in [CW06] proposed a module based floorplanning algorithm for heterogeneous resources of FPGAs. They adopted slicing trees, a representation method for IC design, to represent a FPGA floorplan and integrated it into a simulated annealing algorithm to search the optimal solution. Slicing tree has a strict requirement that a floorplan must can be recursively partitioned into two parts along the horizontal or vertical line until each part has only one module. However, this constraint may result in large waste area and most of floorplans are non-slicing structures which cannot be presented by the slicing trees. [BSKB09b] also used slicing trees to represent the floorplan. It had the same problem that we mentioned above. [YDHW05b] proposed a "less flexibility first" algorithm for heterogeneous FPGA floorplanning problem and each module contained some resource requirements. The algorithm firstly found a region for

the module with the least flexibility and found regions for other modules. However, the algorithm used in this study was a deterministic algorithm which can be easily trapped in a local optimal solution. [LCY11] adopted a non-sling floorplanning representation namely Corner Block List (CBL) for heterogeneous FPGAs. They firstly used min-cost-max-flow algorithm to assign the positions to CLBs. Next, they assigned positions to RAMs and DSPs. However, they did not consider modules with different ratios. Our work supported modules with different ratios and adopted B*-tree representation which has faster area estimation and less solution space.

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 [Xil12, HLM⁺12, Inc11] 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 the large design space, various studies proposed automatic floorplanning for PR modules which aims to ease the placement effort related to PR. During placement, we need to place static modules and PR modules. If we know the possible regions for the PR modules, it may help to place the static modules. In earlier study [BKS00], each PR module was modeled as a block which has the width and height. Each module did not have resource requirements for different types of blocks. They adopted the simulated annealing algorithm to place the modules. However, this assumption did not consider placing CLBs, RAMs and DSPs and was difficult to apply in practical applications. Later studies like [YGR10, BKT13] developed automatic flow for PR floorplanning based on Xilinx process and special bus was needed to connect modules and support run-time reconfiguration. In [RDM⁺17], they provided a floorplanning automation framework for placing logics, integrated in the Xilinx tool chain, which was based on an explicit enumeration of the possible placements of each region. [CPF09] 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. [BSSK11] extended

the floorplanning algorithm in [CW06] to consider the PR floorplanning in heterogeneous resources. In that study, a global floorplan generation approach was introduced to obtain shared positions for common modules across sub-task instances. [SB06] 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. [VF11] proposed an efficient mapping method which performed the floorplanning optimization of reconfigurable modules from the high-level estimation. There was also existing work that performs simultaneously floorplanning and placement. [MSSM10] 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. However, they did not consider modules which may consist of CLBs, RAMs and DSPs and support modules with different ratios. Furthermore, they did not support module reuse. Compared with these studies, our work supported PR and 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 and delay optimization.

Some of the previous studies focused on module based placement. The work in [BKL10] initially found a region for each module and expanded the right boundary of the region until it met the resource requirement. The work in [HLM⁺12] found a partial reconfiguration region for each partial reconfiguration module and the height or width of each region were expanded until they met the resource requirement. However, all the previous studies were not fit for different architectures and did not consider waste (empty/blank) area which existed inside and between modules. This impacted the final design significantly.

In the thesis, we explore the automatic workflow mapping in the interposer based multi-FPGA system (2.5D FPGA). Thus, we also classify the placement according to the different FPGA architectures.

2.2.4 Modular Placement for 2D Single FPGA

There was a large amount of tile-based placement for a single 2D FPGA. They could be based on heuristic approach [RLY⁺12] or analytical approach [CL10, ZML⁺07]. However, as we discussed above, tile-based placement faced great challenge for placement of large-scale design. Thus module based placement and hierarchical placement have been proposed to address the challenge as well as support partial reconfiguration [AGBD07, BSSK11]. However, placement in multiple FPGAs is fundamentally different from placement in a single FPGA due to the impact of inter-chip delay and assignment of modules into different chips.

2.2.5 Placement for 2D Multi-chip and 3D FPGA

Traditional multiple FPGAs had a limited number of IOs for passing signals between different dies. Hence, cut-size optimization was the common goal to reduce the number of signals crossing chips as well as the long inter-chip delay. [SKT92] proposed an algorithm to minimize net crossings for addressing system partitioning and considered the block assignment of multi-chip modules during the high level design of an application-specific IC. [KSY95] developed a new performance driven partitioning algorithm to implement a large circuit in multiple FPGAs to minimize the cut size. [WK93] developed a method named MP2 to minimize the total number of terminals of all blocks during network partitioning. [RNS95] presented a rectilinear partitioning algorithm to minimize the cut sizes of the partitions and handle timing specifications for traditional multiple FPGAs.

Recently, 3D chip emerged for achieving higher integration density and better performance. The 3D FPGA has multiple dies stacked vertically which utilized through-silicon vias (TSVs) as vertical connections [LMW⁺13]. The TSV density was closely related to the cost and hence, most of 3D FPGA placements target the minimization of TSVs which was similar to the cut-size optimization. [AMB06] adopted the popular program package namely hMETIS derived from [KAKS97] to divide the circuit into several parts and minimize the connections between dies. [SS11] adopted a TSV-aware partitioning algorithm for 3D FPGA to achieve higher performance for applications. [LMW⁺13]

proposed a reconvergence-aware layer partition (RALP) algorithm for 3D FPGA design targeting performance optimization. However, compared to our work, they were all tile-based flow rather than module-based flow and targeted different routing models since they did not consider the interposer which is an electric interface routing for connecting different components [HPB14].

2.2.6 Placement for 2.5D IC and FPGA with Interposer

Some of the studies have addressed the interposer-based FPGA placement, although there were some studies proposed for ASIC floorplanning using the 2.5D integration with interposer. In [HC13], the authors proposed the first work of chip-interposer codesign to place multiple chips on an interposer-based IC to reduce the inter-chip wirelength. They also proposed a hierarchical B*-tree to simultaneously place multiple chips and IO buffers. The study in [LCW14] presented an enumeration-based algorithm and a network-flow-based algorithm to solve the multi-die floorplanning and signal assignment problem for 2.5D IC respectively. [SDM15] adopted simulated annealing for die placement and I/O assignment in 2.5D IC. However, all the approaches have been focusing on the ASIC flow instead of the FPGA flow. Since FPGA had a different architecture and has position constraints for different types of blocks such as CLBs, RAMs and DSPs, placement methods targeting FPGA need to be developed and the approaches used in specific IC design were not directly applicable to the FPGA architecture.

The study in [HPB14] was the first work to consider the tile based placement in the multi-FPGA system with interposer. In comparison, our work employed a much more detailed interposer routing model and mainly focused on the development of module based placement flow for delay optimization. Our work also supported modules with different ratios during placement.

2.2.7 PR-aware Routing

The routing interface design is a major step for supporting the PR operation and is 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 [SR12, CCBH07]. 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. [CCBH07] used a bus macro generated based on Xilinx Design Language (XDL) for connecting the static and reconfigurable parts. [KBT08] proposed a tool called ‘ReCoBus-Builder’ to enable the communication between the static and reconfigurable modules through a fixed bus infrastructure or dedicated point-to-point links with other parts of the system. [ABD⁺07] 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 [Xi11] 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. [KBT10] 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. [SR12] 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.

2.3 Summary

In this chapter, we described the basic concepts about FPGA mapping flow and the popular VTR mapping flow that our work is based on. We briefly introduced placement and routing which are two indispensable steps for partial reconfiguration on FPGAs. We also presented the basic concept about partial reconfiguration on FPGAs. Finally we

gave a comprehensive literature review related to the mapping flow related to the partial reconfiguration on FPGAs.

CHAPTER 3

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

There is a lack of efficient computer-aided design (CAD) tools of placement and routing which support the partial reconfiguration on Field-Programmable Gate Arrays (FPGAs). In this chapter, we propose a library-based placement and routing flow which supports the Partial Reconfiguration (PR) operation and best utilizes the pre-placed and routed modules from the library to significantly save the execution time while considering area-delay products of each module with different ratios and optimizing area and delay of the final design.

3.1 Introduction

With the technology advancement, both semiconductor industry and design community face the great challenge that the design effort is continuously increasing [CFIGvRv06]. The rapidly growing design complexity and short time-to-market pressure make the problem even worse [CFIGvRv06]. 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 project to reduce the design, verification cycle and risk [CFIGvRv06, Xiu07]. It also reduces design cost since the reused components have been prepared and verified [CFIGvRv06, Xiu07]. Library based design has been studied in previous work. The work in [WL10] presents a floating-point library to support floating-point computation in general formats and enable a higher level of

parallelism. In [SSAA03], they reuse hardware components of the library in FPGA design for improving the verification of the digital circuits implemented on FPGA. In [LPL⁺11], 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 [Gaj99]. IP modules are used for improving productivity in [ZPJU05]. In [HGS05], 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.

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 find 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 [SR99, GA14]. Hierarchical or modular floorplanning is hence proposed to speed up the process [AGBD07, SJA09]. 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 whose 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 [Xil12, Alt10]. 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 chapter in order to take the advantage of design reuse and efficiently support PR in FPGA, we propose a library-based flow to improve the efficiency of placement and routing and support PR. 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 proposed 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 [Xil11, CPF09, ABD⁺07].

The main differences between our work and previous studies are listed below.

- We introduce B*-tree [CC06] for representing the placement. However, we modify the B*-tree operations described in 3.4.3 to support the different ratios of each module in order to support the multiple contexts of modules during placement.
- B*-tree is used in the ASIC placement and each node of a tree is a single block. However, we consider module based placement in FPGA and module may consist of different types of blocks, e.g., CLB, RAM and DSP. Each node of a tree is a module. We not only need to place each module in the FPGA, but also need to place each type of blocks in their valid positions which increases the complexity.

The proposed flow has been integrated into the Versatile Place and Route (VPR) [RLY⁺12] to substitute the original placer and router. The experiment results demonstrate improvement on execution time with acceptable area and delay overhead compared to the results from the tile-based VPR tool. 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 [CC06] to enable a fast modular placement on both fine-grained and coarse-grained fabric considering different module ratios.
- We propose the detailed pin-to-wire routing interface to support the PR-aware routing.

The rest of this chapter is organized as follows. Section 3.2 introduces the overview of

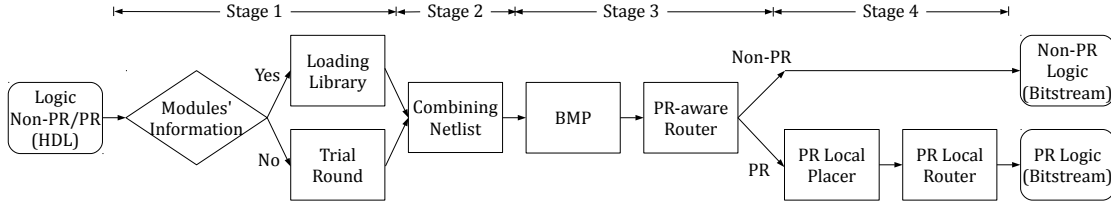


Figure 3.1: Our proposed mapping flow for library based design.

our proposed placement and routing flow. Section 3.3 discusses the methodology for module library construction prepared for the later placement and routing. The detailed placement steps are introduced in Section 3.4, and Section 3.5 describes the PR-aware routing interface design. Section 3.6 describes the local placement and local routing for multiple contexts in the PR region. Section 3.7 discusses the experimental results and parameter optimizations through design space exploration. Finally, we conclude with our remarks in Section 3.8.

3.2 Library-Based Mapping Flow

The mapping flow we propose to solve the library-based placement is shown in Fig. 3.1. 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 get area-delay products for the module. Here k represents the number of ratios for the module. 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

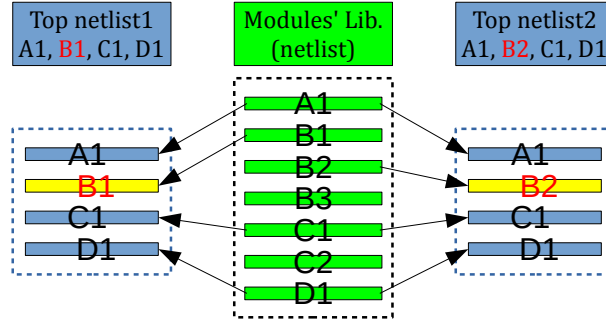


Figure 3.2: Library based structure.

proposed to place the static and reconfigurable modules simultaneously in heterogeneous FPGA resources. BMP introduces B*-tree representation [CC06] 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 [CC06]. 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 also the aspect ratio (width/height) is considered to achieve the area and delay 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. 3.2 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.

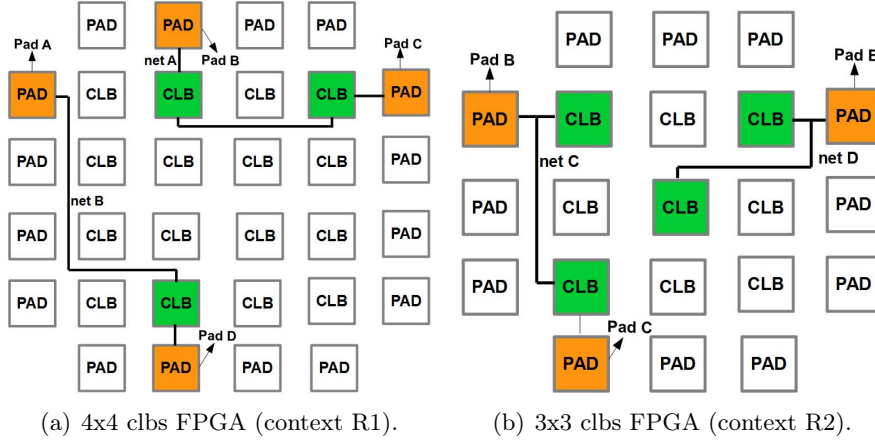


Figure 3.3: Two contexts (R1 and R2) of a reconfigurable module

3.3 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 [RLY⁺12] 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.

3.3.1 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. 3.3 shows the two contexts of a reconfigurable module. Since

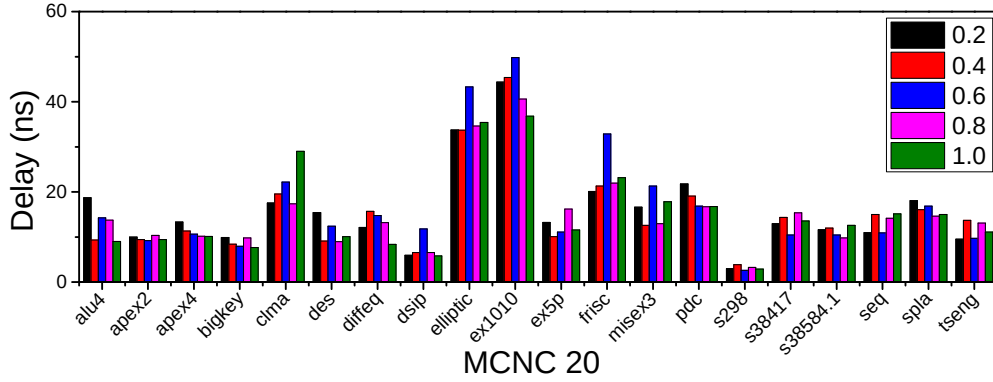


Figure 3.4: Relation between shapes and delay for MCNC20 benchmark.

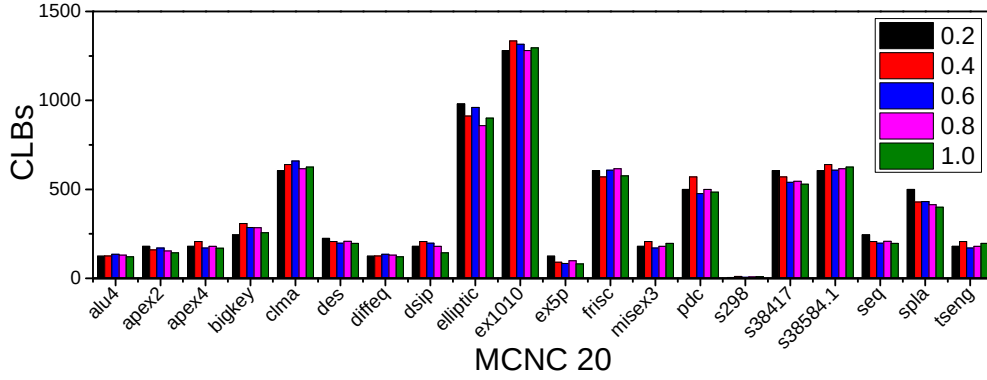


Figure 3.5: Relation between shapes and area for MCNC20 benchmark.

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 netC, but context R1 has a unique IO net of net B and context R2 has a unique IO net of netD. In order to enable the context switch between R1 and R2, all the required IO nets netA, netB, netC and netD 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, PadD, PadE}. Then the local placement and routing of the contexts inside the module are based on the pin arrangement of the module.

3.3.2 Module Ratio Selection

We observe that the shape of a module shows impact on delay and area. Fig. 3.4 and Fig. 3.5 demonstrate simulation results of delay and area of MCNC benchmarks [Min07]

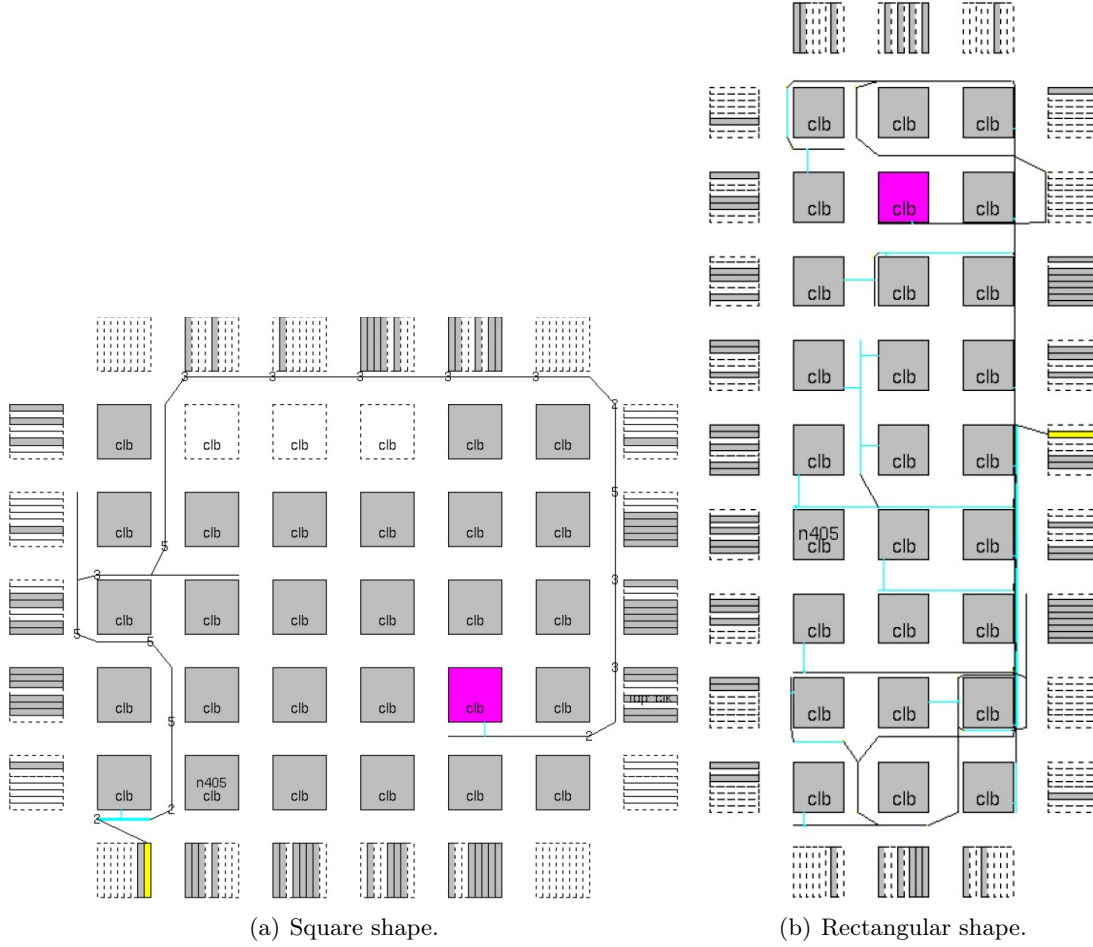


Figure 3.6: Various shapes and delay in two modules with the same logic.

for various aspect ratios, respectively. Legends shown in the figures are aspect ratios, width over height, of the shapes. Note that here we assume the basic reconfigurable unit is one CLB without loss 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. 3.6(a) and Fig. 3.6(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

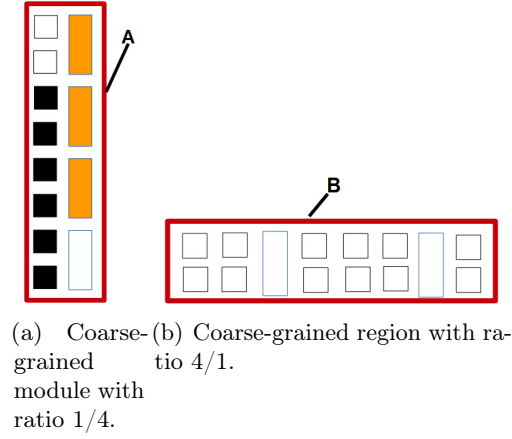


Figure 3.7: Coarse-grained module with different ratios

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. 3.7 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. The reason for choosing this parameter is that a design with small area usually has small delay. Thus, τ is suitable to estimate the module. Shapes of each module are sorted in the ascending order of τ in a list for reference in the later placement stage.

