Timing-Aware Automatic Floorplanning of Partially Reconfigurable Designs for Accelerating FPGA Productivity

(Sureshwar Raja Gopalan)

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Cameron D. Patterson, Chair
Thomas L. Martin
Paul E. Plassmann

September 1, 2010
Blacksburg, Virginia

Keywords: FPGAs, Reconfigurable Computing, Automatic Floorplanning

Timing-Aware Automatic Floorplanning of Partially Reconfigurable Designs for Accelerating FPGA Productivity

(Sureshwar Raja Gopalan)

(ABSTRACT)

FPGA implementation tool speed has not kept pace with the increase in design size and FPGA density. It is difficult to parallelize place-and-route algorithms without sacrificing determinism or quality of results. We address the implementation problem using a divide-and-conquer approach. The PATIS automatic floorplanner enables *dynamic* modular design, which sacrifices some design speed and area optimization for faster implementation of lay-out changes, including addition of debug logic. Automatic generation of a timing-driven floorplan for a partially reconfigurable design aims to remove the need for implementation iterations to meet all constraints. Floorplan speculation may anticipate small changes to a design. Although PATIS supports incremental design, complete re-implementation is still rapid because the partial bitstream for each block is generated by independent concurrent invocations of the standard Xilinx tools.

This work was supported by DARPA and the United States Army under Contract Number W31P4Q-08-C-0314.

To Mom and Dad

Acknowledgments

I would like to thank my advisor Dr. Patterson, whose inputs and guidance were very helpful throughout my graduate life at Virginia Tech

I would like to thank Dr. Martin and Dr. Plassmann for serving in my committee.

I would like to thank my fellow project members Athira Chandrasekharan, Guruprasad Subbarayan, Tony Frangieh, Yousef Iskander and Dr. Stephen Craven for their valuable inputs which helped better shape my ideas.

A special thanks to Bollos and Mill Mountain coffee shops for providing me with tasty caffeine.

Finally, I would like to thank all my family and friends for their encouragement throughout.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past decade, Field-Programmable Gate Arrays (FPGAs) have been used in a multitude of applications, particularly digital signal processing, high-speed communications and cryptography. The ever-increasing resource requirements of these applications along with the steady rise in the number of transistors on a chip have increased the scope for developing large-scale applications on FPGAs. However, this increase is not without a disadvantage: larger designs require the FPGA implementation tools to run longer to completely place and route (PAR) the design. It is not uncommon for the implementation tools to run for a long time (more than 30 hours) and still not meet all the timing constraints. These unreasonable run-times reduce the productivity of the FPGA design flow, particularly during the development stage of large designs. Any small change in the Hardware Description Language (HDL) source, such as changing signal names or adding debug logic, may require complete re-implementation of the design. This is quite different from software development, where rebuilding is necessary only for code that is changed and its dependencies.

Recent developments in multi-processor architectures provide opportunities to speed up the implementation process, which is not effectively used by any of the current vendor-based tools. Hence, it is desirable to reduce the implementation time by modifying the existing flow in two ways: dividing the larger implementation problem into smaller independent problems, and avoiding complete re-implementation in case of any minor changes to the HDL. Xilinx's current Partial Reconfiguration (PR) flow requires manual intervention and decision making; for example, bus macro placement is tedious. Therefore, in addition to the above mentioned improvements, a fully automated, timing-driven flow is required.

1.2 Contributions

This thesis presents an overview of the Dynamic Modular Design (DMD) FPGA implementation flow and a detailed description of the automatic floorplanner used in the flow. DMD modifies the standard implementation flow in the following ways: changes the static design to a PR design, automates the floorplan generation and bus macro insertion, uses speculation to generate a variety of floorplans, incrementally updates the floorplan in case of minor changes, and checks for timing validity of the floorplan before PAR. The DMD flow leverages vendor tools and improves productivity in the early stages of development when the module logic and interfaces change frequently. With the relative scarcity of parallel PAR algorithms in vendor tools, DMD uses a divide-and-conquer methodology to take advantage of modern multi-core platforms. The various modules in the design are implemented using concurrent and independent invocations of the tools.

The focus of this thesis is on automating floorplan generation, given a hierarchical netlist for a PR design. The floorplanning problem has been well studied for Application-Specific Integrated Circuits (ASICs) but FPGA heterogeneity adds a new dimension. Constraints for

placement of modules in a PR design further complicates the problem. Manually creating an optimized floorplan for an n-module design is a hard problem because the number of possible placements is an exponential function of n. To address existing problems in the standard design flow, a Partial module-producing, Automatic, Timing-aware, Incremental, Speculative (PATIS) floorplanner is developed. PATIS automates floorplan generation by assigning area constraints and satisfying resource requirements of all top-level modules in the design. Academic floorplanners generally aim to reduce the total wirelength of the signals after placement, whereas PATIS tries to meet the timing requirements of the design.

1.3 Organization

This thesis is organized into six chapters. Chapter 1 presents the motivation and contributions of this work. Chapter 2 discusses background and related information used in the thesis. Chapter 3 provides an overall description of the DMD design flow. The remainder of the thesis focuses on the author's contribution to DMD and presents a detailed description and analysis of the PATIS automatic floorplanner. Chapter 4 gives an in-depth description of the algorithms used in automatic floorplanning. An analysis of the automatic floorplanner with practical designs is provided in Chapter 5. Lastly, Chapter 6 summarizes the accomplishments and discusses future work.

Chapter 2

Background

This chapter discusses relevant background material pertaining to the need for an enhanced static FPGA design flow. Special emphasis is given to work related to automatic floorplanning as it is the core of this thesis. This chapter begins with a discussion of basic FPGA concepts, specifically the design flow. It concludes with a brief survey of previous work related to automatic floorplanning.

2.1 FPGAs

FPGAs are general-purpose integrated circuits which can be configured to any digital function required by the designer. It is this configurability that sets FPGAs apart from ASICs. ASICs are custom hardware which cannot change their function after fabrication. Configurability also reduces the time overhead for re-implementation when the design is modified. A typical FPGA layout is shown in Figure 2.1. As seen in the figure, FPGAs contain a plethora of configurable logic blocks and routing interconnects which can be used together to implement the desired design.

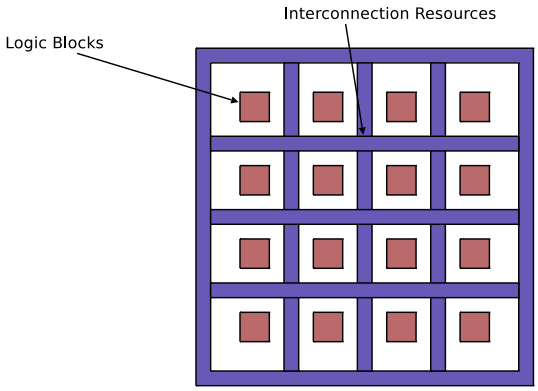


Figure 2.1: Basic FPGA layout

Modern FPGAs are heterogeneous, the need for which stems from the inefficiency of universal logic blocks when implementing certain functions. The main building blocks of modern FPGAs are Configurable Logic Blocks (CLBs), Block Random Access Memories (BRAMs) and Digital Signal Processing units (DSPs). Figure 2.2 shows a portion of the layout of the Virtex-4 FX100 FPGA which highlights the different resource types. Apart from the basic building blocks, FPGAs targeting specific applications have other block types as well. For example, the FPGA shown in Figure 2.2 has PowerPC processors which can be used in embedded applications.