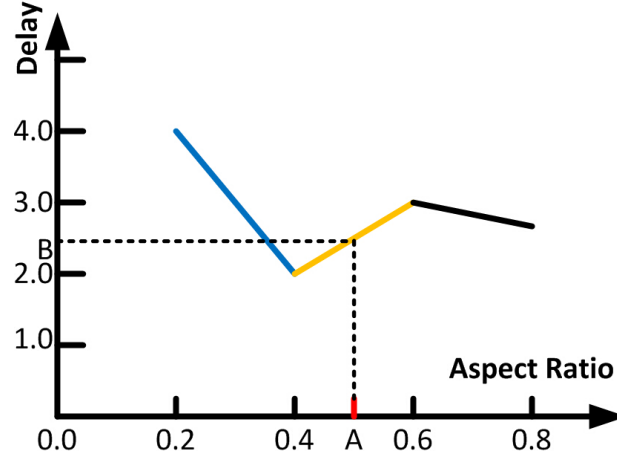


Figure 3.8: Module delay estimation for a specific ratio.

3.3.3 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 $k \tau$ 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 accurate 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. 3.8. 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 based on the available data points 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 based on the current available data points. Some previous studies [NHCB02, MSCL07, HWY⁺09] report that their delay estimation approaches for FPGA have approximate error rate of 13% which shows our approach is feasible.

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

3.4.1 Problem Formulation

The problem of library-based placement can be formulated as the follows. Given a set of n rectangular modules $B = \{b_1, b_2, \dots, b_n\}$ 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. Each module in the library has been pre-placed means that the blocks including CLBs, RAMs and DSPs inside each module have been already placed and these blocks have their unique positions. 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 each block inside the module has a valid unique position based on the pre-placed placement information of each module, and the bottom-left corner

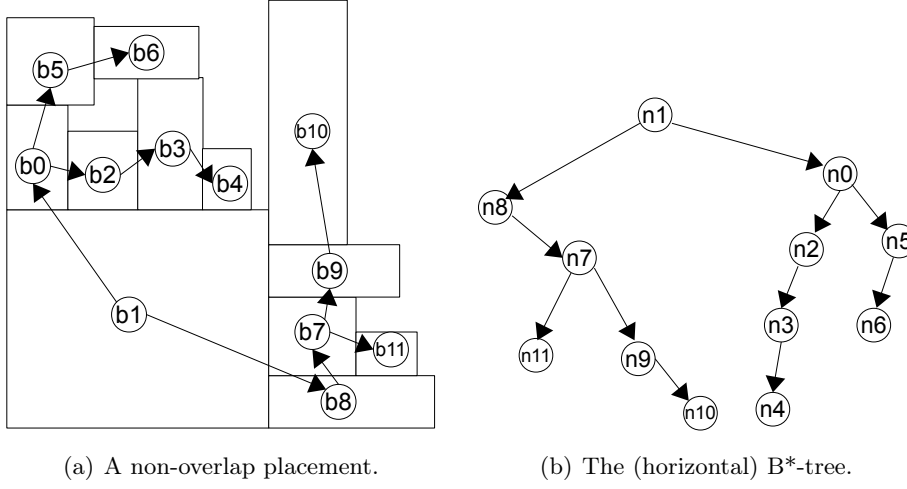


Figure 3.9: A placement and corresponding B*-tree representation.

coordinate of the module b_i is assigned to (x_i, y_i) . The objective is to optimize the area utilization, wirelength and delay.

3.4.2 B*-tree Representation

In order to enable a fast search of optimal placement in the solution space, we modify the B*-Tree to represent the module placement. We describe the main difference between B*-tree used in ASIC design and B*-tree used in module based FPGA design below. B*-tree is used in the ASIC placement and each node of a tree is a single block. However, we consider module based placement in FPGA and module may consist of different types of blocks, e.g., CLB, RAM and DSP which imposes the position constraint on B*-tree operations described in 3.4.3. Each node of a tree is a module. We not only need to place each module in the FPGA, but also need to place each type of blocks in their valid positions which increases the complexity. We also modify the B*-tree to consider the different ratios of modules during placement. Here we introduce the background for the B*-Tree representation [CC06]. B*-tree is an ordered binary tree structure proposed in modern floorplanning of ASIC designs [CC06]. 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 $O(n!2^{2n}/n^{1.5})$ [CC06], 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 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 Fig. 3.9. Take the arrow from module b_9 to module b_{10} for example, the node n_9 and node n_{10} are corresponding nodes respectively in the B*-tree. On the right side of the module b_9 , there is not any module. Thus, the node n_9 does not have a left child. Module b_{10} is the lowest unvisited module above and adjacent to module n_9 . Thus, the corresponding node n_{10} is the right child of node n_9 .

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

- OP1: Load a module with shape of $1/(\text{aspect ratio})$.

It is similar to rotate node/block by 90 degrees in the B*-Tree as shown in Fig. 3.10 (a). Theoretically, the delay and the area of individual module with aspect ratio and $(\text{aspect ratio})^{-1}$ are the same. However, it affects the total placement results.

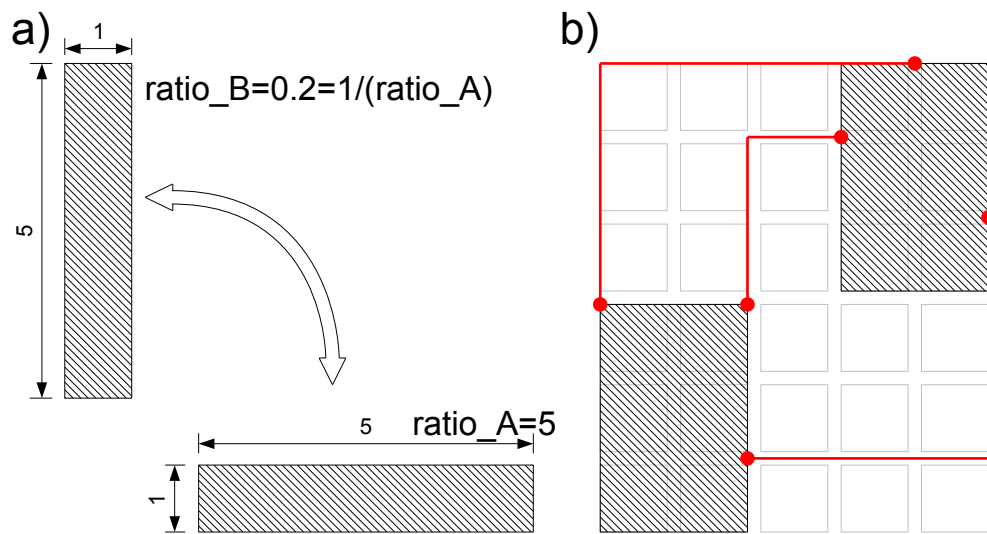


Figure 3.10: Module rotation and wirelength estimation. (a) Rotation of a module. (b) Red line shows wirelength estimation from two modules.

- OP2: Move a module to another place.

It deletes a node and inserts it into another place in the B*-tree.

- OP3: Swap two modules.

It swaps two nodes in the B*-Tree.

- OP4: Load 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 during placement. It is impractical and impossible to get all the possible ratios of modules. Currently we choose some ratios for our work and the reason is described in the last part of subsection 3.7.1.

When placing the modules with heterogeneous resources, position constraints need to be imposed on the operations which. 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.



(a) Coarse-grained module placement generated by BMP. (b) Coarse-grained module placement after mapping to the FPGA architecture.

Figure 3.11: Coarse-grained module placement results

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 [Inc11, BKT13, XZL14]. Fig. 3.11 illustrates an example for the placement of heterogeneous modules. Here each module represents a static region or reconfigurable region. Fig. 3.11 (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. Fig. 3.11 (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 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

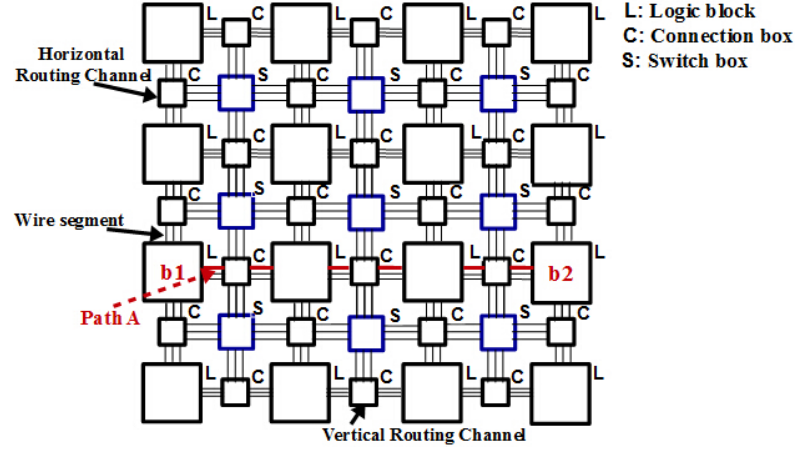
the experiments.

$$Cost_{system} = \alpha \frac{A}{A^*} + (1 - \alpha) \left(\beta \frac{D_T}{D_T^*} + (1 - \beta) \frac{D_M}{D_M^*} \right) \quad (3.1)$$

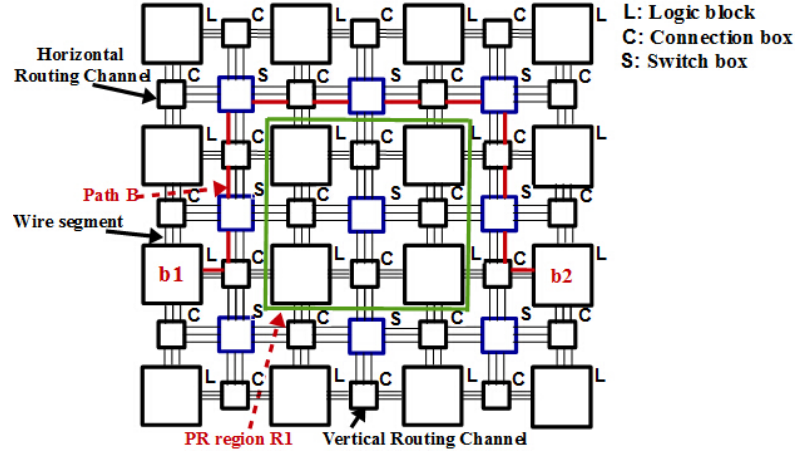
After each operation, we map the placement result generated by BMP to the FPGA architecture and evaluate its quality. The cost function [WW91] is calculated as shown in Eq. 3.1. Here A and A^* represent the current total and average area, respectively. In order to get A^* , we randomly run the flow for many times and calculate the average value of the total area. We currently run 1000 times for getting the average value of area and delay and the number of times can be changed according to different requirements. 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 (module) delay. Both parameters are explored in Section 3.7. 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 [KM00] as shown in Fig. 3.10 (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.

3.5 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 Fig. 3.12. Fig. 3.12 (a) shows that the path A from b1 to b2 connects directly



(a) Normal routing in VPR.



(b) PR-aware routing.

Figure 3.12: Normal routing in VPR and PR-aware routing.

as the red path shown. However, path B cannot go through the PR region R1 (green bounding box) shown in Fig. 3.12(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. 3.13 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)

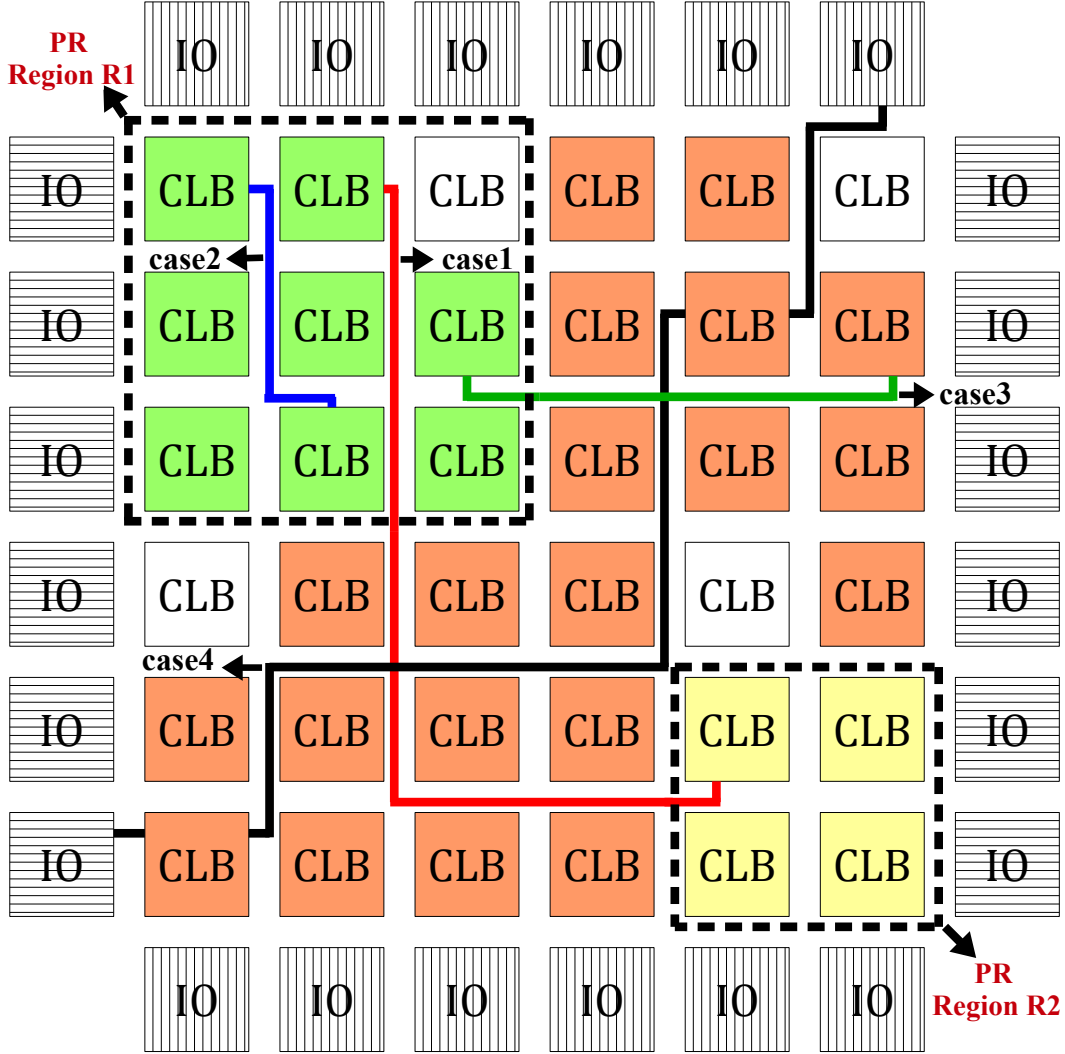


Figure 3.13: Four types of connections of nets.

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.
- 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 [SR12]. We describe the detailed pin-to-wire interface design in the next section.

3.5.1 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 [SR12]. 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 Fig 3.14. 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



Figure 3.14: Defining interface according to different types of nets.

A and track B1 (blue line) in the region B.

- 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 for 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.

3.6 Local Placement and Routing in PR Region

As we have discussed in Section 3.3, 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. 3.15 shows an example for

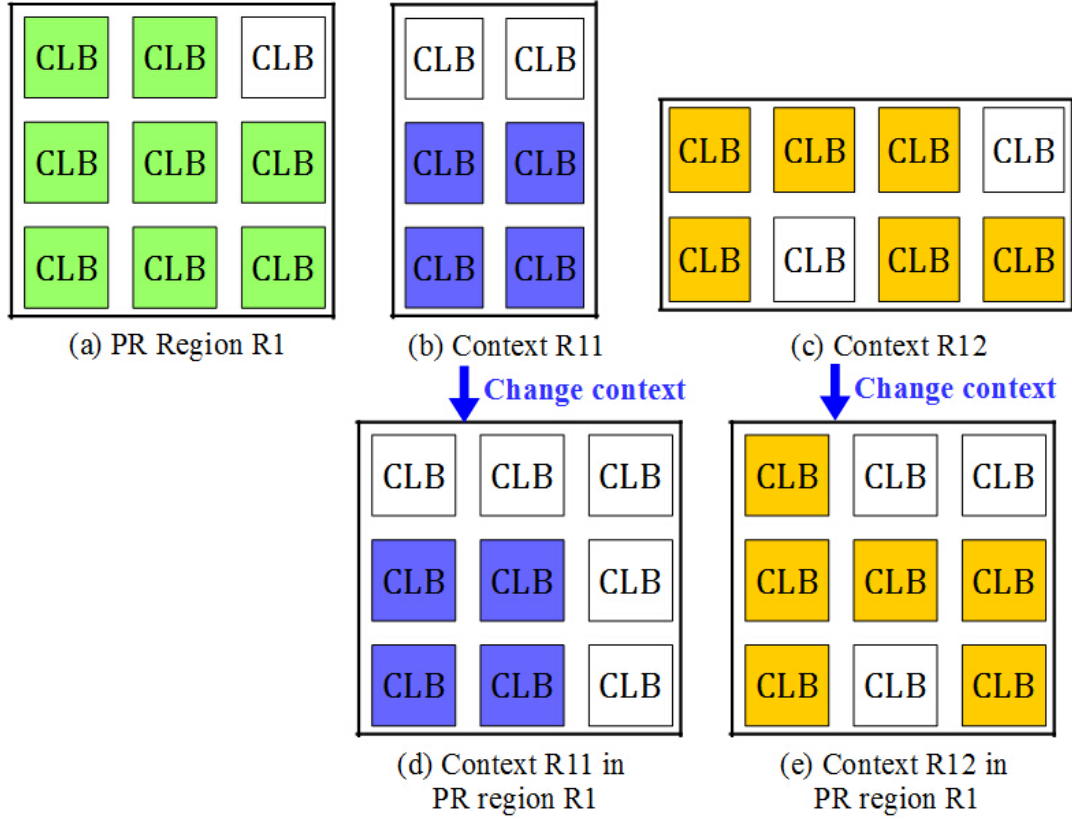


Figure 3.15: Local placement in a PR region.

the local placement in a PR region. We use the PR region R1 from Fig. 3.13 as shown in Fig. 3.15(a). Fig. 3.15 (b) and (c) are two contexts: R11 and R12 which need to be loaded in the PR region R1. Fig. 3.15 (d) shows that the context R11 can be placed in the PR region R1 directly, and Fig. 3.15 (e) shows that the context R12 cannot 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 3.3. 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

Table 3.1: Architectures and Parameters 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

the VLSI and FPGA mapping flow that when the routing failed, the flow can reroute (*i.e.*, re-placement and re-routing). More specifically, we first choose the option 1. Only when option 1 is guaranteed to be not feasible, we choose the option 2.

3.7 Experimental Results and Discussion

Currently, there are no publicly available benchmarks for evaluating the module-based flow. Hence, we derived the cases based on the widely used VTR benchmarks. In the library-based mapping flow, each block is a module which contains some logic (CLB, DSP, etc.) and each module can be regarded as a function module. If the cases in the VTR benchmark suite have the hierarchical structure that could be divided into several modules, we will reconstruct them into the module-based benchmarks.

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 placer BMP. 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 3.1. BLE represents

Table 3.2: 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
mkSMAadapter4B	4	393	0	5	mkDelayWorker32B	8	1117	0	41
LU8PEEng	4	5102	8	45					

the basic logic element and the BLE column represents the maximum number of BLEs in each CLB. 'RAM StartCol' represents the start column in architecture for placing RAMs and 'RAM Gap' represents the distance between two adjacent RAM columns. 'RAM Height' represents the height of each RAM block. For the DSP, the parameters have the similar meanings.