CLBs: The CLB is the basic logic unit of an FPGA and consists of LookUp Tables (LUTs) with 4 or 6 inputs, multiplexers and flip-flops. The LUT can be configured to implement a variety of logic, from shift registers to RAMs. The number of CLBs in the Virtex-4 FPGA varies from 1368 to 22272.

BRAMs: BRAMs are on-chip memories which are available on all modern FPGAs. Xilinx FPGAs provide upto 10 Mbits of on-chip memory in 18 kBit blocks that can support true dual port operation.

DSPs: DSPs implement multiply-accumulate and are primarily used in applications related to signal processing. Computationally intensive tasks can be performed efficiently on the

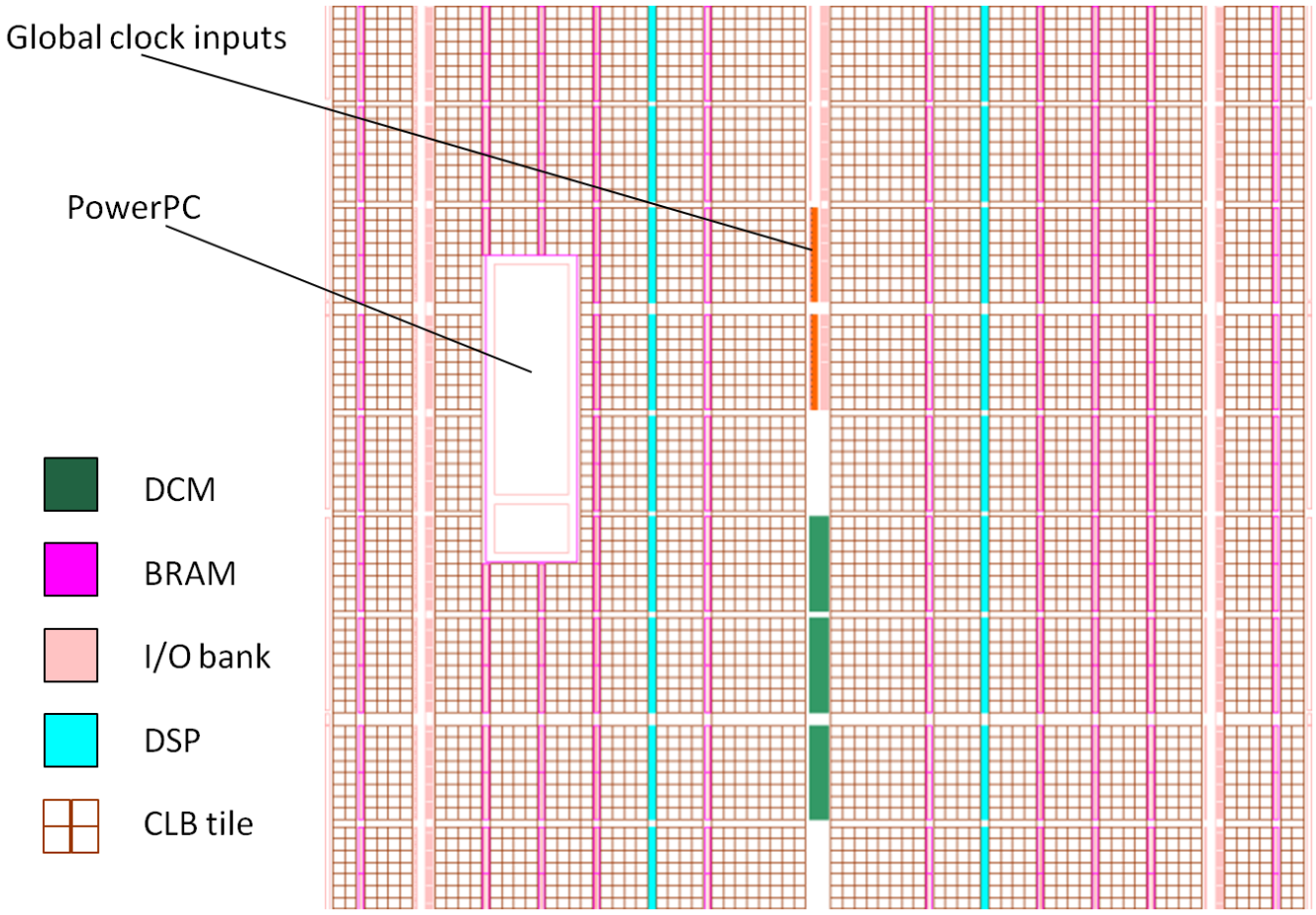


Figure 2.2: Virtex-4 FPGA layout indicating various resources

FPGA using these blocks.

Other Resources: Apart from the above mentioned resources, FPGAs also contain FIFOs, I/O blocks, clock managers and gigabit transceivers.

2.2 FPGA Design Flow

The FPGA, as explained in the Section 2.1, consists of logic and routing which can be configured by the designer. Many existing commercial tools aid the designer in creating and

implementing a design on an FPGA. The Xilinx flow, shown in Figure 2.3, is used to explain the basic methodology. The three main steps are enumerated below [1].

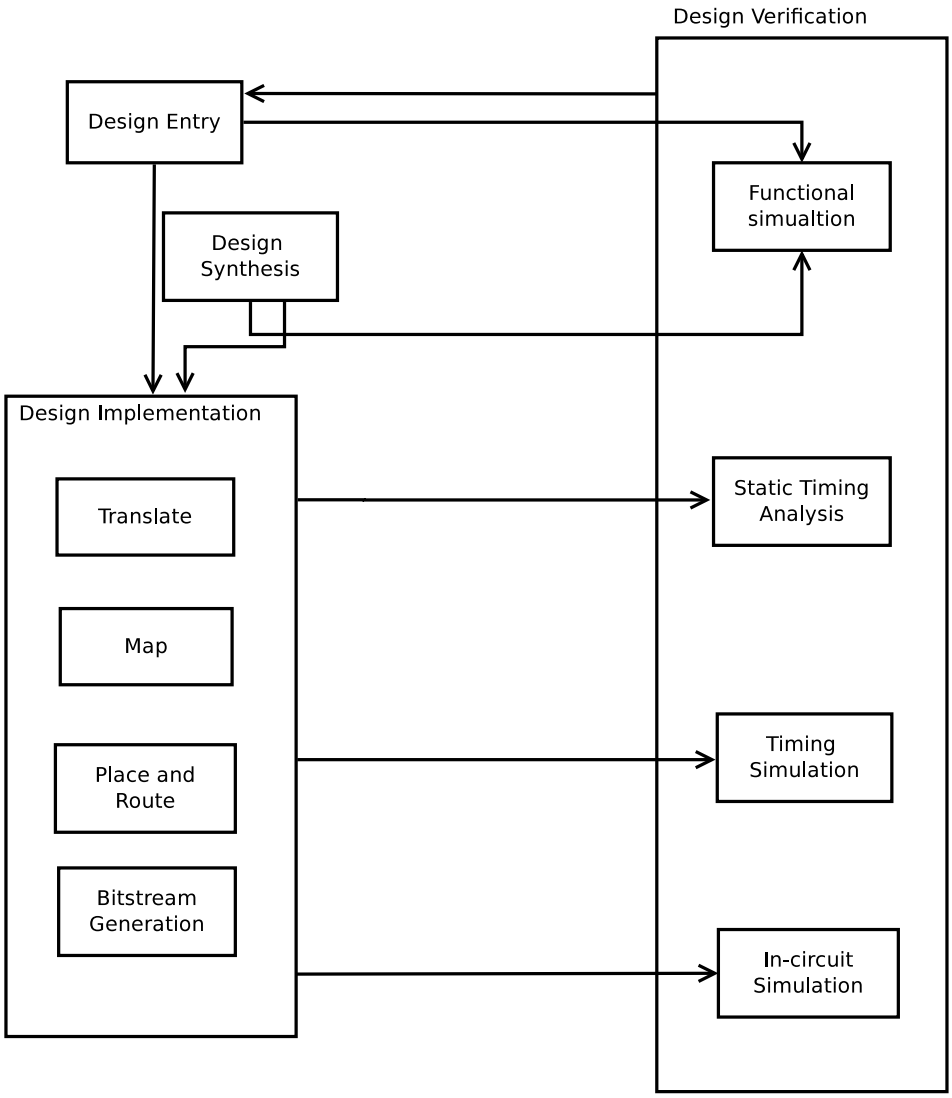


Figure 2.3: Xilinx normal flow

2.2.1 Design Entry

Any design flow begins with user input. In the FPGA design flow, the input is provided either through a schematic or HDL source. HDLs are the preferred means as they are easier

to update and provide technology independence. Apart from HDLs, design constraints are added through timing and placement parameters. Synthesis constraints influence how certain modules and signals are implemented and how logic is inferred. Platform-independent HDL is optimized to a vendor-specific format by design synthesis. Behavioral information in the HDL is converted into a hierarchical or flattened structural netlist. The netlist obtained as output from design synthesis is often proprietary. In order to support third party synthesis tools, most vendors support the common Electronic Design Interchange Format (EDIF) [2].

2.2.2 Design Implementation

Design implementation takes in the synthesized netlist as input and generates a bitstream to configure the FPGA logic. The implementation process is explained using the Xilinx flow. The first step is to map the logic in the design onto specific device resources: LUTs, flip-flops, BRAMs and other. In this step, the Native Generic Circuit (NGC) netlist which is built for behavioral simulation, is translated into a Native Generic Database (NGD) netlist, based on SIMPRIM primitives, which contains approximate information about switching delays. The SIMPRIM library is used for post-NGDbuild (gate-level functional), post-Map (partial timing), and post-PAR (full timing) simulations. These primitives are then mapped onto specific resources which gives precise information about switching delays.

The mapped NGD does not give information regarding propagational delays because none of the logic components are actually placed on the FPGA. This is done by the place-and-route tool, which defines how modules are located and interconnected inside an FPGA. Placement is more critical than routing because bad placement makes good routing impossible. PAR accounts for timing constraints set up by the designer; the timing constraints can either be included at the design entry stage or after design synthesis. If at least one constraint can not

be met, PAR returns an error, or may try to achieve timing closure by using the multi-pass option. The cost table value is modified and the entire design is re-implemented. The PAR tool also has an option to run iteratively until all the constraints are satisfied — this is useful when the design is close to meeting all timing goals. The output of the PAR program is also stored in the Native Circuit Description (NCD) format. Finally, the bitstream is generated to configure the FPGA.

2.2.3 Design Verification

A major component of the design flow is testing the functionality and performance of the design. The Xilinx flow allows three different design verification techniques over various stages of the flow. The first technique is simulation-based verification, primarily functional or timing simulation. Functional simulation verifies the logic of the design before the implementation phase. Early simulation of the design makes it easier and faster to correct design errors. Timing simulation verifies the operation of the design under worst-case conditions. This is done after the design implementation stage when delays are known. The second technique is static timing analysis which is primarily used for quick timing checks of a design after it is placed and routed. Timing analysis is done by determining the critical path delays in the circuit and identifies paths which violate timing constraints. The final technique is in-circuit verification, which verifies the operation of the design under typical conditions by downloading the bitstream onto the target device.

2.3 Improvements in Design Flow

The standard flow has drawbacks when used with large designs. For any small change in the HDL, synthesis, placement and routing need to be rerun on the entire design, which consume a considerable amount of time. In the software domain, this is analogous to recompiling all the libraries every time a change is made in the application code. Changes in logic can increase critical path delays or lead to design timing failure. The tools need to discriminate between small local changes and attempt to re-implement components that are directly affected. This section summarizes the enhancements made in the tools to address these limitations.

Xilinx introduced the *Modular Design Flow* (MDF) [1] to enable the designers to work in parallel on different components of the design. Designers can modify one module while leaving other stable modules intact. The main drawback with this approach is the need to partition the design during the development stage. In general, the design interfaces and implementation algorithms evolve over time, requiring re-partitioning of the design. MDF approach also necessitates efficient communication between designers to ensure proper functioning of the final design.

Synplicity and Xilinx formed a task force to tackle timing closure issues for ultra high-capacity FPGAs [3]. The focus of the task group was "fast implementation run-times and predictable timing results for small design changes, especially towards the end of a design cycle" [4]. Incremental design reduces the run-times of the tools by re-implementing only those parts of the design which have been modified since the previous run. Two different methodologies were used to incorporate the incremental design flow:

1. *SmartGuide* [5]: Name matching between the previous and current netlists is used to guide the tools to re-implement the changed components alone. This methodology

is useful towards the end of the design cycle for rapid re-implementation with minor modifications, primarily changes in pin locations and timing constraints.

2. *Partitions* [6]: This methodology is used to reduce the run-times of block-based designs, where each block is defined as a partition. The specification of partitions is done upfront during the design entry or early implementation phases. Partitions which are not modified during repeated runs of the tools are re-implemented using a "cut-and-paste" approach, hence the time for re-implementation is dependent only on the number and size of partitions that are modified.

An incremental flow necessitates the use of *incremental synthesis*, an enhancement over traditional synthesis. Incremental synthesis is capable of detecting the logic that has changed since the last implementation and works on those changes, while other unmodified portions are reused. Ideally, incremental synthesis is integrated into existing design flows without much effort and easily identifies changes at the functional level, disregarding changes to irrelevant parts of the design such as comments, while preserving the quality of results [7].