3.7.1 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 3.2. The circuits in VPR range in size from 220 to 99700 6-LUTs and the detailed statistic information containing the number of primary inputs and outputs, the number of flip-flops and the maximum data width of all the logical memories is described in [RLY⁺12]. The number of modules ranges from 4 to 32 for current case set. We also 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

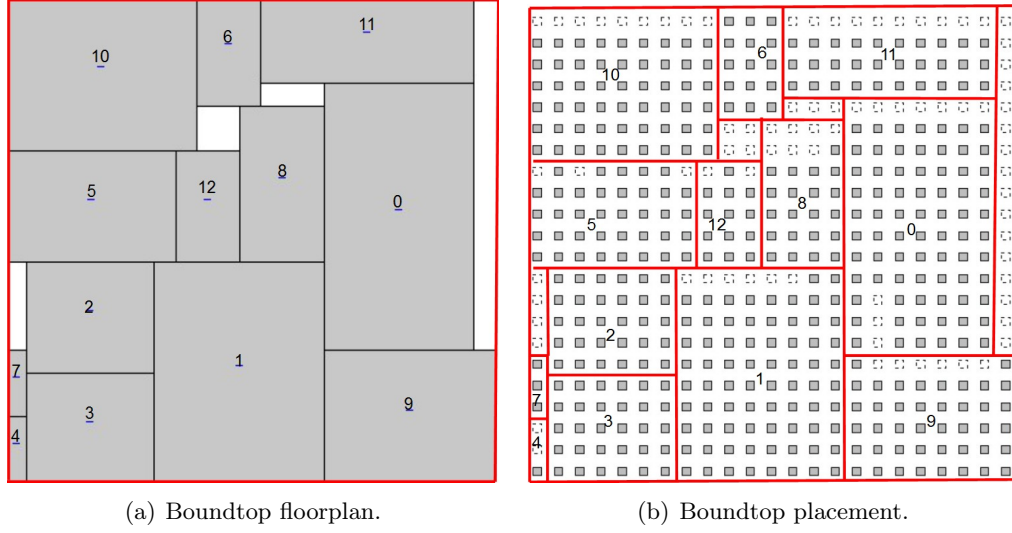


Figure 3.16: Floorplan and placement results for the case boundtop.

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.

3.7.2 Library-based Placement Results and Discussion

First we demonstrate a placement result of benchmark *Boundtop* in Fig. 3.16(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. 3.16(b) shows the placement results on VPR. The red area are the corresponding mapping area of each module showed in Fig. 3.16(a). The proposed placement also supports heterogeneous

Table 3.3: Resource Utilization for all the Cases in the Final Design

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%	mkSMAAdapter4B	88%	0	8%
raygentop	79%	22%	1%	bgm	83%	0	0
boundtop	81%	0	0	stereovision2	89%	0	0
stereovision1	89%	0	0				

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 in the final design are listed in Table 3.3. CLBU, DSPU and RAMU represent the utilization of CLB, DSP and RAM respectively. The utilization of each type of blocks is the value of the number of used blocks of this type divided by the total number of blocks of this type. The experiments show that our flow 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.

Area

Fig. 3.17 shows the area comparison of each benchmark and average area comparison of all the cases. The average value of the area is the value of the total area of all the cases divided by the total number of 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. 3.16. 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

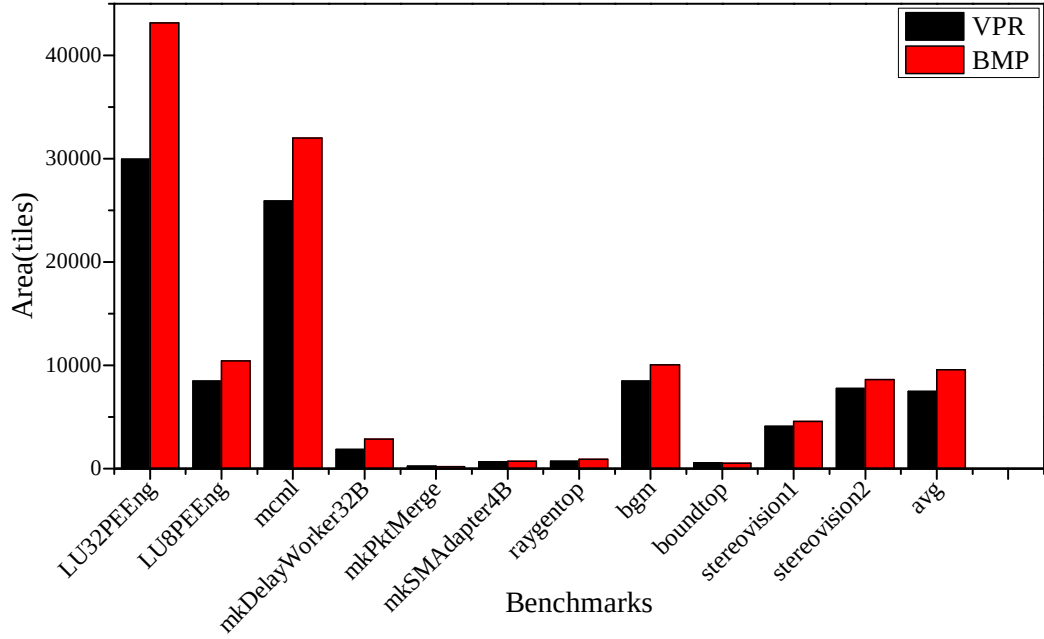


Figure 3.17: Area comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.

cost. As the boundary (red) line of the bounding box showed in Fig. 3.16, 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.

Delay

We demonstrate the delay comparison in Fig. 3.18. The black (left, VPR) bars and red (right, BMP) bars show that the delay of the critical path of each benchmark 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 delay from the VPR

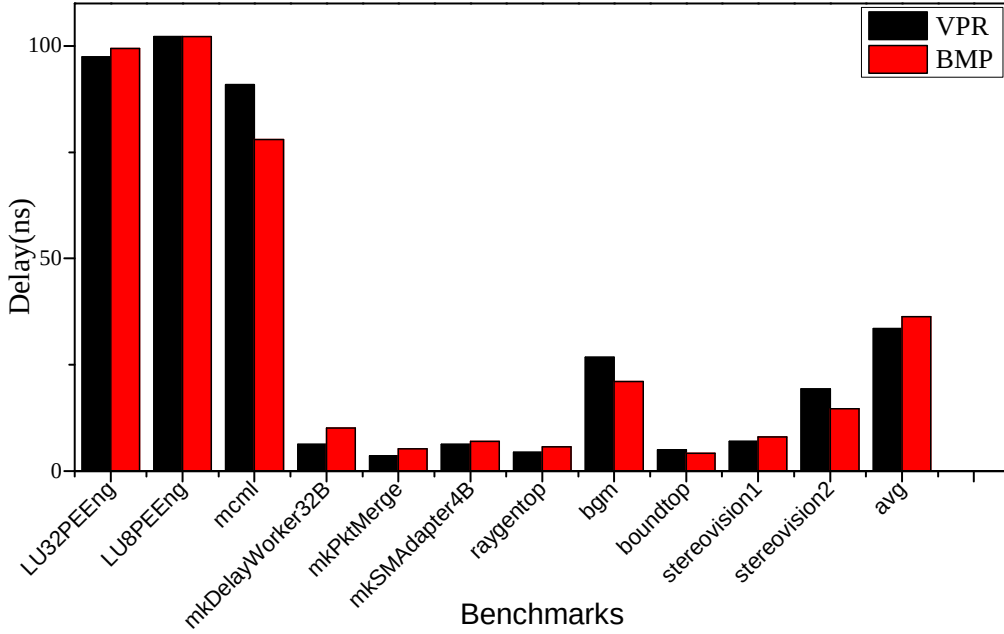


Figure 3.18: Delay comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases.

tool around 8.2%. The cases of LU8PEEng, mcml, bgm, boundtop and stereovision2 demonstrate delay is improved by 0.03%-24% as showed in Fig. 3.18. 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

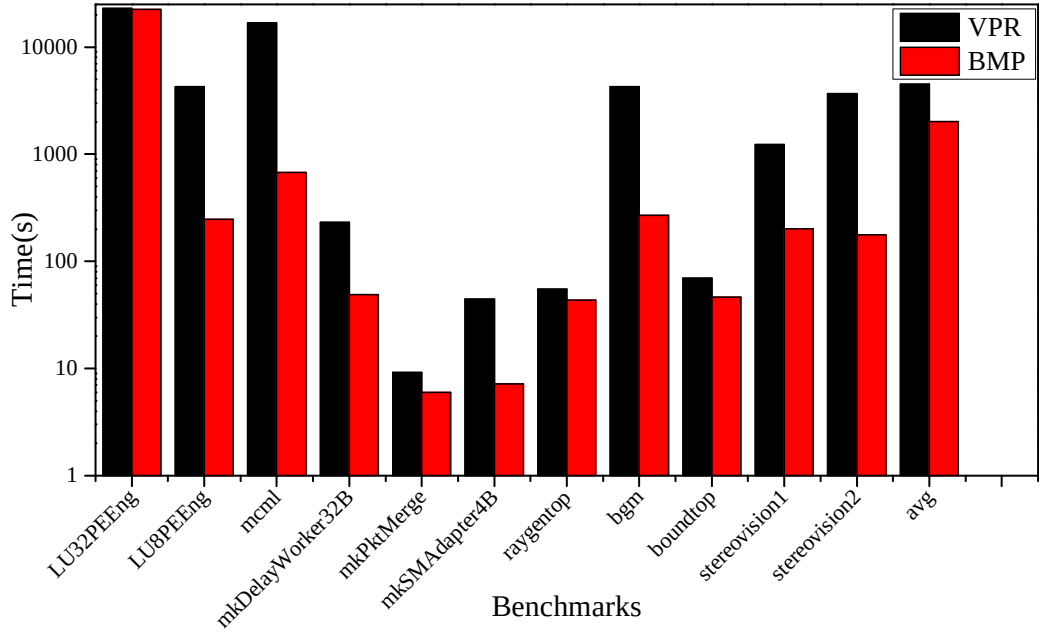


Figure 3.19: Execution time comparison between tile-based results (VPR) and BMP results (our approach) for the tested cases. (The y axis is in log10 scale.)

VPR flow can be improved by 3%.

Execution Time

The total execution time of the placement and routing of benchmarks for the original VPR and BMP are shown in Fig. 3.19 with black (left, VPR) and red (right, BMP) bars, respectively using log10 scale for y axis. 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

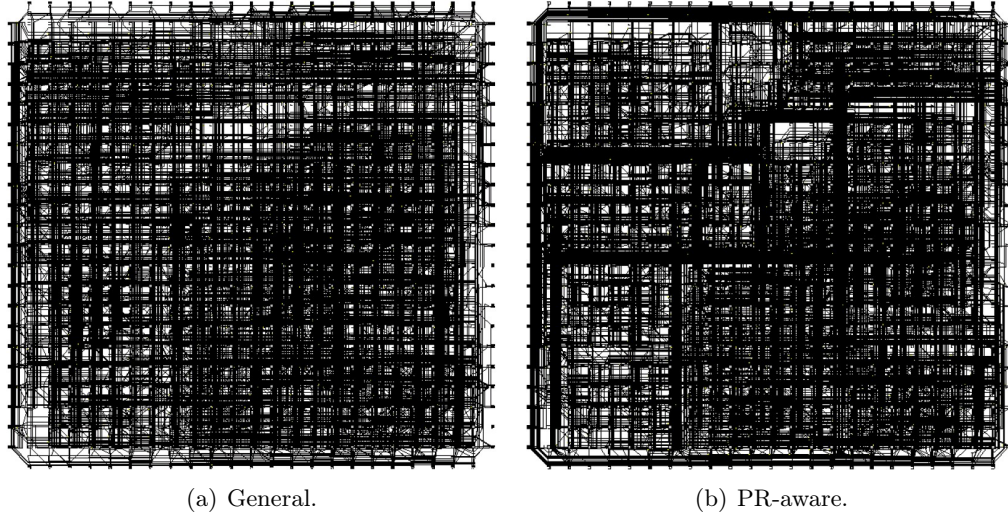


Figure 3.20: Boundtop routing result without/with PR-aware.

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 floorplanning 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.

3.7.3 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 3.20 shows the routing result without PR-aware and with PR-aware. Fig 3.20(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. 3.20(b) shows the routing result with PR-aware. Compared to the routing result without PR-aware in Fig 3.20(a), routing result with PR-aware in Fig. 3.20(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 [SR12] has observed a more than twice increase of routing efforts assuming using 30% more routing tracks than normal routing.

Table 3.4: BMP Comparison Between Modules with Different Ratios and Modules with Only One Ratio

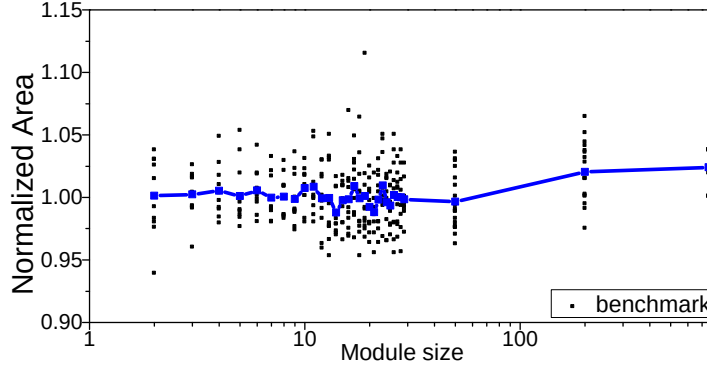
circuit name	one ratio		multi-ratio			
	Area	Delay	Area	AreaIncrease	Delay	DelayImprove
LU32PEEng	40610	9.89E-08	43130	6.21%	9.73E-08	-1.66%
LU8PEEng	9600	1.05E-07	10437	8.72%	1.00E-07	-4.77%
mcml	31439	7.76E-08	32016	1.84%	7.63E-08	-1.72%
mkDelayWorker32B	2856	9.34E-09	2860	0.14%	9.43E-09	0.94%
stereovision2	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%

3.7.4 Design Space Exploration

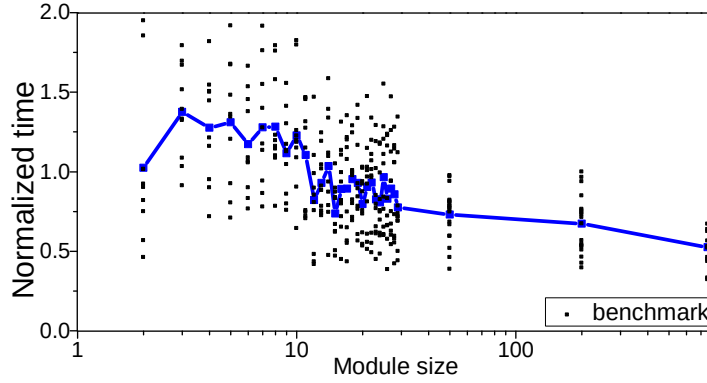
The proposed mapping flow searches solution randomly at beginning which may result in widely various results. Thus, it is set to run 10 rounds for each benchmark of each group of parameters in the exploration. It may give results of distinctive deviation, but multi-round simulation helps to reduce impact 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.

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 3.4. From the table, we can see that we can get better delay when we consider different ratios of



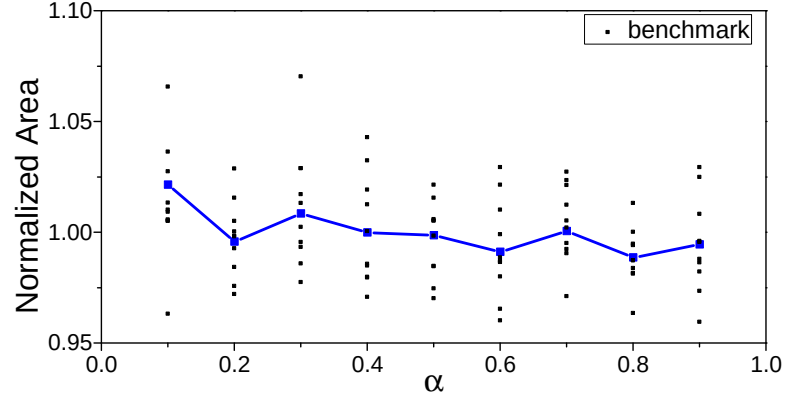
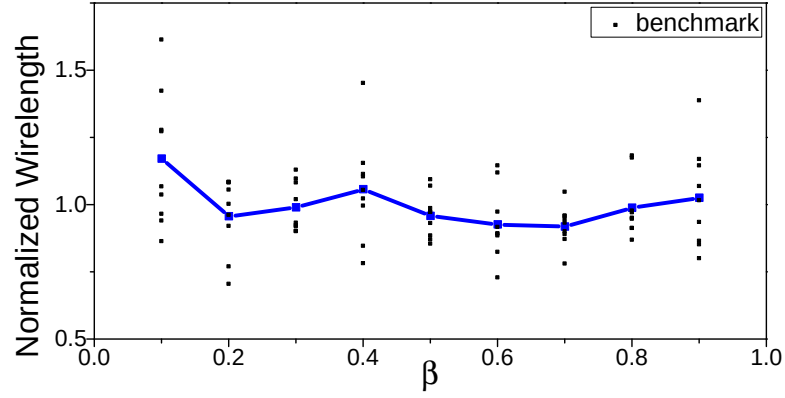
(a) Module size vs. area.



(b) Module size vs. computation time.

Figure 3.21: Design space exploration for definition of small module size. (The x axis is in log10 scale.)

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. 3.21 and we use log10 scale for x axis. 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. 3.21(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. 3.21(a) and time graph Fig. 3.21(b), we can see that blue line

(a) Parameters α (b) Parameters β **Figure 3.22:** Design space exploration for parameters in the cost function.

(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.

Parameters α and β in Eq. 3.1 of the section 3.4.3

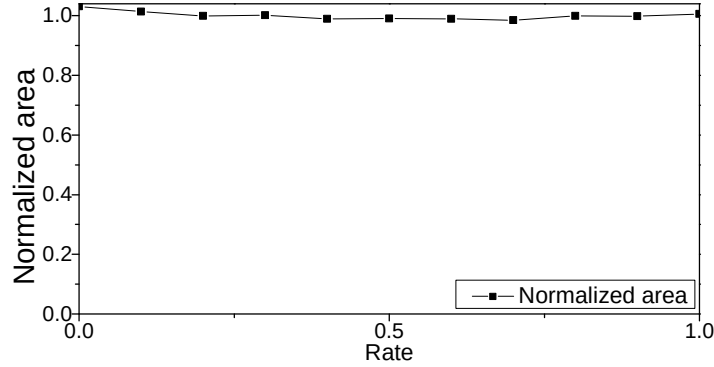
Parameter α and β of the cost function are two important parameters to guide CAD tools in placement. Quality of placement and execution time are strongly related to these two parameters. We demonstrate design space explorations for optimization of the two parameters here and the experiment results are shown in Fig. 3.22.

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. 3.22(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 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.

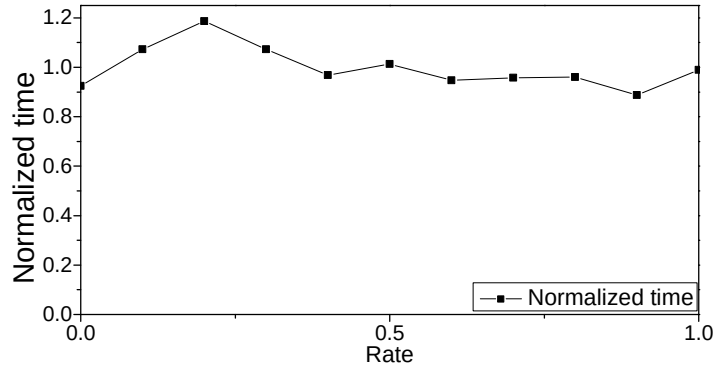
The β values are percentage between wirelength (track delay) and module delay in the placement. To discuss impact of β on the wirelength (track delay), we first preset α from 0.1 to 0.9 and β is varied from 0.1 to 0.9 with step of 0.1. The results of average wirelength (track delay) and deviation to mean value with various β are showed in Fig. 3.22(b). We can see that 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.

Probability of Operations in SA of the section 3.4.3

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 ratios into computation. Hence, the weight for loading operation is 1 means that the ratio of module can change each time. We only discuss the weight for the rest operations including moving module to another place (OP2) and swapping two modules (OP3). Their weights sum up to be 1. Fig. 3.23(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. 3.23(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.



(a) Move rate vs. area.



(b) Move rate vs. computation time.

Figure 3.23: Design space exploration for rate of moving operation.