While incremental synthesis reduces the input size of the problem by reducing the logic to be processed during each iteration, it exhibits the same limitation as its predecessor: optimizing logic without any notion of placement or routing delay estimate, when routing is often the main contributor to the total delay. This leads to several design iterations and makes timing closure more difficult. At this point, knowledge of the underlying device's physical properties, in particular routing delay, necessitates a physical synthesis flow [8]. By integrating the synthesis tool and placer, synthesis is fully aware of module placement and has an improved estimate of routing delay between each module. This new visibility reduces design iterations and promotes faster implementation convergence. Figure 2.4 compares the Xilinx standard and physical design flows.

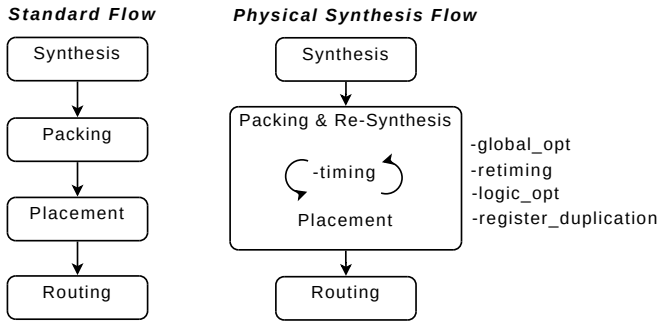


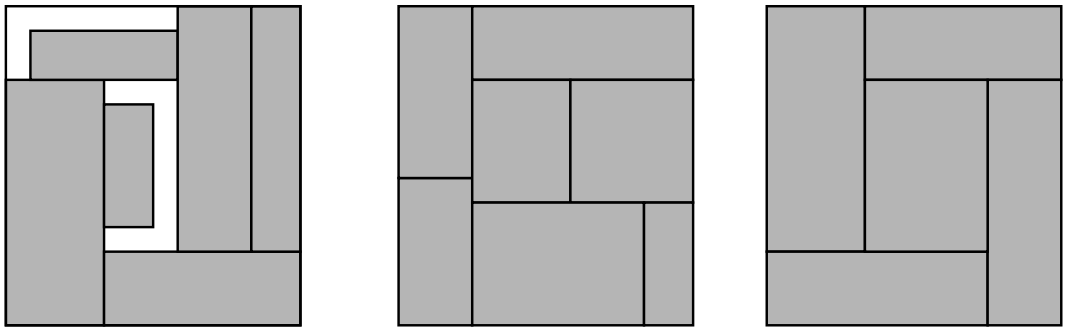
Figure 2.4: Xilinx standard and physical design flows

Today physical synthesis is provided by all major Electronic Design Automation (EDA) and FPGA tools. Xilinx ISE 9.1i (and later versions) includes several physical synthesis options added to the map stage shown in Figure 2.4: enable timing-driven packing and placement, logic remapping and trimming, logic and register duplication, re-timing, restructuring and re-synthesis along with incremental placement and timing analysis.

2.4 Automatic Floorplanning

Floorplanning is the method of dividing a given layout into non-overlapping basic rectangles. A rectangle which is not subdivided by a line segment is called a basic rectangle [9]. Floorplanning is a key step in the hierarchical design flow. The importance of floorplanning in modern, high density FPGAs is described in [10]. Floorplanning is the method of generating an optimized layout for a given set of modules while respecting certain constraints. Sastry and Parker [11] have shown that the two-dimensional floorplan compaction problem is NP-Complete, while Stockmeyer [9] showed that area minimization problem for general floorplans is strongly NP-Complete. Classic floorplanning tries to minimize unused space (hence overall area) and the length of wires connecting modules. Figure 2.5 (i) shows a sample floorplan where the unused space appears as white space. The problem of finding an efficient floorplan is a research area in combinatorial optimization. Most problems related

to finding optimal floorplans are hard and require vast computational resources. Two approaches are generally used to generate optimal floorplans: (i) using heuristics, (ii) restricting the design methodology to certain types of floorplans. There are two classes of floorplans: A slicing floorplan in which modules are assigned through repeatedly dividing the layout using horizontal and vertical lines [9]. Anything else is referred to as a non-slicing floorplan as shown in Figure 2.5 (iii)). Slicing floorplans (Figure 2.5 (ii)) are easier to implement because of their recursive structure, but they restrict flexibility. A slicing floorplan is represented using a slicing tree which is equivalent to the Binary Space Partition (BSP) tree used in graphics. For our purposes, the floorplan should assign a rectangular region to every module and port locations on each module's perimeter.



(i) Simple floorplanned design (ii) slicing floorplan (iii) non-slicing floorplan

Figure 2.5: Floorplan types

2.4.1 Related Work

Traditionally, floorplanning algorithms targeted ASICs or sea-of-gates style FPGAs. The first floorplanning methodology for heterogeneous FPGAs was proposed by Cheng and Wong [12]. A slicing floorplan is initially generated and simulated annealing was used to perturb it with the objective of minimizing the area and half-perimeter wirelength (HPWL). An Irreducible Realization List (IRL) is used to store an enumeration of distinct realizations of a node in a

slicing tree. IRLs were generated based on a basic pattern on the chip. The basic pattern is the building block of an FPGA, repeating the pattern along orthogonal axes gives the FPGA.

The Least Flexibility First floorplanning methodology proposed by Yuan et al. [13], targeted packing the design on the chip rather than optimizing for wirelength. An Atomic Realization List, quite similar to the IRLs described above, is generated and sorted based on the flexibilty of each realization. The flexibility depends on the physical location of the realization on the chip — corner, side or hollow. A list of possible placements of all modules is generated using the individual Atomic Realization Lists, and the one with the best fit is chosen.

Feng and Mehta [14] proposed a two-step solution to the FPGA floorplanning problem. The first step generates a resource-aware floorplan based on a fixed-outline simulated annealing approach. Parquet, a fixed-die ASIC floorplanner, was adapted to handle resource-aware floorplanning of FPGAs. A fixed-outline approach was preferred to area minimization [12] because the FPGA is a bounded rectangle. The cost function of simulated annealing penalizes any mismatch in resources allocated. The second step tweaks the placement of modules obtained from the previous step by a constrained floorplanning approach based on a min-cost-max-flow network formulation.

Banerjee et al. [15] outlined a topology-based floorplanning methodology for heterogeneous FPGAs. It is a three-stage methodology: (i) slicing tree generation, (ii) topology of shapes for each module, and (iii) allocation of rectangular regions satisfying resources. The slicing tree is generated by recursively bisecting the design netlist based on balanced min-cut. The granularity of the shapes is defined by the tile size of the FPGA. The topology of shapes is made irredundant by making individual widths and heights distinct. Modules are assigned rectangular regions through level-order traversal of the slicing tree. This method generates a list of floorplans, and the best is chosen based on least HPWL.

All the methodologies listed above generate floorplans for a static design. The complexity of the floorplanning problem increases for a PR design because of the additional placement constraints on modules. In [16], Banerjee et al. generate floorplans for a PR design using the methodology discussed above with additional constraints. In [17], Singhal and Bozorgzadeh present a novel approach to reduce the reconfiguration overhead by effectively floorplanning the static and dynamic modules. They introduce a new multi-layer sequence pair representation which attempts to maximize the overlap area between consecutive designs. The overlap area is the region which has the same configuration in both designs. If common modules have the same location in both designs (in terms of reconfiguration frames), the overlap would be generated and recognized.

2.5 Need for a New Design Flow

The standard and improved design flows explained in Sections 2.2 and 2.3 try to solve the implementation problem as a global optimization problem. This approach generally requires implementation iterations to satisfy tight timing constraints. Also during the development stage of a project, a small modification in HDL requires complete re-implementation. PATIS recasts the global optimization problem as a set of independent local implementation problems which are solved by a divide-and-conquer approach. To accommodate small updates, PATIS incorporates an incremental floorplanner.

All the floorplanners described in Section 2.4.1 aim to reduce the total wirelength of the floorplanned design. Reducing the overall wirelength does not guarantee satisfaction of all timing constraints. The aim of a timing-driven approach is to minimize the maximum delay. DMD tries to satisfy all timing constraints by using a timing-driven floorplanner.

Chapter 3

System Overview

The standard FPGA flow has several drawbacks when used in the development stage of a fairly large design. For a small change in the HDL, the entire design may have to be re-implemented. This motivates the need for a flow which reduces the turn-around time both when the entire design is implemented and when the design undergoes minor modifications. Implementation time for the entire design can be reduced by breaking down the bigger problem into many smaller problems which can potentially be solved faster. When there are minor changes to the logic circuitry, speedup can be achieved by selectively re-implementing the modified components of the design without affecting the other components. Implementing the design as independent modules offers a way to selectively re-implement only the changed modules. The Xilinx PR flow uses this approach to reconfigure on the fly. PATIS leverages the same methodology to improve the productivity of static designs.

3.1 Xilinx PR Flow

The Xilinx PR flow offers designers a way to time-share FPGA resources between modules. In such designs, one section of the FPGA operates continuously while other sections of the FPGA are partially reconfigured to provide new functionality. This technique is the spatial analogy to a microprocessor managing context switching between software processes. Thus the PR flow provides a way to selectively modify parts of the design while not disturbing the rest of the logic.

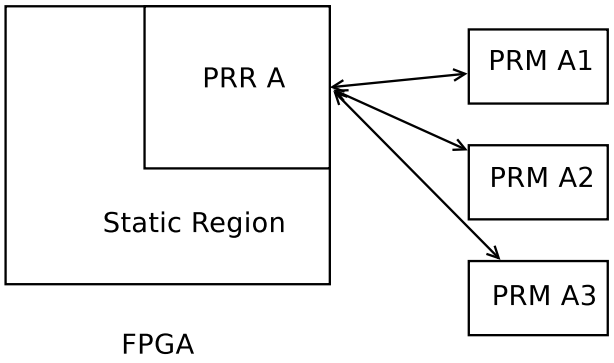


Figure 3.1: Various regions in a PR design

Before continuing with details of the Xilinx PR flow, certain terms are explained below:
Static Region: The static region is the portion of the design that does not change during partial reconfiguration and may include logic that controls the partial reconfiguration process.

Partially Reconfigurable Region (PRR): PRRs are regions on the FPGA fabric which contain logic that can be reconfigured independent of the base region and other PRRs. Figure 3.1 shows PRR `A`.

Partially Reconfigurable Module (PRM): PRMs are modules that can be swapped in and out of a PRR. A PRR can accomodate only one PRM at a time but multiple PRMs over a period of time. Figure 3.1 shows three PRMs – `A1`, `A2`, `A3` – for PRR `A`. Only one of the three PRMs can be implemented in the PRR at a given time.

Run-Time Reconfiguration (RTR): Run-time reconfiguration is defined as the ability to modify or change the functional configuration of certain modules on the fly while other modules continue uninterrupted operation.

Figure 3.2 illustrates the PR flow as described in the Xilinx Early Access PR User Guide [18]. The first four steps are similar to the non-PR FPGA flow while the last three are unique to the PR flow. In the design entry phase, the static and PR regions are clearly demarcated using communication interfaces called bus macros. Following design entry, the constraints for the design need to be specified. In addition to timing requirements, PR designs must be constrained using area groups to create a bounding box for each module. The next step is an optional non-PR implementation which is crucial for design debug, aids initial timing and placement analysis, and helps in determining the best area group range and bus macro locations. This step is used to locate an optimized placement for the modules and bus macros for the design. In a PR design, the static region has to be implemented before any of the PR regions because some routing resources within the PRRs may be used to implement static routes and cannot be used by PRMs. The routes within PRRs used by static region are stored in the `arcs.exclude` file, which is given as input when the PRRs are implemented. Whenever the static region is re-implemented, the `arcs.exclude` file changes necessitating re-implementation of all PRMs. The final step in the Early Access PR flow is merging the static and PRM implementations to generate a single bitstream.

The important features of the PR flow are described below:

1. The location constraints for each PRR are fixed and need to be specified up front. This implies that the constraints assigned to a PRR should satisfy the resource requirement of the biggest PRM associated with it. Consider a video filter application with a simple `grayscale` filter and a complex `convolution` filter implemented using the PR flow. The constraints of the PRR should satisfy the requirements of the `convolution` filter,

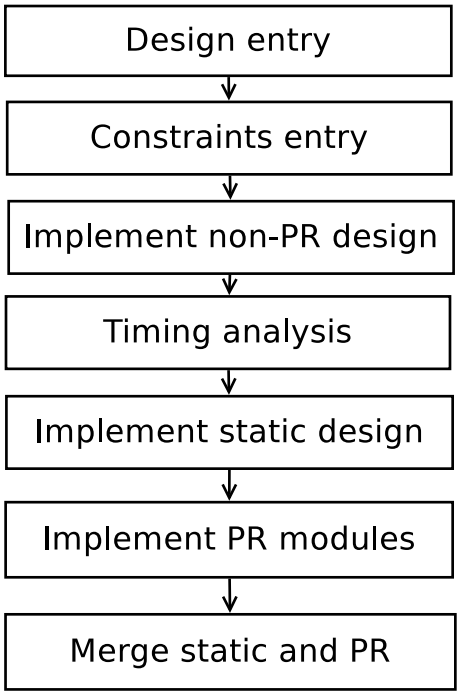


Figure 3.2: Xilinx EAPR flow

hence when the `grayscale` filter is implemented the resources are under-utilized. This can be a disadvantage in cases where the PRM uses a small percentage of the resources allocated to the PRR.

2. Signals used as communication paths between or through reconfigurable modules need to use fixed routing resources. The routing resources used for such intermodule signals must not change when a module is reconfigured. Bus macros are used to provide communication interfaces between PRRs and static region or another PRR. The placement of the bus macros is manual and constrained to the module boundaries. Figure 3.3 shows the bus macro interface between PR modules, and between a PR module and static logic. Every connection in or out of a PR module has to be interfaced through a bus macro.