3.7.5 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 3.1. Moreover, we also set channel width to be different values to evaluate the impact of different channel widths. The experiment results 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

Table 3.5: 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%

achieve delay improvement. Table 3.5 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.

3.8 Summary

In this chapter we proposed a library-based mapping flow which supports the partial run-time reconfiguration and replacement for multi-context functions in a PR region. Furthermore, it could reuse the resource in the module library to reduce the compilation time and to build large circuits. The proposed module-based placer (BMP) used 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 were explored in the experiments. Compared to the original tile-based flow, the delay of the proposed module based flow was slightly worse than the delay from the VPR tool 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 developed the pin-to-wire interface to support the PR-aware routing without adding extra cost.

The main aim of our flow was to facilitate the module reuse in later designs and amortize the module synthesis efforts through module reuse. The experiments still demonstrated the efficiency of our flow that it achieved execution time reduction largely for later designs with available modules under relatively small area and delay overhead. Moreover, the module synthesis time can be significantly reduced by parallel synthesis of different modules under different ratios.

CHAPTER 4

Dynamic Module Partitioning for Library based Placement on Heterogeneous FPGAs

Previous library based design flow faces several unresolved challenges. Firstly, they may result in large waste area between the modules due to the difference in module sizes. While utilizing multiple ratio modules can help to reduce the waste area, pre-synthesis of each module for different ratios is time consuming and would require a large library. In addition, when targeting new Field-Programmable Gate Array (FPGA) architectures, a new library will be required and re-synthesizing the library for different architectures is not feasible. To address these challenges, in this chapter we propose a dynamic module partitioning approach for the library based design flow to dynamically generate the appropriate shape of modules based on single-ratio modules in the library while efficiently utilizing the pre-placement module information.

4.1 Introduction

Traditional fine-grained tile-based placement may incur expensive searching time, especially for large-scale designs [GA14]. Library based design [WL10] and Intellectual Property (IP) reuse [ZPJU05] are promising solutions to address the challenge. It is observed that many next generation products are a variation of previous designs and may inherit previous designed components. Hence design reuse can significantly reduce the design efforts, development cost and turnaround time. Library based design is a design flow which can efficiently support design reuse, and synthesize new designs based on pre-placed

and routed modules stored in a library. Each module here represents a functional unit containing required logic blocks in a rectangle shape. Using the pre-designed components, library based design can significantly reduce the placement time and improve the design productivity [LPL⁺11, GSJ10]. However, waste (empty) area usually exists in the library based design due to the following reasons. First, the module size difference results in the waste area between modules. In order to best match the module shape and reduce the area gap between modules, multiple ratios of the modules are prepared in previous module based design flow [MC16]. However, it takes great efforts to pre-synthesize the multiple ratio modules. In addition, when the targeting FPGA architecture for the design differs from the one when building library, the placement may contain large waste area due to the mismatch of the heterogeneous resources locations. In order to efficiently reduce the waste area and adapt to different FPGA device architectures, an efficient placement optimization flow is needed. However, to the best of our knowledge, no such flow exists for library based design. The floorplanning based flow can search for the suitable region and shape to map a module in different devices. However, such flow cannot reuse the pre-placement information and hence requires longer running time.

In this chapter, we propose a dynamic partitioning approach for library based design which efficiently utilizes the pre-placement module information to better guide the placement of the modules. Partitioning a module means that we can change the shape of the continual region for placing the logic. In order to adapt to different architectures and reduce the waste area between modules as a result of module size variations, we propose a set of effective rules to select the candidate module and perform the dynamic partition of the module into several submodules to best fit the device. In addition, the approach can also improve the module based placement flexibility to address the fixed-outline constraint. Fixed-outline means that the width and height of the outline are given and we cannot place the modules outside the outline.

The experimental results show that with the proposed method, the area can be reduced up to 10% with marginal increase in delay and acceptable runtime.

The remainder of this chapter is organized as follows. Section 4.2 introduces the problem

formulation. Section 4.3 describes our dynamic module partition approach for heterogeneous FPGA placement. Experiment results are presented in section 4.4. Finally, we conclude our remarks in section 4.5.

4.2 Problem Formulation

In library based design, we assume different functional units are stored in a library as modules and a new design will be built based on the integration of required modules. Dynamic partitioning of modules are defined as partitioning a module into several submodules during placement. We formulate the problem of dynamic partitioning for library based placement on heterogeneous FPGA as follows. There is a library which consists of a set of n rectangular modules $B = \{b_1, b_2, \dots, b_n\}$. The logic in these modules has already been placed and routed and the information is used as a reference for optimally placing the logic in the final design. In other words, each module in the library has been pre-placed means that the blocks including CLBs, RAMs and DSPs inside each module have been already placed and these blocks have their unique positions. Each module b_i has a width and height represented by w_i and h_i , $1 \leq i \leq n$, respectively, and its aspect ratio is (w_i/h_i) . The placement of modules assigns the bottom-left coordinates represented by (x_i, y_i) to the modules such that resource requirements of the modules are met and no two modules overlap and each block inside the module has a valid unique position based on the pre-placed placement information of each module. The number of modules in a design depends on design requirements and the design can contain multiple copies of one module. During placement, modules can be dynamically partitioned into several submodules and the corresponding submodules need to be adjacent to each other to have a continual region for supporting PR related work and the region can be non-rectangular. The goal is to minimize the area and delay. If we consider the fixed-outline placement, we can set the width and height of the outline to be W_{max} and H_{max} . The width and height of the final placement must be less than W_{max} and H_{max} , respectively.

Several challenges exist in this problem.

- Placement is usually a time-consuming process since it is a NP-hard problem and hence, it is difficult to find the best solution [Don80].
- Large waste area may exist inside the modules and between the modules due to the module size difference and FPGA architecture mismatch.
- A good trade-off between the partition granularity, the running time and the area reduction needs to be achieved.

4.3 Our Proposed Dynamic Partitioning Flow for Library based Placement

In order to address the challenges discussed in the last section, we proposed the following key techniques. We also adopt B*-tree representation [CC06] and Simulated annealing algorithm (SA) [CC06] which are described in 3.4.2 and 3.4.3 of Chapter 3 for this work. We take a random solution as an initial solution for SA. We develop a set of efficient rules to guide the module selection and partition during placement, to eliminate the waste area inside and between modules and achieve a more compact final placement. In each round of placement, one module is selected and partitioned. The flow terminated when there is no candidate for module partition or the iteration number reaches the maximum or the temperature reaches the lowest temperature specified by the user.

4.3.1 B*-tree Based Module Placement with Dynamic Partitioning

In order to speed up the placement, we use the B*-tree representation for modular placement [CC06]. Each node of a tree represents a module. We extend B*-tree to handle dynamic partitioning in two aspects. Firstly, we can partition a node of a tree into several subnodes during placement. It means that the B*-tree is dynamic. The nodes and the number of nodes in the tree can be changed. Secondly, we add a constraint to ensure that these subnodes from the original node must be adjacent to each other to keep a continual region for supporting PR related work. The main reason for partitioning a

module into several submodules is to increase the flexibility of placing the module in order to reduce the area of the final design. These submodules from the same parent module combine together and they form into a single continual entity. The continual single entity is also a module which may have a different shape. Our approach can find a continual region for placing these submodules together and the region meets the resource requirements for these submodules. In other words, the placement must be successful. We do not consider real routing for these submodules and use wirelength and module delay to evaluate the design during placement. We first obtain an efficient placement for all the modules and connect them through PR-aware routing step. If routing fail, we can adopt the following ways. 1. We perform replacement inside the modules which are related to the failed nets. 2. We expand the region for placing each module. 3. We change the framework from scratch which means redoing the flow for the entire 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). No matter whether we partition modules, the design may face the same problem which is widespread in VLSI and FPGA design flow. When we use these options, we need to give the highest priority to option 1 and give the lowest priority to option 3. If we partition a module, we assume that the module delay does not change since we cannot get the accurate delay during placement. We use the wirelength to evaluate its impact and use the cost function including area, wirelength and delay to evaluate the partition operation and design.

Since each module is presented by a node in B*-tree, partitioning a module into several submodules corresponds to partitioning a node into several subnodes. The logic function in a module is the same as the function consisting of these submodules in a continual region. Fig. 4.1 shows an example for partitioning a module A into two submodules which helps to reduce the final design area. The blocks with black and orange colors represent they have been used and the blocks with white color are not used. Fig. 4.1(a) shows that if we vertically partition A with the red line, we will get two submodules A1 and A2 shown in Fig. 4.1(b). When we place the module A, we finally get the placement result(12 columns x 8 rows) shown in Fig. 4.1(c). When we place the two submodules A1 and A2, we finally get the placement result(10 columns x 8 rows) shown in Fig. 4.1(d).

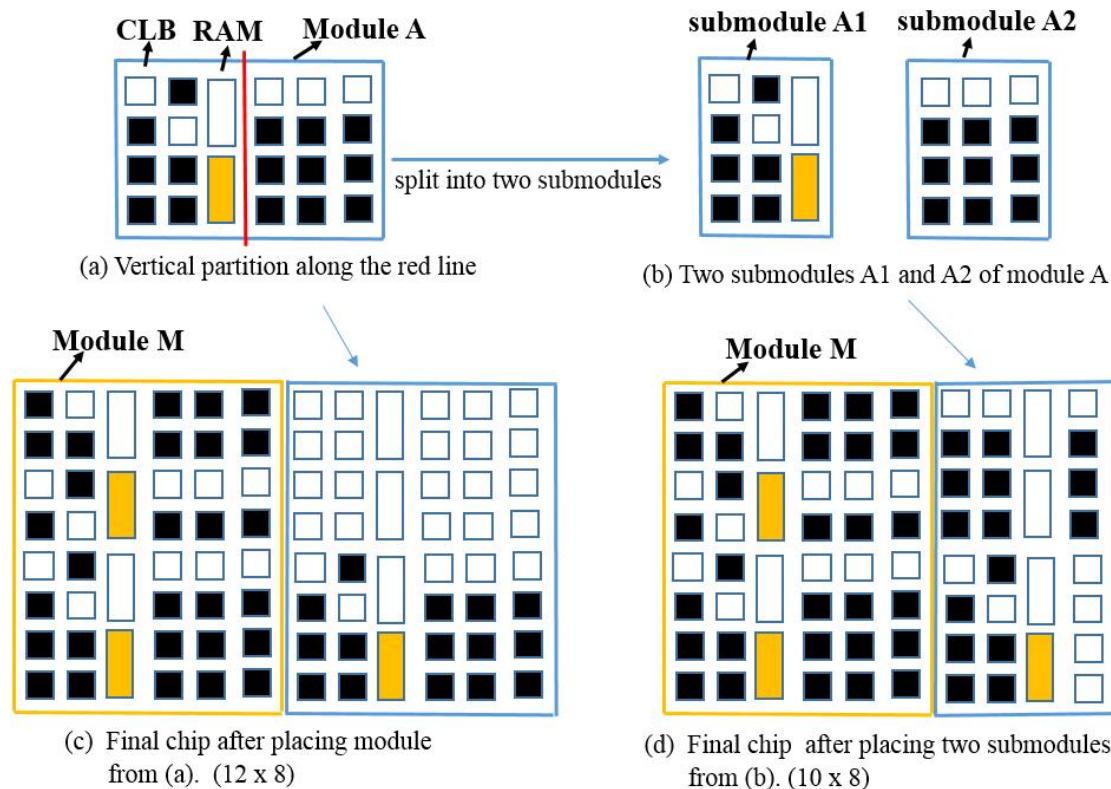


Figure 4.1: Vertically Partition a module A into A1 and A2

We can see that this module partition corresponds to generating a new module ratio which efficiently reduces the waste area of the resulted placement.

Fig. 4.2 shows the B*-tree corresponding to the example shown in Fig. 4.1. The nodes n_0 , n_1 , n_{11} and n_{12} of the trees represent the region with orange color, module A, module A1 and module A2, respectively. The n_{11} and n_{12} are two subnodes of node n_1 . We keep corresponding submodules adjacent to each other in order to have a continual region and minimize module delay. When we map a module into a architecture, we choose the region which meets the resource requirements of the module with minimum area.

To explore the solution space, we apply the following operations used in the SA [CC06].

- OP1: Load a module with a shape (aspect ratio).
- OP2: Move a module to another place.
- OP3: Swap two modules.
- OP4: Split a non-split module into several submodules and keep the corresponding

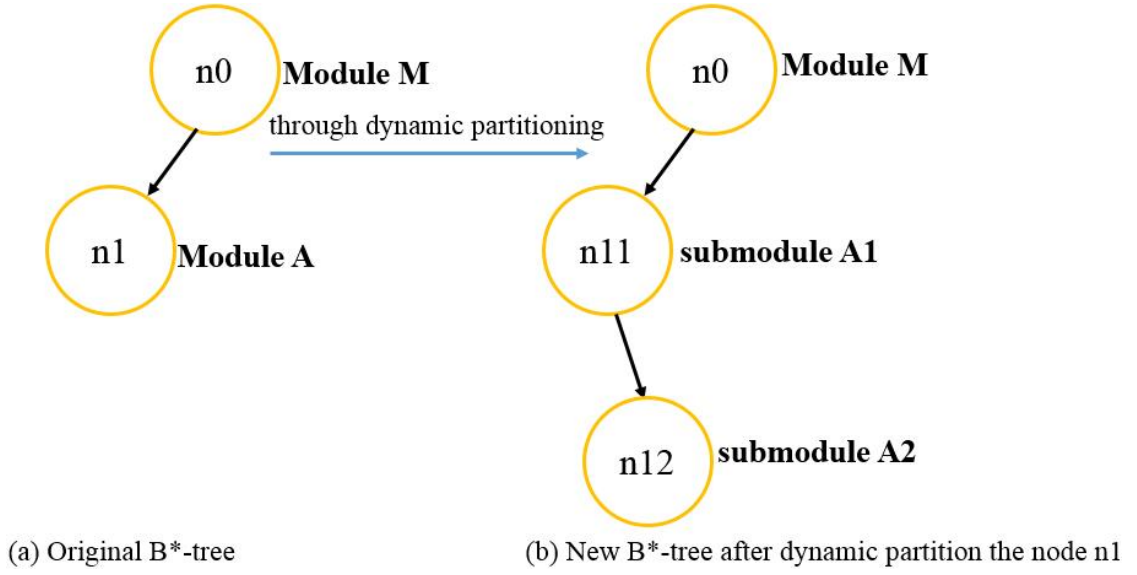


Figure 4.2: Dynamic partitioning node n1 into two subnodes n11 and n12

submodules adjacent to each other to have a continual region.

For OP2 and OP3, we need to check whether the modules are split. If they are non-split modules, we can operate on them directly. If there is a split module, we need to find whether it has sibling split modules. If it has sibling, they need to be moved or swapped together.

Cost Metric in the Placement Flow

Different weights are given to the operations to control the probability of choosing each operation. After each operation, we use the commonly linear weighted cost function [CC06, WW91] as shown in Eq. 4.1 to estimate the quality of placement.

$$Cost_{system} = \alpha \frac{A}{A^*} + (1 - \alpha) \left(\beta \frac{D_T}{D_T^*} + (1 - \beta) \frac{D_M}{D_M^*} \right) \quad (4.1)$$

Here $Cost_{system}$ is the total cost of the placement. A and A^* represent the current total and average placement area respectively. D_M and D_T are current total module delay and total track delay (wirelength) respectively. D_M^* and D_T^* are the average module delay and track delay respectively. 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. Since when we get the placement area A of a design, we can calculate the overall waste area of the placement. The total waste area is the value of the placement area A of a design minus the total area of all the modules. Thus A can reflect the waste area.

4.3.2 Partition Strategy

If large waste area exists in the placement result, we can partition the modules into submodules and then use cost function Eq. 4.1 to guide the placement for a more compact solution. In general, we can partition modules in horizontal and vertical directions. However, there is a constraint that carry chain usually needs a continual region in vertical direction and cannot be separated. The carry chain is the feature which allows FPGAs to be efficient at arithmetic operations such as counters, adders. Thus, we only limit to vertical partition in our work and we consider the width of modules when choosing the modules for partitioning. In this section, we first present the rules for module selection and partitioning to adapt to different underlying architectures and reduce the waste area. Then we discuss how the rules are adjusted to solve the fixed-outline placement.

The Rules for Choosing Modules

When we choose the modules for partitioning, we mainly consider two aspects: the overall waste area and the shape (width and height) of the modules. The overall waste area can be treated as a two dimensional (2D) region and how to calculate the total waste area is described in the last part of 4.3.1. Our goal is to make full use of the waste area to get a more compact placement. We observe if the module size is smaller than the overall waste area, we can directly move the module without partitioning it. Thus we choose the modules with size larger than overall waste area as candidates for partitioning. Moreover, since we only perform vertical partitioning due to carry-chain constraint, we give higher priority to the modules with larger width. We summarize the rule of module selection for partitioning in Alg. 4.1.

Algorithm 4.1: The Rule for Choosing Modules for Partitioning

- 1: **if** flow is aiming to solve fixed-outline placement **then**
 - 2: choose the modules beyond the boundary outline
 - 3: **else if** flow is targeting different architectures than the one for library **then**
 - 4: choose all the modules for partitioning
 - 5: **else**
 - 6: collect the modules with area larger than overall waste area
 - 7: choose the module with the largest width for partitioning
 - 8: **end if**
-

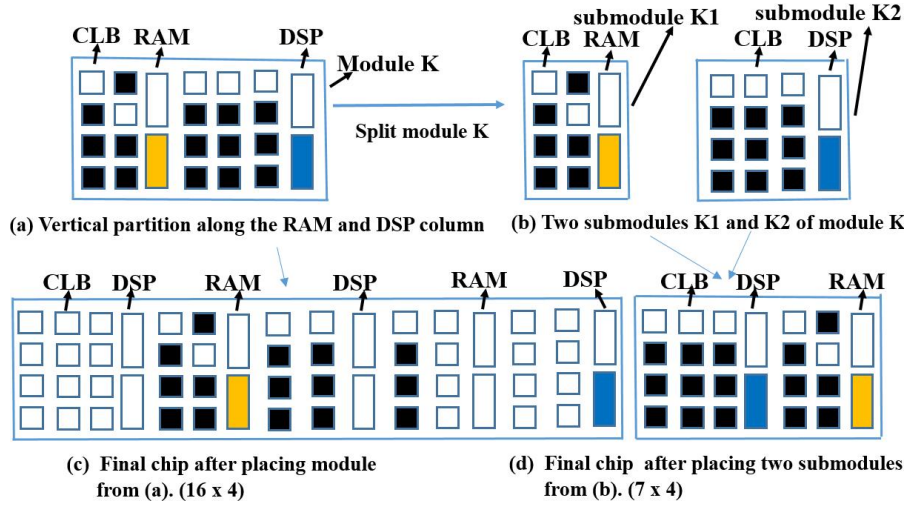


Figure 4.3: Vertically Partition a module K into K1 and K2 based on RAM and DSP columns

The Rules for Partitioning Modules

We adopt different partition strategies for a module based on its resource requirement and the target FPGA architecture. If the design is targeting a different FPGA architecture which is different from the one current library uses, we need to partition the modules based on the Block Memory (RAM) and Digital Signal Processing block (DSP) columns inside the modules since those resources are located in different locations in different architectures and possible to create empty columns inside the modules and we aim to use the relative locations of the blocks which have already been placed in original modules stored in the library. More specifically, the non-empty RAM and DSP columns divide the module into several submodules. Fig. 4.3 shows an example of the case. If we map

Algorithm 4.2: The Rule for Partitioning a Module

```

1: if flow is aiming to solve fixed-outline placement then
2:   vertically partition a module along the chip boundary and mark the corresponding
   submodules as partitioned
3: else
4:   if flow is targeting a different architecture and the module is the first time to be
   partitioned then
5:     vertically partition the module based on the RAM and DSP columns inside the
     module and marked these submodules as partitioned. Then add the submodules
     with size larger than the overall waste area to the partition list
6:   else
7:     compare the overall waste area and the area of the chosen module, then can
     calculate how many parts (N) that we need to partition to make the submodule
     size (the area of the chosen module/N) smaller than the overall waste area
8:     if part number required is larger than two then
9:       choose required partition number
10:    else if part number required is two then
11:      Compare the potential area reduction using part two and part three partition
      and finally choose the best partition
12:    end if
13:  end if
14: end if

```

the module K into a different architecture, we can get the final result (16 columns x 4 rows) shown in Fig. 4.3 (c). However, if we partition the module K based on the RAM and DSP columns, we can get the smaller final placement result (7 columns x 4 rows) shown in Fig. 4.3 (d). After this round of partition, then submodules will be treated as placing in the same architecture in the following iterations.

If the design is targeting the same architecture as the library, then we can focus on partitioning the module aiming to reduce the waste area between the modules. As we have discussed, we aim to partition the selected module to be submodules with the size of each submodule less than the overall waste area of the current iteration. It is a tradeoff

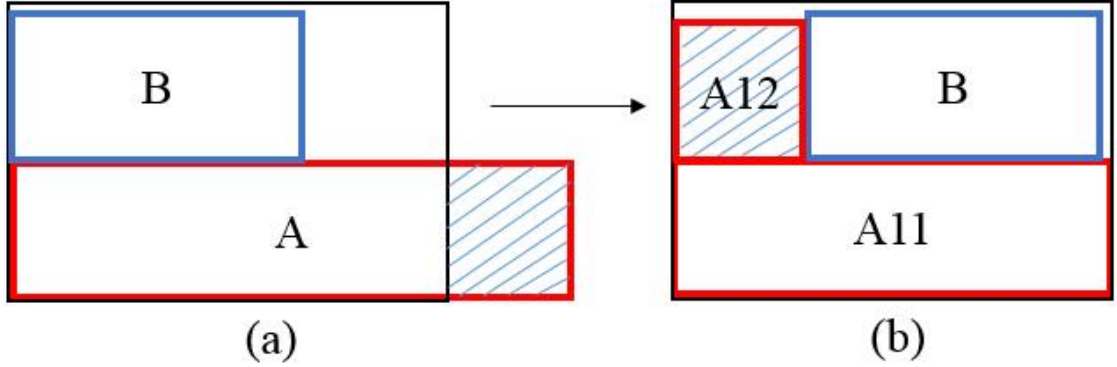


Figure 4.4: Vertically Partition a module A into submodules A11 and A12

problem between the number of parts for partitioning a module and runtime. Usually partitioning into two submodules can satisfy the requirement. However, since smaller submodule has better flexibility to fit in the wasted area, we will also consider three-part partition of the module and choose the partition which can better fit in the waste area shape. Please note that since we only consider vertical partition, for some tall module, when very small width is needed to generate the submodule, we will not perform the partition to avoid the extreme ratio out of normal range, e.g. 0.2-5.0. The rules for module partitioning are summarized in Alg. 4.2.

Solving Fixed-outline Constraint

When having fixed-outline constraint, the boundary modules may be partitioned. Fig. 4.4 is an example that dynamic partitioning solves the fixed-outline constraint. In Fig. 4.4 (a) the outline with black color is the boundary for a design and module A is beyond the boundary. We vertically partition module A into submodule A11, A12 and keep the submodules be adjacent to each other in order to have a continual region and the locations of A11 and A12 meet the B*-tree requirement [CC06]. Through partitioning we can improve the flexibility of modules. It provides an opportunity to address the fixed-outline constraint that may not be solved in previous module based placement design and we consider the wirelength during partition.

Table 4.1: Architectures Used in the Experiment

Architecture In VPR Suite						
BLEs	RAMStart	RAMGap	RAMH	DSPStart	DSPGap	DSPH
4	3	5	2	5	5	4
Architecture Arch1						
BLEs	RAMStart	RAMGap	RAMH	DSPStart	DSPGap	DSPH
4	5	5	2	7	5	4

Table 4.2: Benchmarks in VPR Suite

circuit name	modules	CLB	RAM	DSP
LU32PEEng	4	17897	32	150
mkPktMerge	4	34	0	15
mkSMAadapter4B	4	393	0	5
raygentop	15	427	7	1
mcml	5	15504	30	38
mkDelayWorker32B	8	1117	0	41
LU8PEEng	4	5102	8	45

4.4 Experimental Results

4.4.1 Experimental Setup

We implement our flow in the C++ programming language and perform our experiments on a IBM server x3650 with Intel Xeon(R) CPU and 42 GB DDR2 RAM. In general, we use the architecture file which is similar to k6_N10_memDepth16384_memData64_40nm_timing, in the VTR project [RLY⁺12] and we perform experiments based on this architecture. We set the channel width to be 200 and the percentages for length-4, length-2 and length-1 wires are 60%, 20% and 20%, respectively. The parameters for the architecture are summarized in Table 4.1 (Name with Start represents the start column, and with H represents the height). We compare our approach with the previous library based flow

Table 4.3: Comparison between Module Based Placement Flow and Our Proposed Approach Considering Modules with only One Ratio

circuit name	Module Based Flow			Our Proposed Approach		
	Area	Delay (second)	Time (second)	Area	Delay (second)	Time (second)
mkSMAAdapter4B	930	6.94E-09	9.14	812	6.62E-09	11.65
mkDelayWorker32B	3721	9.69E-09	110.93	2970	1.01E-08	213.32
mkPktMerge	340	5.74E-09	8.08	225	4.57E-09	37.68
raygentop	1296	5.44E-09	79.77	1156	5.30E-09	172.52
mcml	32220	7.89E-08	717.98	35532	1.15E-07	632.43
LU8PEEng	14161	9.94E-08	292.76	14042	1.00E-07	238.47
LU32PEEng	43890	1.02E-07	5574.89	43470	1.02E-07	4935.23
Average	1	1	1	-10%	3%	80%

(an open source code) in [MC16, MCZ⁺16]. For fair comparison, we use the same random seeds of SA used in [MCZ⁺16] and the same benchmarks shown in Table 4.2 and set the module ratios to the same range as 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.6, 2.5, 5.0 for the case described in [MCZ⁺16].

4.4.2 Area, Delay and Runtime Comparison Considering Modules with One Ratio

As we have discussed, preparing multiple ratio modules may require great efforts. In order to demonstrate the efficiency of the proposed dynamic partitioning flow based on single ratio module, we compare the area, delay and runtime between our dynamic partition based flow and previous BMP flow (an open source code) in [MC16] based on only one module ratio. Each ratio is randomly chosen for each module and will not change. The reason is that we choose a general placement to verify our approach. Table 4.3 shows that our proposed approach reduces the area by 10%. The delay increases by 3% and running time increases by 80%. The reason for the delay increase is that we cannot accurately estimate the delay in module based placement stage as the problem most delay estimation methods face. In order to minimize the delay, it takes longer time to search for the best placement. However, it is a trade-off between delay and running time which can be further tuned through cost function modification to achieve the user's goal.

Table 4.4: Comparison between Module Based Placement Flow Considering Multiple Ratios and Our Proposed Approach

circuit name	Module Based Flow			Our Proposed Approach		
	Area	Delay (second)	Time (second)	Area	Delay (second)	Time (second)
mkSMAAdapter4B	841	6.50E-09	9.41	812	6.60E-09	7.37
mkDelayWorker32B	3481	1.03E-08	51.38	2970	1.04E-08	43.57
mkPktMerge	256	4.51E-09	5.4	441	4.58E-09	8.95
raygentop	1156	5.69E-09	18.5	1156	5.60E-09	18.64
mcml	32220	7.85E-08	603.41	35721	8.42E-08	660.5
LU8PEEng	11881	1.02E-07	178.56	14161	9.94E-08	187.45
LU32PEEng	42436	9.90E-08	4447.76	43056	9.89E-08	1914.53
Average	1	1	1	12%	1%	-2%

4.4.3 Area, Delay and Runtime Comparison Considering Modules with Multiple Ratios

We also compare the results between our dynamic partition flow based on one ratio and previous BMP flow considering multiple ratios (an open source code) in [MC16] to evaluate how well the dynamic partition flow can perform. Table 4.4 shows that our proposed approach can achieve the similar delay with 12% increased area in less runtime. It is impractical and impossible to get all the aspect ratios for each module and if we target a new architecture, pre-synthesis for each module is time consuming and not feasible. The results show that our partitioning approach using only one ratio can obtain encouraging results due to the similar delay (performance) of the design and help to reduce complexity and time required by preparing the library.

4.4.4 Area, Delay and Runtime Comparison for Different Architectures

Our approach allows the library to adapt to different architectures and we compare the area, delay and runtime between module based flow and our proposed approach. In particular, we focus on one ratio for fitting to a different architecture namely ARCH1 shown in Table 4.1. The reason is that pre-synthesis of each module for different ratios at a new architecture is time consuming and re-synthesizing the library for different architectures is not feasible. Thus, we try to propose a solution to address this problem.

Table 4.5: Comparison between Module Based Placement Flow and Our Proposed Approach for Different Architecture

circuit name	Module Based Flow			Our Proposed Approach		
	Area	Delay (second)	Time (second)	Area	Delay (second)	Time (second)
mkSMAAdapter4B	1089	6.85E-09	10.95	1089	7.33E-09	12.1
mkDelayWorker32B	4032	9.45E-09	66.39	2809	9.77E-09	124.47
mkPktMerge	528	4.61E-09	10.61	324	4.81E-09	15.74
raygentop	2601	8.78E-09	46.85	1681	6.38E-09	143.89
mcml	35258	9.23E-08	606.23	38316	1.23E-07	4999.21
LU8PEEng	12543	1.03E-07	189.71	10100	1.13E-07	309.57
Average	1	1	1	-19%	5%	190%

The other parameters for the architecture is the same as described above. Table 4.5 shows that our proposed approach reduces the area by 19% with 5% delay increase. There is an increase in the running time since the number of modules (nodes) increased is larger than before [MC16]. When we perform partition based on RAM and DSP columns, the number of submodules for each module depends on the number of RAM and DSP columns. However, note that the increase in running time is still much less than the time and efforts for preparing a new library with multiple ratios.

In summary, all the experiment results show that our approach can efficiently eliminate the waste area and can reduce the area up to 10% and 19% for the same and different architectures respectively with small increased delay in reasonable runtime. It significantly saves the complexity and efforts for preparing modules for different ratios and architectures.

4.5 Summary

In this chapter we proposed a new dynamic partitioning approach for library based design which improved the flexibility of modules and tried to reduce the waste area inside and between modules. Our approach provided an opportunity to address the fixed-outline placement which was an unsolved problem in module based placement flow. The experiment results showed that our approach was efficient and could reduce the area by up to 10% with marginal increase in delay and acceptable runtime. The flow

reduced the significant complexity and efforts for large number of module preparations with different ratios and targeting different architectures.

CHAPTER 5

Modular Placement for Interposer based Multi-FPGA Systems

In this chapter, we explore the automatic workflow mapping in the interposer based multi-FPGA system. Novel device with multiple Field-Programmable Gate Arrays (FPGAs) on-chip based on interposer interconnection has emerged to resolve the IOs limit and improve the inter-FPGA communication delay. However, new challenges arise for the placement on such architecture. We propose a fast two-stage modular placement flow for interposer based multiple FPGAs aiming for delay optimization with the incorporation of a detailed interposer routing model for wirelength and delay estimation.

5.1 Introduction

Recently, interposer-based FPGAs are proposed to significantly improve the logic capacity, interconnection bandwidth and performance [HPB14]. For instance, Xilinx Inc., a major FPGA vendor, firstly developed the largest interposer-based FPGA, the Virtex-7 XC7V2000T, which has 4 dies and 1.95 million logic elements [HPB14]. A general two-and-a-half-dimensional (2.5D) FPGA with interposer is shown in Fig. 5.1 [HPB14], where four FPGA dies are placed vertically and are connected by the interposer. The interposer is an electric interface routing for connecting different components and can be used to spread connections, e.g., providing alternative routes for a connection [HPB14]. In order to connect between different dies, the signal needs to go through the internal track of the current die, microbump, and super long line (SLL) of interposer, and then into the other die.

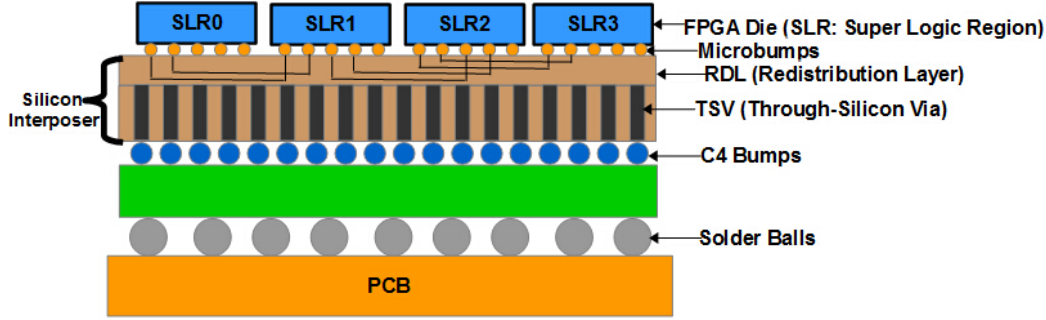


Figure 5.1: A general 2.5D FPGA with interposer.

FPGA placement is a very important step for synthesis in modern FPGAs [LR08] which significantly affects the implementation delay and the optimal placement is a NP-complete problem [Don80]. The increasing design complexity and logic capacity of the FPGA system, especially the multi-FPGA system, have further complicated the problem. In traditional design flow, fine-grained tile-based (CLB level) placement is performed to achieve the efficient solution. However, it usually requires a prohibitively long searching time from hours to days, especially for a large-scale design [GA14]. To accelerate the placement of large-scale design, module based placement has been employed for single FPGAs [AGBD07]. Here, module represents a rectangle region containing some logic resources and each module can be a reconfigurable region [Xil12]. To our best knowledge, there is no existing flow to support the module based placement for multiple FPGAs, especially taking into consideration the new interconnect architecture through interposer.

In this chapter, we propose a new flow to address the placement for multi-FPGA system with interposer targeting delay optimization. Our approach consists of two steps. Firstly a force-directed placement is performed to obtain an efficient initial solution. Secondly, the initial solution is represented by the hierarchical B*-tree (HB*-tree) [CC06] and based on it, simulated annealing is performed to obtain the refined solution. A detailed routing model for interposer is built and integrated in the flow for addressing the interposer impact on the placement. The B*-tree representation is introduced to speed up the search and convergence of the solution [CC06]. The HB*-tree matches the multi-FPGA system well where the multi-FPGA can correspond to a B*-tree and each chip can correspond to a sub B*-tree to enable the swap of modules in different chips. In addition, we also exploit the impact of different width-height ratios of the modules for delay optimization.

Our results demonstrate that 1) Interposer impacts the placement result significantly and compared with the traditional model, our proposed model can reduce the wirelength, total delay, delay from nets crossing the interposer by 8%, 7%, 19%, respectively; 2) The ratios of the module impact the placement results. When considering the different ratios of modules, the wirelength, total delay and delay from nets crossing interposer are improved by 8%, 4%, 9% compared to the results considering modules with only one ratio; 3) Our proposed flow can speed up the placement and achieve higher quality compared to the state-of-the-art min-cut based flow derived from [KAKS97].

The main contributions of our work are as follows:

- We develop a wirelength estimation model involving interposer according to the commercial FPGA with interposer [HPB14, Inc13]. We optimize timing and consider the interposer simultaneously.
- We first propose a HB*-tree based modular placer for the multi-FPGA system with interposer.

The remainder of this chapter is organized as follows. Section 5.2 formulates the modular placement for the multi-FPGAs. Section 5.3 describes our proposed flow for multi-FPGAs with interposer model including the force-directed initial placement and HB*-tree based solution refinement. Experiment results are presented in Section 5.4 and we conclude our remarks in Section 5.5.

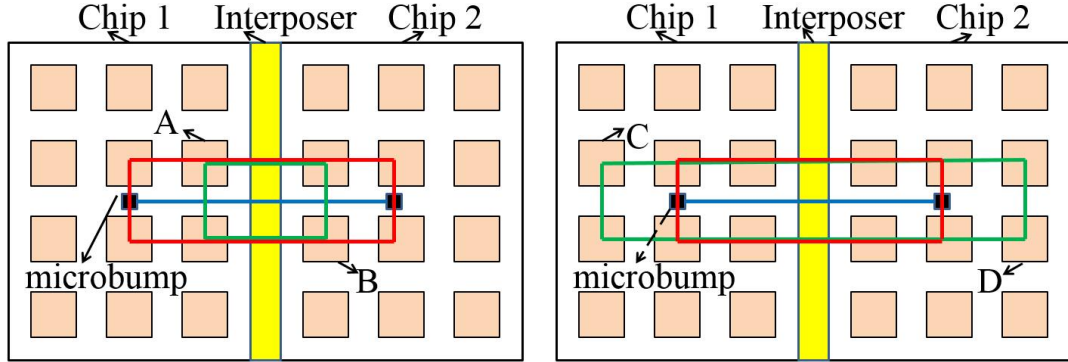
5.2 Problem Formulation

In this work, we aim to develop the modular placement flow for the interposer-based 2.5D FPGAs. A wirelength estimation model for the interposer routing is developed for the placement flow. We describe the notations used in the placement problem below. $E = \{e_1, e_2, \dots, e_m\}$ is the set of dies in the 2.5D FPGAs. $IO = \{io_1, io_2, \dots, io_k\}$ is the set of available IOs in the 2.5D FPGAs. $B = \{m_1, m_2, \dots, m_n\}$ is the set of modules to be placed in the 2.5D FPGAs.

We define the problem of interposer based multi-FPGA placement as follows. Given a set of B with n rectangular modules $B = \{m_1, m_2, \dots, m_n\}$, where each module has a width and height denoted by w_i, h_i , $1 \leq i \leq n$ respectively. Each module in the library has been pre-placed which means that the blocks including CLBs, RAMs and DSPs inside each module are already placed and these blocks have unique positions. The aspect ratio of module b_i is defined by w_i/h_i . Given a set of E with m dies $E = \{e_1, e_2, \dots, e_m\}$, where each die has the same width and height denoted by EW, EH . Given a set of IO with k IOs $IO = \{io_1, io_2, \dots, io_k\}$. A placement $p = \{(x_i, y_i)\} (1 \leq i \leq n)$ with modules (m_i) is an assignment of the rectangular modules such that the bottom-left corner coordinate of module m_i is assigned to (x_i, y_i) , no two rectangular modules overlap and each block inside the module has a valid unique position based on the pre-placed placement information of each module, and the different dies are connected by interposer. A routing model described in 5.3.1 is used to estimate the wirelength and delay through interposer. We consider each function in the overall design as a rectangle module in this work, and the objective is to optimize the total delay of the mapped circuit as well as meet the IO and area constraints.

5.3 Proposed Modular Placement Flow

The complexity of the placement for a multi-FPGA system is much larger than that of a single FPGA. An efficient approach should consider both the runtime and placement quality. In order to optimize the placement as well as speed up the convergence, we take two stages to achieve the placement for the interposer based multi-FPGA system. Firstly, we adopt the force-directed method [ZML⁺07] for its global property to obtain a high-quality initial placement and then partition modules into different dies. Next, we employ the simulated annealing (SA) [RLY⁺12] for its efficiency and effectiveness in finding the approximately optimal solution based on the initial solution derived from the previous stage. In order to accelerate the refinement, we introduce the hierarchical B*-tree (HB*-tree) [HC13] to enable a fast search [CC06] and convergence. Moreover, a wirelength estimation model for the interposer is developed to allow a more accurate evaluation of the interposer impact on the placement flow. The relation between the



(a) Original bounding box smaller than SLL length. (b) Original bounding box larger than SLL length.

Figure 5.2: Difference of wirelength estimation when using interposer compared traditional bounding box calculation.

force-directed based placement and the simulated annealing based placement is that force-directed based placement is used for preparing initial placement as an input for simulated annealing based placement and they are separated stages.

5.3.1 Routing Model for Interposer

Based on the study of Xilinx commercial FPGA with interposer, e.g., Virtex-7 XC7V2000T [HPB14], we know that the logic block in one FPGA die connects to another logic block in a different FPGA die in the following way. Firstly, the logic block in one FPGA die connects to the track (length-12) and the track connects to the corresponding microbump. Next, the microbump links to another microbump of the different FPGA die through the super long line (SLL) in the interposer [Inc12]. Then, the signal goes through microbump and corresponding track and finally connects to the logic block in a different FPGA die. The SLL wires for connecting the adjacent FPGA dies through interposer have the same length as the chip length according to the Xilinx interposer-based FPGA, Virtex-7 XC7V2000T [HPB14, Inc13]. To extend the model for general multiple FPGAs with interposer interconnection, we assume that any two adjacent chips can connect to each other directly through interposer and the SLL wire for connecting the adjacent FPGA dies through interposer have the same length as the die length, and the non-adjacent chips are connected by the intermediate chips.