3. All PRMs for a given PR region must be pin-compatible with each other, have the

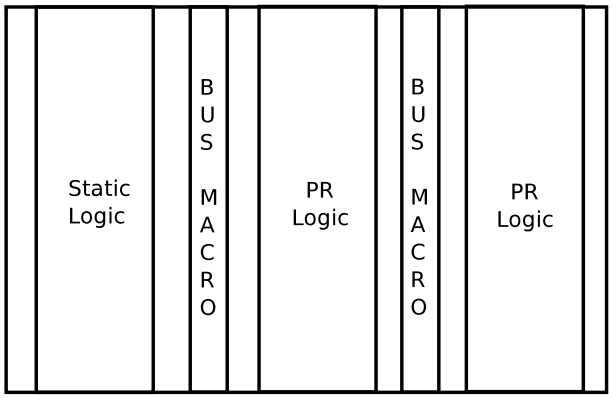


Figure 3.3: Bus macro connections

same port definitions and entity names. Consistent naming allows each PRM to be linked from the same top-level description.

4. The granularity of configuration frames is another important feature of the PR flow. A configuration frame is the smallest portion of configuration data which may be read from, or written to the configuration memory. To necessitate logic changes within a PRR without affecting the functionality of the rest of the design, no two PRRs should share a configuration frame. Less logic associated with a configuration frame allows for higher granularity of configuration data. In the Virtex-II FPGAs, the configuration frame was as tall as the height of the chip, while the configuration frame of Virtex-4 series is 16 CLBs tall.

3.2 PATIS - A Modified PR Flow

Despite the application advantages of RTR, it is still not used as a commercial mainstream design methodology mainly because of the additional design complexity seen from the PR flow features enumerated above. The act of swapping modules into and out of the reconfigurable region turns a single design into multiple, distinct designs, effectively increasing the

test space. Unlike most components within the manufacturer's portfolio, RTR cannot be simulated and developers must test directly on the hardware where visibility is limited.

DMD uses RTR as a design-time methodology rather than as a run-time strategy. Every instance in the top-level module of the static design is assigned a PRR which can be expanded when required, based on the *viscosity*. In this context, viscosity is defined as the likelihood of a module to change in logic and interface. Less viscous modules are subject to structural and constraint (such as interface) changes. These modules are tagged as partially reconfigurable and are allocated extra white space to minimize the ripple effect of changes on the overall layout. High viscosity is generally associated with frozen, debugged modules. The top-level logic generally consists of interconnections between module instances and constitutes the static logic. The DMD flow, as shown in Figure 3.4, is explained in detail in this section.

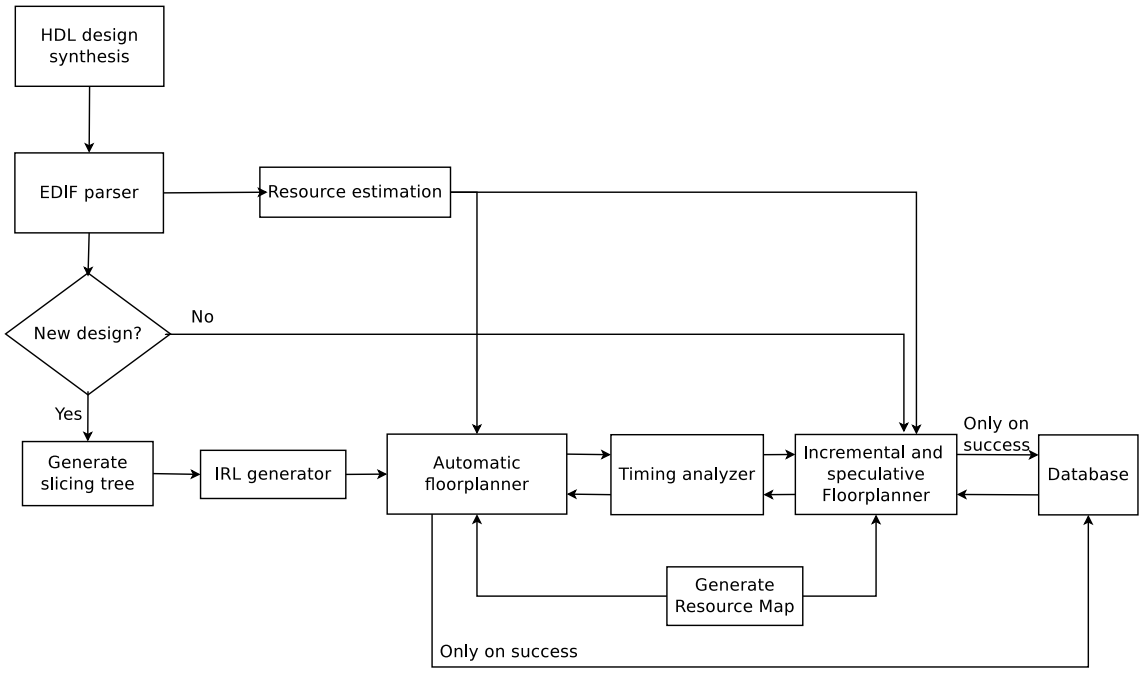


Figure 3.4: Overall DMD flow

DMD modifies the existing PR flow to improve the productivity of static designs during development. There are two different modes of operation:

1. *Designs implemented for the first time*: In this case, the PATIS automatic floorplanner is used to generate a floorplan. The floorplan is timing verified by placing bus macros using the Xilinx PAR tool. To accommodate future changes in the modules, PATIS makes local changes to the valid floorplan by allocating extra resources. New floorplans obtained are stored in a database used by the incremental floorplanner.

2. *Designs re-implemented after modifications*: The resource requirements of modules may change drastically during the development stage. PATIS tries to generate a new floorplan for the modified design in two steps: (i) the database is searched for a floorplan that satisfies all the resource constraints, (ii) if no such floorplan exists, then the current floorplan is modified incrementally to accomodate the changes. If PATIS cannot generate an incremental floorplan , the automatic floorplanner is used to create one from scratch.

The floorplan generated is represented as a User Constraints File (UCF) and is used during implementation of the design. UCF provides the implementation tools with information about placement of modules, nets and required timing. Static logic is implemented first, followed by the PRMs which are implemented simultaneously on different cores. Important components of the DMD flow are explained below.

3.2.1 Automatic Floorplanner

The automatic floorplanner is used to assign area constraints to module instances minimizing the distance between interconnected modules. Floorplanning is used between synthesis and implementation. Floorplanning any design requires a precise resource estimate for each module, for which a synthesized netlist is required. PATIS generates a floorplan under two conditions: (i) whenever a design is processed by the DMD flow for the first time, and (ii)

when the incremental floorplanner is not able to accommodate changes in the design by modifying the floorplan. A floorplan is feasible if it satisfies the resource requirements for each module and achieves timing closure. The first constraint is satisfied while generating the floorplan, and the latter is verified when placing bus macros, explained in Section 3.2.3. The automatic floorplanner is discussed in detail in Chapter 4.

3.2.2 Incremental Floorplanner

The PATIS incremental floorplanner operates in two modes, as shown in Figure 3.5. PATIS initially determines whether a design is new or modified. New designs go through a speculative flow, in which the PATIS incremental floorplanner attempts to generate several feasible floorplans. Modified designs are checked against this pool of floorplans to find one that fits the new resource requirements.

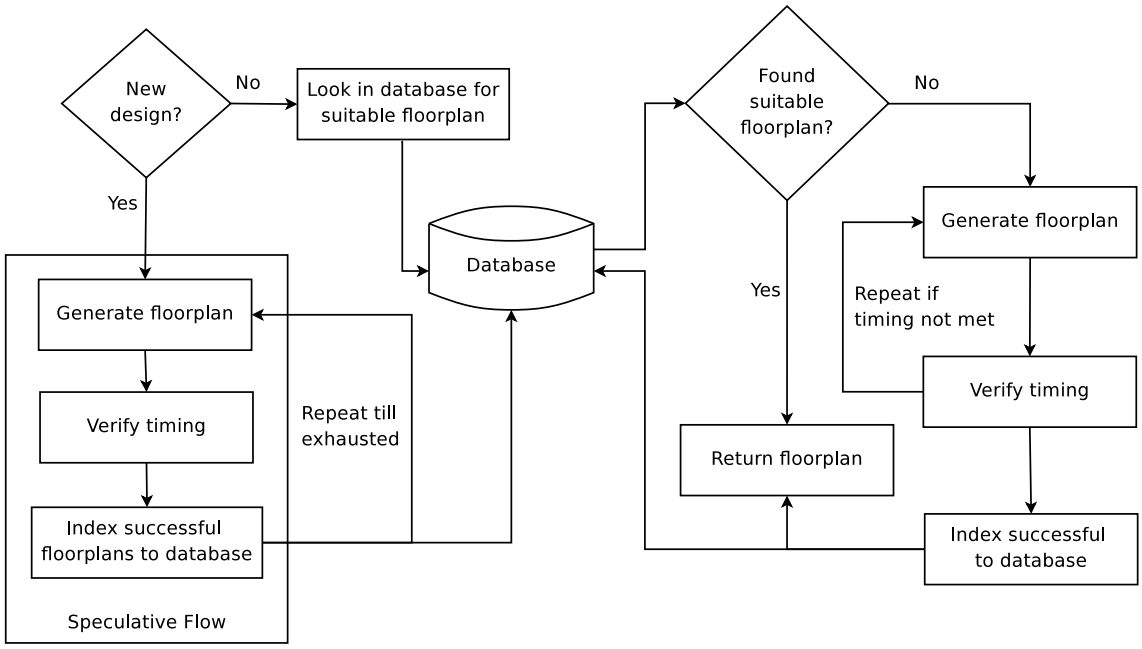


Figure 3.5: Speculative incremental flow

Once the automatic floorplanner generates a floorplan for a new design, PATIS speculatively floorplans the design to generate multiple variants of the initial floorplan. Several metrics control this speculative behavior. Modules are floorplanned in the order of their increasing viscosities to generate several possible layouts. Since modules with low viscosities have a higher susceptibility to change, PATIS prioritizes them. In addition, *fickle* modules are frozen in place and hence not speculatively floorplanned by PATIS. Each module has a fickleness weight that reflects the difficulty of meeting its timing constraints. Such modules may need to be implemented several times before finally satisfying timing.

Modules have a minimum resource constraint that cannot be violated. Timing constraints of generated floorplans are verified before PATIS indexes them into a database. PATIS speculatively floorplans the design until all reasonable aspect ratios are explored. Floorplan variants alternately shift each vertical and horizontal edge of the modified module until prevented by the minimum resource constraint of the adjacent modules. Initially, neighboring unoccupied resource rows or columns will be allocated to the module. If these resources are exhausted or cannot be occupied, PATIS attempts to move or shrink neighboring modules. Once all possible aspect ratios are explored, the process stops.

When a module in a design has changed, PATIS scans the database for a floorplan meeting the new resource requirements. If the search is unsuccessful, the incremental floorplanner attempts to modify the current floorplan and generate a new feasible floorplan. During incremental floorplanning, PATIS tries to preserve the current floorplan topology while maintaining reasonable aspect ratios for the modules. To meet these often-conflicting restrictions, PATIS first looks for unoccupied resources (white space) around a module and only then attempts to modify neighboring modules. Once a floorplan is incrementally modified to generate a new suitable floorplan, PATIS again verifies its timing.

Incremental floorplanning of a module, both speculative and otherwise, is iteratively performed by two algorithms: *white space occupation* and *neighbor displacement*. The white space occupation algorithm looks for unoccupied resources around a module. Should there be any unoccupied resources in the vicinity of the module being floorplanned, the tool alternately shifts the vertical and horizontal edges of the module. If neighboring white spaces are exhausted or cannot be occupied, the neighbor displacement algorithm modifies neighboring modules in addition to the current module. Adjacent module edges are shifted away from the module being floorplanned while still respecting their resource requirements. The neighbor displacement algorithm performs translation and shrink operations. Occasionally, an adjacent module could be frozen in place due to severe resource or timing constraints. In such a case, one or more edges of the module being modified are locked and exploration continues along the remaining directions. If no solution exists that meets timing, an entirely new floorplan is generated.

3.2.3 Bus Macro Insertion

A PR design contains one or more reconfigurable modules that are implemented inside designated PR regions. Bus macros are physical ports on the PR region boundary that provide route-locking for the signals from/to the reconfigurable module, with the exception of the clock signal. Route-locking is necessary to connect the routing for the static region to the ports of the reconfigurable module constant. The PR module uses the routing resources contained only within the PR region. The `arcs.exclude` file reserves routing resources inside a PR module for use by the static region. As a result, the static region will remain unaffected when the PR module is reconfigured.

The conventional PR flow requires bus macros to be instantiated and placed manually. Bus macros for Virtex-4 are directional, and the directionality of the bus macro to be used is relative to the placement of the PR module and its interconnections. As the DMD flow implements modules in PR regions, bus macros are needed for module interfaces. Manual instantiation and placement of bus macros is a time-consuming process. The placement of bus macros influences timing and hence the design frequency [19]. Simulated annealing is used in [19] to automate the placement of bus macros based on the timing score reported by PAR. The tool placed and routed the entire design and required 250 iterations of simulated annealing. PATIS instead uses PAR to directly place the bus macros, thus improving the tool run-time. The bus macro instantiation process is automated and requires no additional direction from the designer as compared to a standard PR design.