We use an example as shown in Fig. 5.2 to illustrate the difference of wirelength estimation

when considering the interposer compared to traditional bounding box calculation. We assume that the CLB A in the FPGA die one (Chip 1) connects with the CLB B in FPGA die two (Chip 2) and the CLB C in the FPGA die one (Chip 1) connects with the CLB D in FPGA die two (Chip 2). For the case in Fig. 5.2(a), we cannot use the green bounding box for wirelength estimation between CLB A and CLB B as in the traditional model, since there is routing restriction that SLL only connects to the local tracks at its ending points. Instead, we consider the SLL length and use the red bounding box for wirelength estimation which is more accurate to model the interposer. For the case in Fig. 5.2(b) where the bounding box for wirelength estimation between CLB C and CLB D is larger than the SLL length, the traditional way of bounding box calculation can be applied.

5.3.2 Force-directed Placement

We adopt the force-directed approach since it is a scalable approach for large designs and able to avoid being trapped in local minimum solution. It is also suitable for optimizing average delay of the FPGA. The force-directed algorithm simulates the mechanics problem in which particles are attached to springs and their movement obeys Hook's law [ZML⁺07, OJ04]. We build the system of quadratic equations based on connectivity between modules [OJ04, VC05].

$$T = \sum_{\substack{0 \leq i < n \\ 0 \leq j < n}} c_{ij} [(x_i - x_j)^2 + (y_i - y_j)^2] \quad (5.1)$$

where (x_i, y_i) and (x_j, y_j) are the coordinates of the modules i and j and n is the number of modules. c_{ij} is the weight of the connection between two modules i and j . We can convert the objective function into a matrix-vector notation [VC05, SSJ08].

$$T = \frac{1}{2} x^T C x + d_x^T x + \frac{1}{2} y^T C y + d_y^T y + constant \quad (5.2)$$

where x, y are coordinates of the modules and d_x, d_y are n dimensional vectors. C is n

$\times n$ matrix and is the net stiffness matrix [VC05, SSJ08] which reflects the connections between modules and [VC05] give a detailed description for how to get the C . We can get a minimum solution by solving the derivative function for Eq. 5.2. We set its first derivative to zero about x and y respectively and then we get two following functions. We can solve the following systems of equations to minimize the wirelength and delay.

$$Cx + d_x = 0, Cy + d_y = 0 \quad (5.3)$$

Force Computing

We define two types of force for balancing the placement result: Filling Force and Attraction Force [ZML⁺07]. The two forces are combined together using linear weighted function to form the d_x and d_y which is described in [ZML⁺07]. We adopt the approach (Barnes-Hut quad-tree for n-body force calculation) described in [BH86] to compute the filling force due to its accuracy, $O(n \log n)$ complexity and high-quality of force calculation. To tradeoff the computation time and accuracy, in our work we model a quad-tree with at most ten levels for the placement region and recursively divide the placement region into grids with the termination condition that the bin size is not less than $(w_{min} + h_{min})/4$ [MTB00], where w_{min}, h_{min} are the minimum width and height of all the modules.

Filling Force (f_x^F, f_y^F): it is used to remove overlap between modules and distribute the modules evenly over the 2D placement area. It pushes modules away from the regions with high density and pull the modules towards regions with low density in the 2D space (filling force direction). We define the density as the total module area that covers the bin. The filling force of each bin is equal to the bin density of each bin. The filling force of each module is equal to the sum of the filling force of all the bins that the module covers. We add an additional force for the modules which lie on the partition boundary to prohibit them from crossing the interposer and ease the final partition stage, since if modules are placed in a partition boundary, it is difficult to choose which die they belongs to.

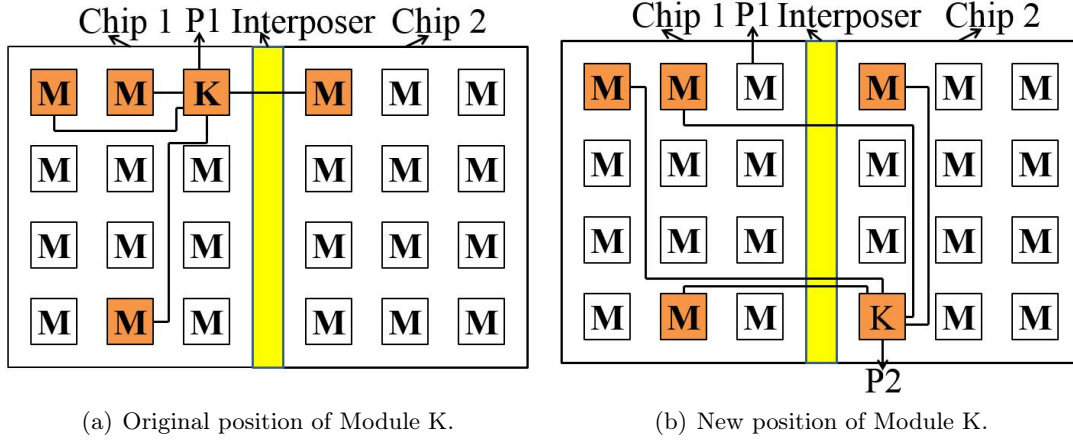


Figure 5.3: Example for attraction force considering interposer.

Attraction Force (f_x^A, f_y^A): it is used to reflect the relation between the old coordinate (start point of the force) and new coordinate (end point of the force) of the module. The two different positions of the module form an attraction force. Since the delay in the interposer is larger than the wire delay with the same length in the single die, we consider the interposer in the attraction force in the following way. If the two positions of the module cross the interposer, we add an additional force where the force can be positive or negative. If the number of nets crossing interposer increases after moving the module to a new position, we will add a negative force, e.g., decrease the attraction force to avoid delay increase and vice versa. Fig. 5.3 shows an example for considering the interposer in the attraction force. We assume that Chip (die) 1 and Chip (die) 2 are two FPGA dies connected by interposer and 'M' represents a module. Currently we consider a module labelled with 'K' and the modules with color represent that they are used. P1 and P2 are the original position and new position for the module K, respectively. Fig. 5.3(a) shows one net crossing the interposer and Fig. 5.3(b) shows three nets crossing the interposer. Hence, we decrease the attraction force for moving K from P1 to P2 which may reduce the delay. Considering interposer in the force and cost function Eq. 5.4 of Section 5.3.3 can help to reduce the number of wires crossing the interposer.

Partition and Legalization

In each iteration of force-directed placement, we consider the interposer in the filling force and attraction force described above since the delay in the interposer is larger than the wire delay with the same length in a single die. Through considering interposer in the filling force it eases the partition stage. We divide the force-directed result into several parts according to their coordinates and chip boundary and each part represents a chip (die). After we assign the modules to the chips, we legalize (remove overlap) them using sequence pair [MFNK95a] which is a floorplan representation and powerful for maintaining the relative relation of modules.

Aspect Ratio Selection of Modules

Since ratio impacts the circuit delay, we consider it in our flow. For each iteration of force calculation, we pick one of the ratio from the ratio list of the module and update its width and height. Thus it may impact the density of the bin that the modules cover. Through this way the filling force may change and finally it impacts the placement results.

5.3.3 Hierarchical B*-tree Based Module Placement

Hierarchical B*-tree

B*-tree is an ordered binary tree structure proposed in modern floorplanning of ASIC designs [CC06]. Compared to other data structures, B*-Tree provides faster search and area estimation, convenient handling of constraints, linear time transformation between the tree and placement [CC06]. B*-tree representation has used for the single FPGA placement [CC06]. To support the multi-FPGA placement, we adopt the hierarchical B*-tree (HB*-tree) since it matches the multi-FPGA system well in which the multi-FPGA corresponds to a B*-tree and each chip corresponds to a sub B*-tree. Moreover, HB*-tree based placement can enable a fast search [CC06] and convergence of the solution.

The hierarchical B*-tree consists of two levels: interposer level and chip level. At interposer level, each chip is a node of a B*-tree and the nodes determine the relative position of each chip. At chip level, each module is a node of sub B*-tree and the nodes in this level

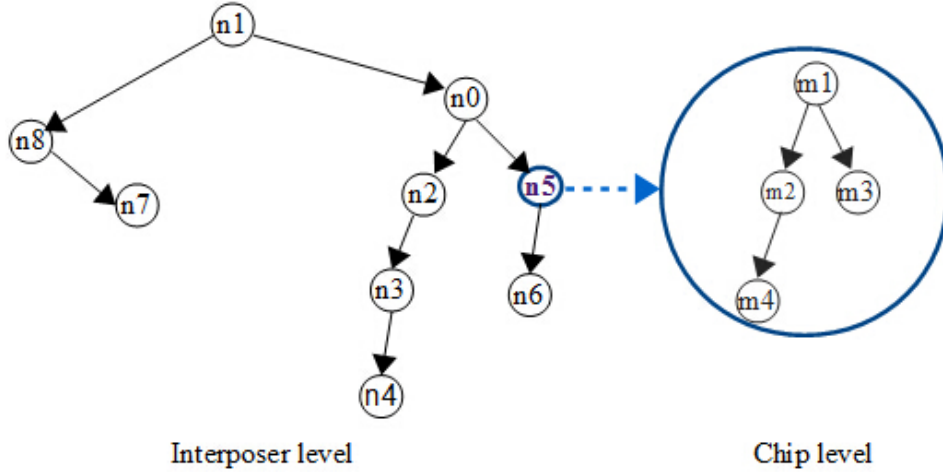


Figure 5.4: An Hierarchical B*-tree. In the interposer level, it has 9 chips. Chip n_5 has 4 modules in the chip level.

determine the placement of modules in each chip. Fig. 5.4 shows a structure of HB*-tree for illustration. At the interposer level, it has 9 chips and each chip is a node represented by n_i , $0 \leq i \leq 8$ respectively. At chip level, chip n_5 (one of the nodes in interposer level), contains 4 modules, and each module is a node represented by m_j , $1 \leq j \leq 4$ respectively.

Hierarchical B*-tree Operations

To explore the solution space, we apply the following operations.

At the interposer level, each chip is a node of the B*-tree and the following operations are performed on the nodes.

- OP1: Swap chips.
- OP2: Move a chip to another place.

At the chip level, each module is a node of the sub B*-tree and the following operations are performed on the nodes.

- OP3: Load a module with shape of $1/(\text{aspect ratio})$.
- OP4: Move a module to an another place.
- OP5: Swap two modules.
- OP6: Load a different aspect ratio of the module.

The following operations at the interposer level and chip level are performed at the same time.

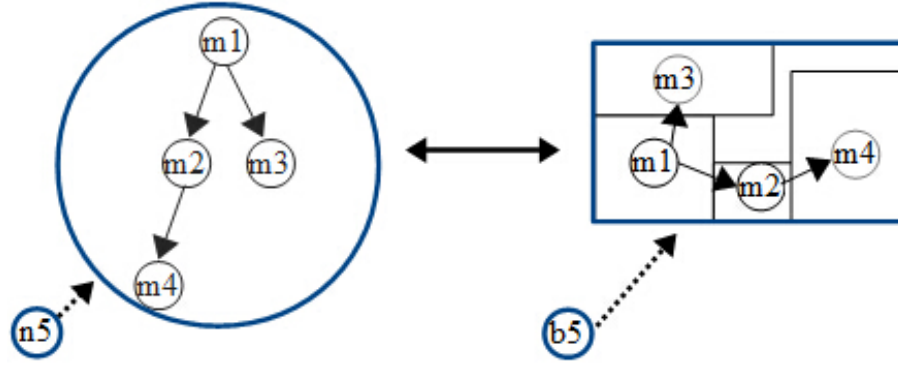


Figure 5.5: Chip-level packing: a sub-B*-tree of the chip n_5 and its corresponding placement.

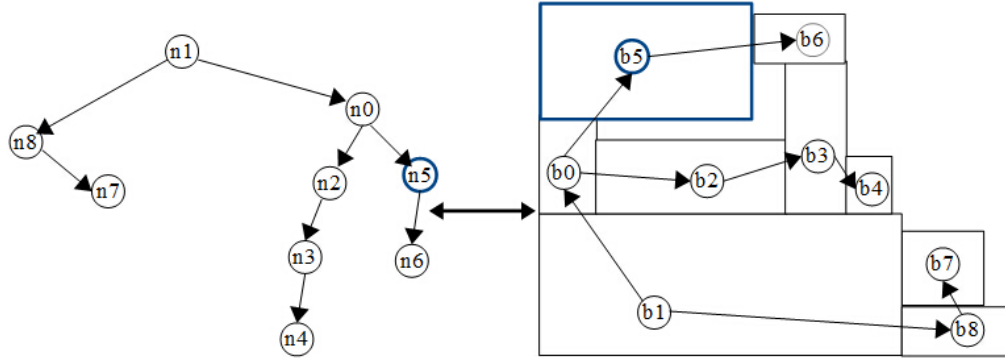


Figure 5.6: Interposer-level packing: a sub-B*-tree for chips and its corresponding placement

- OP7: Move a module in a chip to another chip.
- OP8: Swap two modules that are in different chips.

Here we use 'swap chips' (OP1) since all chips are the same and each node represent a chip in a tree of the interposer level. Swapping nodes in the tree is similar to swap chips.

The Packing of the Hierarchical B*-tree

We use two steps to pack the HB*-tree to a placement. Initially, we pack all the sub-B*-tree of the chip level one by one and get all the placement results for all the chips. Next, we pack the chip of interposer-level B*-tree to get a placement of the whole design. Finally, we map the placement of whole design using our proposed approach to the corresponding architecture.

A general example of a chip-level and interposer-level packing is shown in Fig. 5.5 and Fig. 5.6. Fig. 5.5 shows the sub-B*-tree of the chip n_5 and its corresponding placement

result. Fig. 5.6 shows that after packing the sub-B*-tree of each chip, we can pack the B*-tree at the interposer level, and each node of the B*-tree at the interposer level corresponds to the placement result of the chip. In our work, the dies are the same (same size: width and height).

Cost Metric in HB*-tree based Module Placement

To simplify the operation, we combine OP.3 and OP.6 together. We use τ which is area $\times$ delay of a shape of a module to evaluate the module generated by running VPR on the corresponding verilog file. The internal block information has already predefined, since VPR can achieve an efficient placement for the module. Simulated annealing is performed together with HB*-Tree operation to explore the solution space and the approach for choosing the temperature is the same as the [CC06]. In each iteration, the operations are randomly selected. Different weights are given to the operations to control the probability of choosing each operation. After each operation, the cost function [WW91] is calculated as shown in Eq. 5.4. The equation including wirelength and delay is used to evaluate each design and finally choose the best one based on the cost value. Here "Wirelength" and "Delay" represent the current total wirelength and total delay containing all nets of the design respectively. $Wirelength^*$ and $Delay^*$ represent the average wirelength and delay of the design respectively. Each FPGA (die) has the same size (width and height). We aim to propose a flow for multiple FPGAs with interposer and integrate the proposed routing model for interposer into the flow to reduce the delay. Thus, we give an fixed width and height for each FPGA (die) which is similar to the commercial multiple FPGAs with interposer [HPB14]. The size (width and height) of each FPGA die depends on each benchmark and the initial placement from the first stage.

$$Cost_{system} = \alpha \left(\frac{Wirelength}{Wirelength^*} \right) + (1 - \alpha) \frac{Delay}{Delay^*} \quad (5.4)$$

We divide the result by the average wirelength and delay to normalize the results. The α is a parameter to bias the cost function to wirelength or delay. If the net crosses the

interposer, we will use our wirelength estimation model to calculate the wirelength and also consider the delay in the interposer described in detail in Section 5.3.1. Thus the delay of each net includes the module delay and wirelength delay generated by Elmore delay [XGA⁺09]. We use the popular linear weighted function which is similar to [EH08] to bias the wirelength and total delay.

5.4 Experimental Results

To evaluate the performance of our proposed flow, we first create benchmarks based on the VPR benchmark suite [RLY⁺12] by decomposing each of the verilog file into multiple verilog files according to subfunctions, which are known as modules. Each module with one aspect ratio is obtained by running VPR on each verilog file. Information of the cases are shown in Table 5.1. The number of modules ranges from 110 to 320 for the current case set. We also 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. In the experiment, each module has 9 ratios 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.6, 2.5, 5.0 based on our evaluating and we set α to be 0.4. The multi-FPGA system is assumed to contain 6 chips (3 columns $\times$ 2 rows). All the experiments were performed on a IBM server x3650 with Intel Xeon(R) CPU and 42 GB DDR2 RAM.

Table 5.1: Benchmarks from VPR Suite

Casename	Modules	Tiles	Casename	Modules	Tiles
bgm128	128	33160	bgm224	224	58030
bgm320	320	82900	spree121	121	2453
spree209	209	4237	spree308	308	6244
stereovision2112	112	60944	stereovision2210	210	40010
stereovision2308	308	180488	stereovision1105	105	20005
stereovision1210	210	40010	stereovision1315	315	60015

Table 5.2: Comparison between Constant Delay Model used in [HPB14] and Our Proposed Model in Enhanced Force-Directed Placement

Casename	Traditional Model used in [HPB14]				Our Proposed Model			
	WL	Delay	CrossD	Time	WL	Delay	CrossD	Time
bgm128	22624	2140.28	145	9.49	21118	2052.59	104	10.63
bgm224	60676.5	4409.36	264	14.87	66748	4618.57	285	15.96
bgm320	110199.5	7072.01	421	45.26	105856.5	6936.38	420	48.67
spree121	10554.5	835	239	8.33	10416.5	827.73	236	8.57
spree209	23372	1575.85	387	39.15	24506.5	1617.02	393	39.74
spree308	48270	2810.62	630	132.3	48110	2775.66	600	135.97
stereovision1105	19941.5	1096.39	176	3.63	20353.5	1085.16	152	3.86
stereovision1210	59339	2892.92	449	16.09	54163	2672.46	389	16.7
stereovision1315	121681	5198.73	520	50.96	120186.5	5177.4	545	51.49
stereovision2112	61572	3035.49	296	4.86	62939	3042.87	261	4.83
stereovision2210	193964	8148.55	578	22.02	173945.5	7494.98	545	22.68
stereovision2308	330379.5	13498.35	972	68.55	342486.5	13877.66	976	69.94
Average	1	1	1	1	0.99	0.99	0.95	1.04

Table 5.3: Comparison between Constant Delay Model used in [HPB14] and Our Proposed Model in Our Proposed Flow

Casename	Traditional Model used in [HPB14]				Our Proposed Model			
	WL	Delay	CrossD	Time	WL	Delay	CrossD	Time
bgm128	14493	1849.22	106	26.24	11962	1722.75	58	28.27
bgm224	48274	4025.88	265	62.17	41004	3752.51	217	63.64
bgm320	77255	6028.74	399	144.58	77292	6022.88	392	126.04
spree121	11605	945.57	317	52.46	9706	775.7	206	53.4
spree209	28582	1935.36	585	152.58	25840	1702.35	437	158.78
spree308	57323	3391.27	930	344.77	52813	3083.46	762	331.17
stereovision1105	16639	990.01	172	64.93	16051	945.79	146	62.73
stereovision1210	63454	3116.48	545	207.44	56506	2772.1	416	204.56
stereovision1315	151125	6390.49	799	379.34	136079	5819.06	694	362.14
stereovision2112	37709	2215.74	216	58.3	33657	2017.12	143	58.59
stereovision2210	133026	6239.48	558	196.85	140514	6458.6	545	189.62
stereovision2308	236712	10409.65	787	351.94	247117	10784.21	839	313.57
Average	1	1	1	1	0.92	0.93	0.81	0.98

5.4.1 Impact of Interposer

In the first experiment, we evaluate the interposer impact using proposed routing model to the force-directed placement for the initial placement and to the whole flow for the finally refined placement. In previous work, the constant delay, e.g., $1ns$ has been used for interposer [HPB14]. Here we compare our results with the constant delay model used in [HPB14] for interposer in our flow to show the impact of the detailed routing model. The results are listed in Table 5.2 and Table 5.3, respectively. In the table, **WL**,

Table 5.4: Comparison between Modules with only one ratio and different ratios in Enhanced Force-directed Placement

Casename	Modules with Only One Ratio				Modules with Multiple Ratios			
	WL	Delay	CrossD	Time	WL	Delay	CrossD	Time
bgm128	21118	2052.59	104	10.63	17143.5	1933.66	97	12.41
bgm224	66748	4618.57	285	15.96	43157	3798.16	174	27.21
bgm320	105856.5	6936.38	420	48.67	72906.5	5847.29	324	74.55
spree121	10416.5	827.73	236	8.57	10399.5	832.27	233	8.67
spree209	24506.5	1617.02	393	39.74	24960	1662.51	413	40.07
spree308	48110	2775.66	600	135.97	48810	2852.7	640	134.77
stereovision1105	20353.5	1085.16	152	3.86	17306.5	991.03	141	4.05
stereovision1210	54163	2672.46	389	16.7	52014	2594.46	359	16.74
stereovision1315	120186.5	5177.4	545	51.49	111136	4841.78	460	52.42
stereovision2112	62939	3042.87	261	4.83	42946	2419.61	241	3.06
stereovision2210	173945.5	7494.98	545	22.68	116621.5	5649.92	446	16.04
stereovision2308	342486.5	13877.66	976	69.94	216465.5	9700.96	658	73.41
Average	1	1	1	1	0.82	0.89	0.88	1.07

Delay, CrossD, Time denote the total wirelength (tile), total delay (ns), the delay generated by the nets crossing the interposer (ns) and the runtime (s), respectively. The results in Table 5.2 show that our proposed model does not impact the force-directed placement much. The wirelength, total delay and delay from nets crossing the interposer are reduced by 1%, 1%, 5% respectively, for the initial placement step. The time increases by 4%. However, the proposed model shows great impact on the final placement results for our flow. Table 5.3 shows that our proposed model reduces the wirelength, total delay, delay from nets crossing the interposer and runtime by 8%, 7%, 19%, 2% respectively. Hence these results demonstrate that a detailed routing model for interposer can help to improve the placement quality.

5.4.2 Impact of Different Module Ratios

We compare the results between modules with only one ratio and the modules with different ratios in the placement flow to show the impact of module ratios. Similarly, we evaluate the impact of different module ratios on the force-directed placement and the whole flow which usually uses one ratio in the previous work. The corresponding results are listed in Table 5.4 and Table 5.5. In the tables, '1' means that the data is

Table 5.5: Comparison between Modules with **only one ratio** and **different ratios** in **Our Proposed Flow**

Casename	Modules with Only One Ratio				Modules with Multiple Ratios			
	WL	Delay	CrossD	Time	WL	Delay	CrossD	Time
bgm128	13015	1773.4	76	32.1	11205	1713.57	61	42.35
bgm224	47655	3994.69	253	69.22	39079	3686.74	189	101.59
bgm320	80676	6092.79	357	126.96	70302	5792.55	350	216.36
spree121	9112	739.29	188	56.87	8368	704.29	168	79.49
spree209	26101	1724.45	451	160.23	27454	1788.82	462	252.81
spree308	57054	3286.93	834	368.82	52148	3022.18	706	555.97
stereovision1105	14872	891.24	128	61.38	14386	892.49	133	98.11
stereovision1210	59430	2893.74	447	225.12	47977	2421.32	311	342.94
stereovision1315	130223	5576.53	633	440.75	122744	5379.63	638	581.6
stereovision2112	39531	2265.22	209	65.54	30941	1965.45	159	86.8
stereovision2210	121576	5806.53	480	212.96	134006	6292.84	550	292.07
stereovision2308	238414	10487.42	812	377.91	229232	10208.72	770	548.46
Average	1	1	1	1	0.92	0.96	0.91	1.46

the baseline. For example, if A is the baseline for a case and B is the compared data for the same case, we label A with '1' and use $(B-A)/A$ to get the rate of change for the case. Table 5.4 shows that when considering different ratios of modules in force-directed placement, the wirelength, total delay and delay from nets crossing interposer are significantly improved by **18%**, **11%**, **12%** respectively with an extra 7% runtime. The results in Table 5.5 indicate that when considering different ratios of modules in the whole flow, the wirelength, total delay and delay from nets crossing interposer are improved by **8%**, **4%**, **9%** respectively and the runtime increase 46%. We can see that considering different module ratios in the force-directed placement can significantly optimize the total delay of the circuits and improve the placement quality. However, in the HB*-tree based solution refinement, the benefits from module ratios is reduced. It is because that when we consider modules with multiple ratios in the SA, the design space is enlarged significantly and within a comparable time, only suboptimal placement can be reached by SA. When we consider multiple module ratios, it will not increase its running time much since force-directed placement is based on an analytical solution.

Table 5.6: Comparison between Min-cut + SA and Our Proposed approach

Casename	Min-cut + SA				Our Approach			
	WL	Delay	CrossD	Time	WL	Delay	CrossD	Time
bgm128	15599	1871.5	94	33.84	11962	1722.75	58	28.27
bgm224	58225	4360.36	291	75.12	41004	3752.51	217	63.64
bgm320	93559	6610.16	475	132.66	77292	6022.88	392	126.04
spree121	12794	960.43	295	64.8	9706	775.7	206	53.4
spree209	33301	2069.65	573	195.18	25840	1702.35	437	158.78
spree308	66616	3601.35	852	386.76	52813	3083.46	762	331.17
stereovision1105	23122	1234.99	216	78.96	16051	945.79	146	62.73
stereovision1210	90325	3970.49	566	236.1	56506	2772.1	416	204.56
stereovision1315	214682	8442.76	881	465.18	136079	5819.06	694	362.14
stereovision2112	38384	2221.66	201	67.2	33657	2017.12	143	58.59
stereovision2210	151469	6793.21	540	215.94	140514	6458.6	545	189.62
stereovision2308	215396	9542.86	581	345.54	247117	10784.21	839	313.57
Average	1	1	1	1	0.79	0.86	0.83	0.85

5.4.3 Speed Up of the Placement Flow

We compare our flow with the traditional flow (min-cut + SA) (module-level) [LMW⁺13, KAKS97]. We use hMETIS [KAKS97] for partitioning for its high quality and fast speed. Hmetis [KAKS97] is a software package for partitioning large hypergraphs and is based on min-cut. Its goal is to minimize the total cut weight. The weights of hyperedges and vertices are two parameters in hMETIS. Here the hyperedges and vertices are set to the same weight. The results are shown in the Table 5.6 and it shows that our approach improves the wirelength, total delay and the delay from nets crossing interposer by **21%**, **14%**, **17%** respectively in a shorter time (improve by **15%**). The benefits are achieved from the following two parts. On one hand, min-cut is optimized to minimize the cut size. Although it also helps on reduction of the delay, our force-directed placement can achieve similar or better initial placement due to its optimization for delay and consideration of the detailed interposer model. On the other hand, HB*-tree based SA can search and converge much faster than SA which enables our flow to find a more efficient solution than SA within the same time.

5.5 Summary

In this chapter we proposed a module based mapping flow to address the placement for multi-FPGA systems with interposer. A routing model is developed to more accurately model the wirelength and delay of the interposer. The proposed mapping flow contains two stages. Firstly, we adopted the force-directed method to get a good-quality initial solution as a starting point of the refinement step. Secondly, we adopted the simulated annealing (SA) to efficiently search the solution space and find the approximately optimal placement solution. In order to speed up the refinement of placement in SA, we employed the HB*-tree representation to enable a fast search and convergence. The mapping flow supported the modules with different ratios which impact the total delay of the circuit to achieve the delay optimization. Our proposed approach can obtain efficient result in reasonable time and can scale well for different design sizes.

CHAPTER 6

Conclusions and Future Work

In this chapter, we give concluding remarks on our study in section 6.1 and discuss several interesting research directions for our future research in section 6.2.

6.1 Conclusions

In Field-Programmable Gate Arrays (FPGAs) partial reconfiguration (PR) design flow, many challenging research problems remain, e.g., how to efficiently place the reconfigurable modules and how to properly manage the reconfigurable modules and static modules. Although there are commercial partial reconfiguration (PR) computer-aided design (CAD) tools for FPGAs, their mapping steps still mainly rely on manual efforts. For academic CAD tools, little work has considered module based placement and routing with PR support. The commonly used placer and router, *e.g.* VPR, is based on fine-grained tiles. Still there is a missing link between the modern CAD tools and the PR applications. Efficient CAD tools of placement and routing which support the partial reconfiguration on FPGAs are urgently needed. This research attempts to fill this gap and proposes an automatic mapping flow for efficient PR.

In Chapter 3, we propose 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 delay from the VPR tool 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.

In Chapter 4, we propose a new dynamic module partitioning approach for library based design which improves the flexibility of modules and try to reduce the waste area inside and between modules. Our approach provides an opportunity to address the fixed-outline placement which is an unsolved problem in module based placement flow. The experiment results show that our approach is efficient and can reduce the area by up to 10% with marginal increase in delay and acceptable runtime. The flow reduces the significant complexity and efforts for large number of module preparations with different ratios and targeting different architectures.

In Chapter 5, we present our work involving placement for the interposer based multi-FPGA system. We develop a wirelength estimation model for the interposer to allow a more accurate evaluation of the interposer impact on the placement flow and propose a hierarchical B*-tree (HB*-tree) based modular placer for the interposer-based multi-FPGA system for its efficiency and effectiveness in finding the efficient placement. The results show that 1) Interposer impacts the placement result significantly and compared with the traditional model, our proposed model can reduce the wirelength, total delay, delay from nets crossing the interposer by 8%, 7%, 19%, respectively; 2) The ratios of the module impact the placement results. When considering the different ratios of modules, the wirelength, total delay and delay from nets crossing interposer are improved by 8%, 4%, 9% compared to the results considering modules with only one ratio; 3) Our proposed flow can speed up the placement and achieve higher quality compared to the state-of-the-art min-cut based flow derived from [KAKS97].

We believe the automatic tools will bridge the gap between the academic research and industry.

6.2 Future Work

Besides the problems that we addressed above, there are many other factors that affect the performance of the flow for partial reconfiguration on FPGAs. In this section, we discuss some promising directions for our further research in future.

6.2.1 Further Performance Optimization

Reducing Power

PR on FPGA involves many factors about performance. One of the most important things is to reduce overhead. The runtime overhead includes several aspects. Firstly, we should manage the reconfigurable resource well and define where to place the static modules and partial reconfigurable modules. Secondly, we should manage the reconfiguration time. Different task schedules for partial reconfiguration region affect the performance much. Thirdly, we should manage the unused resources that are not used as a fragmentation. If we make use of the unused resource, we may reduce the size of the chip and reduce the cost. We also need to take the power into consideration. The power is dominated by static power and dynamic power. We plan to reduce power by scheduling the tasks based on their difference in power/energy. For example, we can swap out the power-hungry task out when it is not needed at the moment to reduce the power overhead [WAT14].

Exploring Different Floorplan Representations

We will explore the use of different floorplan representations based on our proposed flow for the interposer based multi-FPGA system. In our work described in Chapter 3, we adopt the B*-tree representation. There are a few other floorplan representations for different types of floorplans. There are totally two types of floorplans: slicing and non-slicing [CC06]. A slicing structure indicates that it can be obtained by recursively cutting rectangles in horizontal direction or vertical direction. Otherwise floorplan is a non-slicing structure. The authors in [Ott82] propose a binary-tree to represent the slicing

floorplans. The work in [WL86] propose a normalized polish express to represent a slicing floorplan. However, most of the real designs are non-slicing structure. Several non-slicing representation are proposed, i.e, the wheel structure [PL95], sequence pair [MFNK95b], bounded sliceline grid [NFMK96], O-tree [GCY99], B*-tree [CC06], and corner block list [HHC⁺00], TCG [LC05]. Different representations may have their own advantages. Thus, we plan to investigate them for the PR problem and explore their metrics for different requirements.

Considering Congestion Model and Constraints

Partial reconfiguration has several advantages. Firstly, it can improve system flexibility by implementing more functions while other parts are still running. Secondly, it can reduce size and cost by using a smaller FPGA to time-multiplex the hardware. Thirdly, it can reduce power by swapping in/out different tasks, i.e, when a power-hungry task is not needed, we can swap out this task [dB10]. Thus, we need to implement the context switching to support these metrics mentioned above. We plan to investigate PR-aware routing and the context switching in the PR region for meeting the requirement of different applications. In order to avoid signal interruption incurred by PR, a constraint is involved that nets that do not belong to one reconfigurable region cannot go through the reconfigurable region which makes the routing more difficult. The constraint is prone to generate congestion between different regions and costs much more routing time. The routing congestion can make the design constraint fail or generate low performance design. We can mainly divide the congestion model into three types: empirical model [IEC94, WYS00], global router-based model [PBS98, WS00] and probability analysis model [LHJX06, MMX⁺09]. Empirical model depends much on the design-specific parameters. Global router-based model is not accurate and cost too much runtime. Probability model can be an ideal model for placement optimization. However, the previous studies and models do not consider the congestion status in the PR-aware routing and the interposer which impacts the final placement result much. To address these challenges, we plan to propose a routing algorithm and develop a congestion model to take the congestion into consideration and reduce the runtime for the single FPGA. If we consider the multiple FPGAs with

interposer, we may consider the interposer in the congestion routing model since the interposer has limited routing resource for connecting different FPGAs (dies).

6.2.2 Extension of Current Work

Extension to 3D FPGAs

Recently, three dimensional (3D) FPGAs have shown significant performance improvements compared to the 2D FPGAs. Especially it allows to create a larger chip than a single die and it has low defect density compacted to an extremely large single one [KWK⁺11]. Three dimension integrated technologies provides several advantages. Firstly, it can reduce wirelength which can help to reduce the power consumption and can also improve the performance. Secondly, it can improve the bandwidth since it can provide other resources besides the IO resources. Thirdly, it can support the heterogeneous integration, which can offer more flexibility for architecture designs and applications. Fourthly, it can increase density due to the stacked silicon integration technology and the technology can offer potentially lower cost for the designs [SSPS07, LEGLW06, CM10]. 3D FPGA has through-silicon vias (TSVs) which is a different characteristic compared to the 2D/2.5D FPGA. The TSVs are used for connecting different layers of 3D FPGA and usually the number of TSVs is small due to its high expense and the limited process level. Previous work for 3D FPGAs only considers CLB-level placement and it cannot be used for PR. Our work supports PR and mainly focus on the 2D/2.5D FPGA, but we can take the the structure of 3D FPGAs into consideration, e.g, considering the TSVs in the wirelength estimation model. We will extend our work to 3D FPGAs for supporting PR.

Considering Machine learning for FPGA Placement

Machine learning is the study of computer algorithms that can learn from data and experience [YPMZ15]. Supervised learning is one type of machine learning and it is widely used in physical design [YPMZ15, LJ98, MSM⁺05]. It mainly consists of training and testing stages [YPMZ15]. The authors in [LJ98] adopt machine learning to estimate the

layout shape functions of full-custom integrated circuit cells. They use a neural network to estimate one dimension of cell layout from some properties of the cell. The authors in [MSM⁺05] use support vector machines (SVMs) to perform placement. The authors in [YPMZ15] describe key techniques and results of machine learning and introduce its application in physical design. For library based design, it is difficult and impractical to collect the implementations of a logic with all possible aspect ratios in the library. Machine learning can learn from the data in the library. Each module has some ratios and the information about delay, area and placement result under each ratio of each module are already stored in the library. We can use the information in the library as the training data for each module. We can then predict the delay information for ratios which do not exist in the library for each module during placement. Thus it provides a good chance to get a high quality-of-result prediction of the delay and possible placement for the logic with a aspect ratio that does not exist in the library. It helps us to conveniently explore better solutions.

List of publications

- Journal Publications:

- **Fubing Mao**, Yi-chung Chen, Wei Zhang, Hai (Helen) Li and Bingsheng He. Library-based Placement and Routing in FPGAs with Support of Partial Reconfiguration. ACM Transactions on Design Automation of Electronic Systems. vol. 21, no. 4, pp.71:1-71:26, May 2016. (TODAES 2016).

- Conference Publications:

- Weichen Liu, Wei Zhang, **Fubing Mao**. An Efficient Soft Error Protection Scheme for MPSoC and FPGA-based Verification. IEEE International Conference on Anti-Counterfeiting, Security and Identification.Taiwan, August 24-26, 2012.(ASID 2012)
- **Fubing Mao**, Wei Zhang, and Bingsheng He. Towards Automatic Partial Reconfiguration in FPGAs. IEEE International Conference on Field-Programmable Technology, Dec 10-12, 2014. (FPT2014).
- **Fubing Mao**, Wei Zhang, Bo Feng, Bingsheng He and Yuchun Ma. Modular Placement for Interposer based Multi-FPGA Systems.USA, May 18-20, 2016. (GLSVLSI 2016).
- **Fubing Mao**, Wei Zhang, Bingsheng He and Siew-Kei Lam. Dynamic Module Partitioning for Library based Placement on Heterogeneous FPGAs. 2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications. Taiwan Aug 16-18, 2017. (RTCSA2017).

- Posters and Demos:

- **Fubing Mao**, Y.-C. Chen, Wei Zhang, and H. Li. BMP: A Fast B*-Tree based Modular Placer for FPGAs, ACM/SIGDA International Symposium

- on Field-Programmable Gate Arrays. February 26-28, 2014. (FPGA 2014). (Poster).
- **Fubing Mao**, Wei Zhang, Bingsheng He, and Siew-Kei Lam. Dynamic Partitioning for Library based Placement on Heterogeneous FPGAs, ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. USA, February 22-24, 2017. pp. 296-296.(FPGA 2017). (Poster).
 - **Fubing Mao**, Wei Zhang, Bingsheng He and Siew-Kei Lam. Dynamic Module Partitioning for Library based Placement on Heterogeneous FPGAs. USA, April 30- May 2, 2017. (FCCM2017). (Poster).
 - **Fubing Mao**, Wei Zhang and Bingsheng He. BMPR: B*-tree based Modular Placer & PR-aware Router. IEEE International Symposium on Field-Programmable Custom Computing Machines. May 3-5, 2015. (FCCM2015 Demo).

References

- [ABD⁺07] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf. Wires on demand: Run-time communication synthesis for reconfigurable computing. In *2007 International Conference on Field Programmable Logic and Applications*, pages 513–516, Aug 2007.
- [Aga15] S. D. Agashe. Introduction to FPGA and Verilog Programming, 2015. <http://coep.vlab.co.in/index.php>.
- [AGBD07] S. Areibi, G. Grewal, D. Banerji, and P. Du. Hierarchical fpga placement. *Electrical and Computer Engineering, Canadian Journal of*, 32(1):53–64, Winter 2007.
- [Alt10] Altera. Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs, 2010. <http://www.altera.com>.
- [AMB06] C. Ababei, H. Mogal, and K. Bazargan. Three-dimensional place and route for fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1132–1140, June 2006.
- [BH86] J. E. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324:446, 1986.
- [BKL10] T. Becker, M. Koester, and W. Luk. Automated placement of reconfigurable regions for relocatable modules. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 3341–3344, May 2010.
- [BKS00] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. *IEEE Design & Test of Computers*, 17(1):68–83, 2000.

- [BKT13] Christian Beckhoff, Dirk Koch, and Jim Torreson. Automatic Floorplan-
ning and Interface Synthesis of Island Style Reconfigurable Systems with
GOAHEAD. In *Architecture of Computing Systems (ARCS)*, pages 303–316.
Springer, 2013.
- [BMS11] C. Bolchini, A. Miele, and C. Sandionigi. Automated resource-aware floor-
planning of reconfigurable areas in partially-reconfigurable fpga systems.
In *Field Programmable Logic and Applications (FPL), 2011 International
Conference on*, pages 532–538, Sept 2011.
- [BR97] Vaughn Betz and Jonathan Rose. Vpr: a new packing, placement and
routing tool for fpga research. In Wayne Luk, PeterY.K. Cheung, and
Manfred Glesner, editors, *Field-Programmable Logic and Applications*,
volume 1304 of *Lecture Notes in Computer Science*, pages 213–222. Springer
Berlin Heidelberg, 1997.
- [BSKB09a] P. Banerjee, S. Sur-Kolay, and A. Bishnu. Fast unified floorplan topology
generation and sizing on heterogeneous fpgas. *Computer-Aided Design of
Integrated Circuits and Systems, IEEE Transactions on*, 28(5):651–661,
May 2009.
- [BSKB09b] P. Banerjee, S. Sur-Kolay, and A. Bishnu. Fast unified floorplan topology
generation and sizing on heterogeneous fpgas. *IEEE Transactions on
Computer-Aided Design of Integrated Circuits and Systems*, 28(5):651–661,
May 2009.
- [BSSK11] Pritha Banerjee, Megha Sangtani, and Susmita Sur-Kolay. Floorplanning
for Partially Reconfigurable FPGAs. *IEEE Transactions on Computer-
Aided Design of Integrated Circuits and Systems (TCAD)*, 30(1):8–17,
2011.
- [CC04] Gang Chen and Jason Cong. Simultaneous placement with clustering and
duplication. *ACM Trans. Des. Autom. Electron. Syst.*, 11(3):740–772, June
2004.

- [CC06] T.C. Chen and Y.W. Chang. Modern Floorplanning Based on B*-Tree and Fast Simulated Annealing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(4):637–650, 2006.
- [CCBH07] Zhang C. Claus and M B. Hubner. An xdl-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems. In *Works on Reconfigurable Computing Education*, 2007.
- [CCC14] Y. C. Chen, S. Y. Chen, and Y. W. Chang. Efficient and effective packing and analytical placement for large-scale heterogeneous fpgas. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 647–654, Nov 2014.
- [CCP06] Deming Chen, Jason Cong, and Peichen Pan. Fpga design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(3):195 – 330, 2006.
- [CJA08] S. S. Sapatnekar C. J. Alpert, D. P. Mehta. Fpga technology mapping, placement, and routing. In *Handbook of Algorithms for Physical Design Automation*, Auerbach Publications, 2008.
- [CL10] Jason Cong and Guojie Luo. An analytical placer for mixed-size 3d placement. In *Proceedings of the 19th International Symposium on Physical Design*, ISPD '10, pages 61–66, New York, NY, USA, 2010. ACM.
- [ClFGvRv06] R. Castro-lópez, F.V. Fernández, O. Guerra-vinuesa, and A. Rodríguez-vázquez. A reuse-based design framework for analog ics. In *Reuse-Based Methodologies and Tools in the Design of Analog and Mixed-Signal Integrated Circuits*, pages 27–62. Springer Netherlands, 2006.
- [CM10] Jason Cong and Yuchun Ma. Thermal-aware 3d floorplan. In Yuan Xie, Jason Cong, and Sachin Sapatnekar, editors, *Three Dimensional Integrated Circuit Design*, Integrated Circuits and Systems, pages 63–102. Springer US, 2010.

- [CPF09] Jeffrey M Carver, Richard Neil Pittman, and Alessandro Forin. Automatic Bus Macro Placement for Partially Reconfigurable FPGA Designs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 269–272, 2009.
- [CSRB11] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable Domain-specific Computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2011.
- [CW04] Lei Cheng and M. D. F. Wong. Floorplan design for multi-million gate fpgas. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 292–299, Nov 2004.
- [CW06] Lei Cheng and Martin DF Wong. Floorplan Design for Multimillion Gate FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(12):2795–2805, 2006.
- [dB10] Frank de Bont. In and Outs of Partial Reconfiguration with Xilinx FPGAs, 2010. http://www2.fhi.nl/de/archief/2010/images/corevision_partial_reconfiguration_f.de_bont.pdf.
- [Don80] W.E. Donath. Complexity theory and design automation. In *Design Automation, 1980. 17th Conference on*, pages 412–419, June 1980.
- [EH08] K. Eguro and S. Hauck. Enhancing timing-driven fpga placement for pipelined netlists. In *2008 45th ACM/IEEE Design Automation Conference*, pages 34–37, June 2008.
- [FMM12] Umer Farooq, Zied Marrakchi, and Habib Mehrez. Fpga architectures: An overview. In *Tree-based Heterogeneous FPGA Architectures*, pages 7–48. Springer New York, 2012.
- [GA12] M. Gort and J.H. Anderson. Analytical placement for heterogeneous fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 143–150, Aug 2012.

- [GA14] M. Gort and J. Anderson. Design re-use for compile time reduction in fpga high-level synthesis flows. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 4–11, Dec 2014.
- [Gaj99] D. D. Gajski. Ip-based design methodology. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 43–, June 1999.
- [GCY99] Pei-Ning Guo, Chung-Kuan Cheng, and T. Yoshimura. An o-tree representation of non-slicing floorplan and its applications. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 268–273, 1999.
- [GSJ10] F. Gharibian, L. Shannon, and P. Jamieson. Finding system-level information and analyzing its correlation to fpga placement. In *2010 International Conference on Field Programmable Logic and Applications*, pages 544–549, Aug 2010.
- [HC13] Yuan-Kai Ho and Yao-Wen Chang. Multiple chip planning for chip-interposer codesign. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–6, May 2013.
- [HGS05] A. Hekmatpour, K. Goodnow, and Hemen Shah. Standards-compliant ip-based asic and soc design. In *Proceedings 2005 IEEE International SOC Conference*, pages 322–323, Sept 2005.
- [HHC⁺00] Xianlong Hong, Gang Huang, Yici Cai, Jiangchun Gu, Sheqin Dong, Chung-Kuan Cheng, and Jun Gu. Corner block list: an effective and efficient topological representation of non-slicing floorplan. In *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, pages 8–12, Nov 2000.
- [HLM⁺12] Ruining He, Guoqiang Liang, Yuchun Ma, Yu Wang, and Jinian Bian. PDPR: Fine-Grained Placement for Dynamic Partially Reconfigurable FPGAs. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 350–356. Springer, 2012.

- [HMS04] Bo Hu and M. Marek-Sadowska. Fine granularity clustering-based placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):527–536, April 2004.
- [HPB14] Andre Hahn Pereira and Vaughn Betz. Cad and routing architecture for interposer-based multi-fpga systems. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 75–84, New York, NY, USA, 2014. ACM.
- [Hu06] Bo Hu. Timing-driven placement for heterogeneous field programmable gate array. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 383–388, Nov 2006.
- [HW95] BradL. Hutchings and MichaelJ. Wirthlin. Implementation approaches for reconfigurable logic applications. In *Field-Programmable Logic and Applications*, volume 975, pages 419–428. Springer Berlin Heidelberg, 1995.
- [HWY⁺09] E. Hung, S. J. E. Wilton, H. Yu, T. C. P. Chau, and P. H. W. Leong. A detailed delay path model for fpgas. In *2009 International Conference on Field-Programmable Technology*, pages 96–103, Dec 2009.
- [Inc11] Xilinx Inc. Early Access PR User Guide, 2011. <http://www.xilinx.com>.
- [Inc12] Xilinx Inc. White Paper: Vivado Design Suite, 2012. <http://www.xilinx.com>.
- [Inc13] Xilinx Inc. 7 Series FPGA Overview, 2013. <http://www.xilinx.com>.
- [Jay01] Rajeev Jayaraman. Physical design for fpgas. In *Proceedings of the 2001 International Symposium on Physical Design*, ISPD '01, pages 214–221, New York, NY, USA, 2001. ACM.
- [JGS13] P. Jamieson, F. Gharibian, and L. Shannon. Supergenes in a genetic algorithm for heterogeneous fpga placement. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 253–260, June 2013.

- [KAKS97] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multi-level hypergraph partitioning: Application in vlsi domain. In *Proceedings of the 34th Annual Design Automation Conference, DAC '97*, pages 526–529. ACM, 1997.
- [KBT08] D. Koch, C. Beckhoff, and J. Teich. Recobus-builder - a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *2008 International Conference on Field Programmable Logic and Applications*, pages 119–124, Sept 2008.
- [KBT10] Dirk Koch, Christian Beckhoff, and Jim Torresen. Zero logic overhead integration of partially reconfigurable modules. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design, SBCCI '10*, pages 103–108, New York, NY, USA, 2010. ACM.
- [KM00] Andrew A Kennings and Igor L Markov. Analytical Minimization of Half-perimeter Wirelength. In *ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 179–184, 2000.
- [KSY95] Chunghee Kim, Hyunchul Shin, and Younguk Yu. Performance-driven circuit partitioning for prototyping by using multiple fpga chips. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, pages 113–118, Aug 1995.
- [KT10] Dirk Koch and Jim Torresen. Advances and Trends in Dynamic Partial Run-time Reconfiguration, 2010. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.231.5217&rep=rep1&type=pdf>.
- [KTR08] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008.

- [KWK⁺11] Namhoon Kim, D. Wu, Dongwook Kim, A. Rahman, and P. Wu. Interposer design optimization for high frequency signal transmission in passive and active interposer using through silicon via (tsv). In *Electronic Components and Technology Conference (ECTC), 2011 IEEE 61st*, pages 1160–1167, May 2011.
- [LC05] Jai-Ming Lin and Yao-Wen Chang. Tcg: A transitive closure graph-based representation for general floorplans. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(2):288–292, Feb 2005.
- [LCW14] Wen-Hao Liu, Min-Sheng Chang, and Ting-Chi Wang. Floorplanning and signal assignment for silicon interposer-based 3d ics. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6, June 2014.
- [LCY11] Nan Liu, Song Chen, and T. Yoshimura. Floorplanning for high utilization of heterogeneous fpgas. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1–6, March 2011.
- [IEC94] Chih liang Eric Cheng. Risa: Accurate and efficient placement routability modeling. In *Computer-Aided Design, 1994., IEEE/ACM International Conference on*, pages 690–695, Nov 1994.
- [LEGLW06] Mingjie Lin, Abbas El Gamal, Yi-Chang Lu, and Simon Wong. Performance benefits of monolithically stacked 3d-fpga. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays, FPGA '06*, pages 113–122, New York, NY, USA, 2006. ACM.
- [LFy09] Wang Lie and Wu Feng-yan. Dynamic partial reconfiguration in fpgas. In *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, volume 2, pages 445–448, Nov 2009.
- [LGW⁺14] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn

- Betz. Vtr 7.0: Next generation architecture and cad system for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2):6:1–6:30, July 2014.
- [LHJX06] Shenghua Liu, Xianlong Hong, Tong Jing, and Jingyu Xu. Free: A fast routability estimator. In *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, volume 4, pages 2454–2458, June 2006.
- [LJ98] Xiao Quan Li and M. A. Jabri. Machine learning-based vlsi cells shape function estimation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(7):613–623, Jul 1998.
- [LMW⁺13] Qingyu Liu, Yuchun Ma, Yu Wang, W. Luk, and Jinian Bian. Ralp: Reconvergence-aware layer partitioning for 3d fpgas. In *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, pages 1–6, Dec 2013.
- [LPL⁺11] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Hmflow: Accelerating fpga compilation with hard macros for rapid prototyping. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 117–124, May 2011.
- [LR08] Sang-Joon Lee and K. Raahemifar. Fpga placement optimization methodology survey. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 001981–001986, May 2008.
- [MC16] Fubing Mao and Yi-Chung etc Chen. Module based placer and router, 2016. <https://github.com/Frankfbmao/BMPR>.
- [MCZ⁺16] Fubing Mao, Yi-Chung Chen, Wei Zhang, Hai (Helen) Li, and Bingsheng He. Library-based placement and routing in fpgas with support of partial reconfiguration. *ACM Trans. Des. Autom. Electron. Syst.*, 21(4):71:1–71:26, May 2016.

- [MFNK95a] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 472–479, Nov 1995.
- [MFNK95b] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, pages 472–479, Nov 1995.
- [Min07] Minkovich K. MCNC benchmarks, 2007. <http://cadlab.cs.ucla.edu/~kirill/>.
- [MM11] T. Marconi and T. Mitra. A novel online hardware task scheduling and placement algorithm for 3d partially reconfigurable fpgas. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–6, Dec 2011.
- [MMX⁺09] Fubing Mao, Yuchun Ma, Ning Xu, Shenghua Liu, Yu Wang, and Xianlong Hong. Congestion-driven floorplanning based on two-stage optimization. In *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, pages 1298–1301, Oct 2009.
- [MSCL07] T.S.T. Mak, P. Sedcole, P.Y.K. Cheung, and W. Luk. Average interconnection delay estimation for on-fpga communication links. *Electronics Letters*, 43(17):918–920, August 2007.
- [MSM⁺05] R. Manimegalai, E. Siva Soumya, V. Muralidharan, B. Ravindran, V. Kamakoti, and D. Bhatia. Placement and routing for 3d-fpgas using reinforcement learning and support vector machines. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 451–456, Jan 2005.
- [MSSM10] Alessio Montone, Marco D Santambrogio, Donatella Sciuto, and Seda O-grenci Memik. Placement and Floorplanning in Dynamically Reconfigurable FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, 3(4):24, 2010.

- [MTB00] Fan Mo, A. Tabbara, and R. K. Brayton. A force-directed macro-cell placer. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD - 2000. IEEE/ACM Digest of Technical Papers (Cat. No.00CH37140)*, pages 177–180, Nov 2000.
- [NFMK96] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module placement on bsg-structure and ic layout applications. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 484–491, Nov 1996.
- [NHCB02] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Accurate area and delay estimators for fpgas. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 862–869, 2002.
- [OJ04] B. Obermeier and F.M. Johannes. Temperature-aware global placement. In *Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific*, pages 143–148, Jan 2004.
- [Ott82] R.H.J.M. Otten. Automatic floorplan design. In *Design Automation, 1982. 19th Conference on*, pages 261–267, June 1982.
- [PBS98] P.N. Parakh, R.B. Brown, and K.A. Sakallah. Congestion driven quadratic placement. In *Design Automation Conference, 1998. Proceedings*, pages 275–278, June 1998.
- [PL95] Peichen Pan and C.L. Liu. Area minimization for floorplans. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(1):123–132, Jan 1995.
- [RDM⁺17] M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis, and M. D. Santambrogio. Floorplanning automation for partial-reconfigurable fpgas via feasible placements generation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(1):151–164, Jan 2017.
- [RLY⁺12] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson.

- The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 77–86, 2012.
- [RNS95] K. Roy-Neogi and C. Sechen. Multiple fpga partitioning with performance optimization. In *Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on*, pages 146–152, 1995.
- [SB06] Love Singhal and Elahieh Bozorgzadeh. Multi-layer floorplanning on a sequence of reconfigurable designs. In *IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2006.
- [SB07] L. Singhal and E. Bozorgzadeh. Novel multi-layer floorplanning for heterogeneous fpgas. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 613–616, Aug 2007.
- [SBA⁺05] Pete Sedcole, Brandon Blodget, J Anderson, P Lysaghi, and T Becker. Modular Partial Reconfigurable in Virtex FPGAs. In *IEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 211–216, 2005.
- [SDM15] D. P. Seemuth, A. Davoodi, and K. Morrow. Automatic die placement and flexible i/o assignment in 2.5d ic design. In *Sixteenth International Symposium on Quality Electronic Design*, pages 524–527, March 2015.
- [SJA09] M. Samaranayake, H. Ji, and J. Ainscough. Module placement based on hierarchical force directed approach. In *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, pages 1–6, Nov 2009.
- [SKT92] Minshine Shih, E.S. Kuh, and Ren-Song Tsay. Performance-driven system partitioning on multi-chip modules. In *Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE*, pages 53–56, Jun 1992.
- [SR99] Yaska Sankar and Jonathan Rose. Trading quality for compile time: Ultra-fast placement for fpgas. In *Proceedings of the 1999 ACM/SIGDA Seventh*

- International Symposium on Field Programmable Gate Arrays, FPGA '99*, pages 157–166, New York, NY, USA, 1999. ACM.
- [SR12] N. Shah and J. Rose. On the difficulty of pin-to-wire routing in fpgas. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 83–90, Aug 2012.
- [SRRK04] N. Selvakkumaran, A. Ranjan, S. Rajee, and G. Karypis. Multi-resource aware partitioning algorithms for fpgas with heterogeneous resources. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 741–746, July 2004.
- [SS11] K. Siozios and D. Soudris. A tabu-based partitioning and layer assignment algorithm for 3-d fpgas. *IEEE Embedded Systems Letters*, 3(3):97–100, Sept 2011.
- [SSAA03] V. Sklyarov, I. Skliarova, P. Almeida, and M. Almeida. Design tools and reusable libraries for fpga-based digital circuits. In *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, pages 255–263, Sept 2003.
- [SSJ08] P. Spindler, U. Schlichtmann, and F.M. Johannes. A fast force-directed quadratic placement approach using an accurate net model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1398–1411, Aug 2008.
- [SSPS07] K. Siozios, K. Sotiriadis, V.F. Pavlidis, and D. Soudris. Exploring alternative 3d fpga architectures: Design methodology and cad tool support. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 652–655, Aug 2007.
- [SVN04] Greg Stitt, Frank Vahid, and Shawn Nematbakhsh. Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(1):218–232, 2004.

- [TLC04] Nicolas Telle, Wayne Luk, and Ray CC Cheung. Customising Hardware Designs for Elliptic Curve Cryptography. In *Computer Systems: Architectures, Modeling, and Simulation*, pages 274–283. Springer, 2004.
- [VC05] N. Viswanathan and C.C.N. Chu. Fastplace: efficient analytical placement using cell shifting, iterative local refinement, and a hybrid net model. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(5):722–733, May 2005.
- [Ver04] Vereen, L. Soft FPGA Cores Attract Embedded Developers, April 2004. <http://www.embedded.com/electronics-blogs/-include/4024958/Soft-FPGA-cores-attract-embedded-developers>.
- [VF11] K. Vipin and S.A. Fahmy. Efficient Region Allocation for Adaptive Partial Reconfiguration. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 1–6, 2011.
- [WAT14] Alexander Wold, Andreas Agne, and Jim Torresen. Relocatable hardware threads in run-time reconfigurable systems. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture Notes in Computer Science*, pages 61–72. Springer Berlin Heidelberg, 2014.
- [Wir97] M.J. Wirthlin. Improving functional density through run-time circuit reconfiguration. In *PhD thesis*. Springer Berlin Heidelberg, 1997.
- [WK93] Nam-Sung Woo and Jaeseok Kim. An efficient method of partitioning circuits for multiple-fpga implementation. In *Design Automation, 1993. 30th Conference on*, pages 202–207, June 1993.
- [WL86] D.F. Wong and C.L. Liu. A new algorithm for floorplan design. In *Design Automation, 1986. 23rd Conference on*, pages 101–107, June 1986.
- [WL10] Xiaojun Wang and Miriam Leeser. Vfloat: A variable precision fixed- and floating-point library for reconfigurable hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 3(3):16:1–16:34, September 2010.

- [WS00] Maogang Wang and M. Sarrafzadeh. Modeling and minimization of routing congestion. In *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pages 185–190, June 2000.
- [WW91] Ting-Chi Wang and DF Wong. An Optimal Algorithm for Floorplan Area Optimization. In *ACM/IEEE Design Automation Conference (DAC)*, pages 180–186, 1991.
- [WYS00] Maogang Wang, Xiaojian Yang, and M. Sarrafzadeh. Congestion minimization during placement. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(10):1140–1148, Oct 2000.
- [XGA⁺09] M. Xu, G. Grewal, S. Areibi, C. Obimbo, and D. Banerji. Near-linear wirelength estimation for fpga placement. *Canadian Journal of Electrical and Computer Engineering*, 34(3):125–132, Summer 2009.
- [Xil11] Xilinx. Xilinx partial reconfiguration user guide, 2011. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/ug702.pdf.
- [Xil12] Xilinx. Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite, 2012. <http://www.xilinx.com>.
- [Xil16] Xilinx. Vivado Design Suite User Guide - Parital Reconfiguration, 2016. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug909-vivado-partial-reconfiguration.pdf.
- [Xiu07] Liming Xiu. Vlsi circuit design methodology demystified : A conceptual taxonomy. *IEEE Press.*, page 200, 2007.
- [XZL14] C. Xu, W. Zhang, and G. Luo. Analyzing the impact of heterogeneous blocks on fpga placement quality. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, Dec 2014.
- [Yas11] Muhammad Yasir. Introduction to FPGA Technology, May 2011. <http://www.fpgarelated.com/showarticle/17.php>.

- [YDHW05a] Jun Yuan, Sheqin Dong, Xianlong Hong, and Yuliang Wu. Lff algorithm for heterogeneous fpga floorplanning. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 2, pages 1123–1126 Vol. 2, Jan 2005.
- [YDHW05b] Jun Yuan, Sheqin Dong, Xianlong Hong, and Yuliang Wu. Lff algorithm for heterogeneous fpga floorplanning. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 2, pages 1123–1126 Vol. 2, Jan 2005.
- [YGR10] Shaon Yousuf and Ann Gordon-Ross. DAPR: Design Automation for Partially Reconfigurable FPGAs. In *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 97–103, 2010.
- [YPMZ15] B. Yu, D. Z. Pan, T. Matsunawa, and X. Zeng. Machine learning and pattern matching in physical design. In *The 20th Asia and South Pacific Design Automation Conference*, pages 286–293, Jan 2015.
- [YVC14] Jackey Z. Yan, Natarajan Viswanathan, and Chris Chu. An effective floorplan-guided placement algorithm for large-scale mixed-size designs. *ACM Trans. Des. Autom. Electron. Syst.*, 19(3):29:1–29:25, June 2014.
- [ZML⁺07] Pingqiang Zhou, Yuchun Ma, Zhouyuan Li, R.P. Dick, Li Shang, Hai Zhou, Xianlong Hong, and Qiang Zhou. 3d-staf: scalable temperature and leakage aware floorplanning for three-dimensional integrated circuits. In *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 590–597, Nov 2007.
- [ZN00] Xuejie. Zhang and Kam W Ng. A review of high-level synthesis for dynamically reconfigurable fpgas. *Microprocessors and Microsystems*, 24(4):199–211, Aug 2000.
- [ZPJU05] N. E. Zergainoh, K. Popovici, A. Jerraya, and P. Urard. Ip-block-based design environment for high-throughput vlsi dedicated digital signal processing systems. In *Proceedings of the ASP-DAC 2005. Asia and South*

Pacific Design Automation Conference, 2005., volume 1, pages 612–618
Vol. 1, Jan 2005.