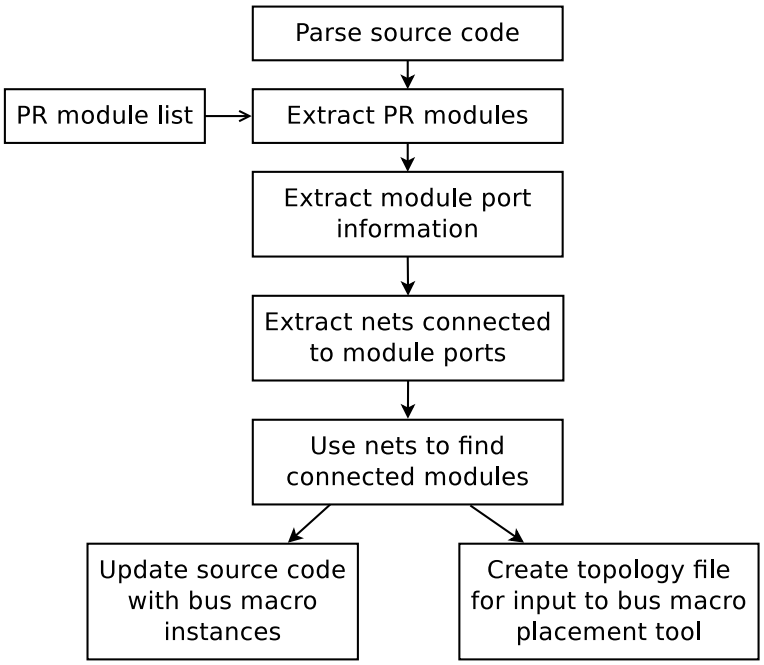


Figure 3.6: Bus macro insertion flow

The overall flow of the bus macro insertion tool is illustrated in Figure 3.6. Module instances are obtained by parsing the top-level HDL. PATIS uses the input and output port information

to compute the number of bus macros required on the module boundary, and creates the inter-connection graph. The input to the PAR tool is a topology file with the module and bus macro instances. Placement of bus macros is constrained to module boundaries using a Physical Constraints File. The `fpga_edline` tool extracts the bus macro positions from the placed and routed NCD file, and the UCF is updated.

3.3 Debug Support

Debug activities in DMD are handled by two mechanisms: Low-Level Debugging (LLD) and High-Level Validation (HLV). Much like simulators and the embedded logic analyzer cores provided by FPGA vendors, LLD handles data at the bit-level. However, LLD is not based on capture methodologies which rely heavily on embedded memory to record signal activity, but instead is based on conditional breakpoints such as those used in software development to halt the design and enable it to be stepped and analyzed using a microprocessor. Breakpoint logic is implemented in a designated reconfigurable region where it can be modified and rapidly re-integrated without altering the rest of the design. Register state is retrieved through an Internal Configuration Access Port (ICAP), a Xilinx proprietary interface allowing direct access to register state. LLD is illustrated in Figure 3.7.

HLV, as shown in Figure 3.8, abstracts away low-level implementation details by creating a framework for validating individual modules against a functional model written in a high-level language implementation. Design validation can be automated in the same manner as used in software unit-testing, where individual components are tested against known conditions.

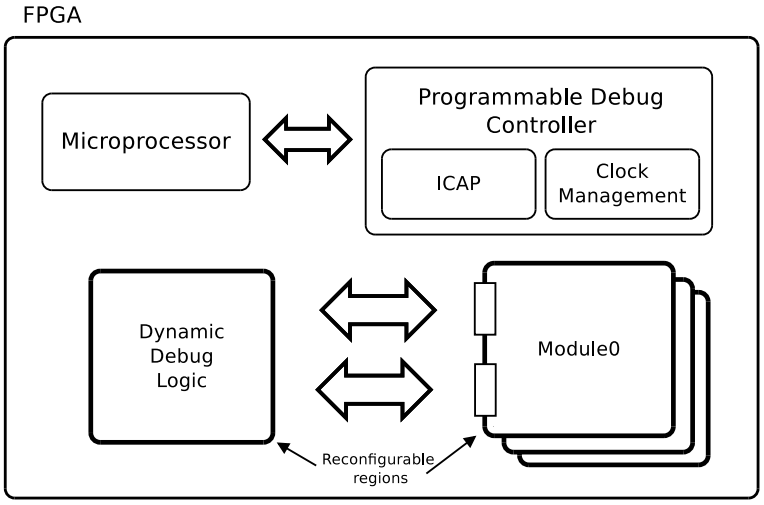


Figure 3.7: LLD interaction

3.4 PlanAhead

DMD aims to automate the entire FPGA design flow from synthesis to bitstream generation while leveraging the efficiency of commercially available tools to perform place-and-route, synthesis and bitstream generation. For this purpose, it is necessary to have an interface to the Xilinx tools. A Graphical User Interface (GUI) tool is required to view the floorplans generated by the automatic and incremental floorplanners. *PlanAhead*, the manual floorplanning tool used in the Xilinx PR flow, is used as the interface. Figure 3.9 shows a screenshot of the PlanAhead GUI for one of the example designs which was manually floorplanned.

PlanAhead 10.1i offers several advantages that can be leveraged by the PATIS flow:

1. *Hierarchical floorplanning*: PlanAhead supports a hierarchical, block-based and incremental design methodology. These features are necessary in the DMD reconfigurable design flow.

2. *ISE 9.2i PR runs*: PlanAhead 10.1i can be used to invoke the ISE 9.2i PR implemen-

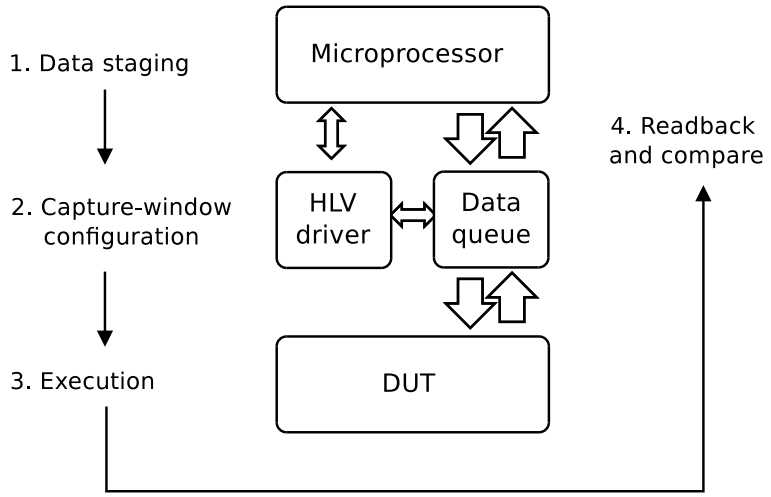


Figure 3.8: HLV flow block

tation tools.

3. *Parallel runs*: When implementing a reconfigurable design, PlanAhead 10.1i allows PAR jobs to run in parallel on different reconfigurable regions. This can be exploited by a multi-processor system with adequate memory and memory bandwidth.

4. *Different PAR strategies*: PlanAhead allows PAR jobs to be run with different strategies. Each strategy tries to intelligently optimize various PAR options.

5. *Assembling static and partial bitstreams*: It is necessary to generate the complete bitstream whenever the design is implemented the first time or has been changed in such a way that incremental floorplanning is not possible. PlanAhead 10.1 can generate and assemble the static and partial bitstreams using the PR assemble option.

6. *Batch mode usage*: In batch mode, primarily using Tcl scripts, PlanAhead can be used to obtain accurate resource estimates for a hierarchical design. Also UCF generation with Relationally Placed Macros (RPMs) coordinates is made easy using PlanAhead.

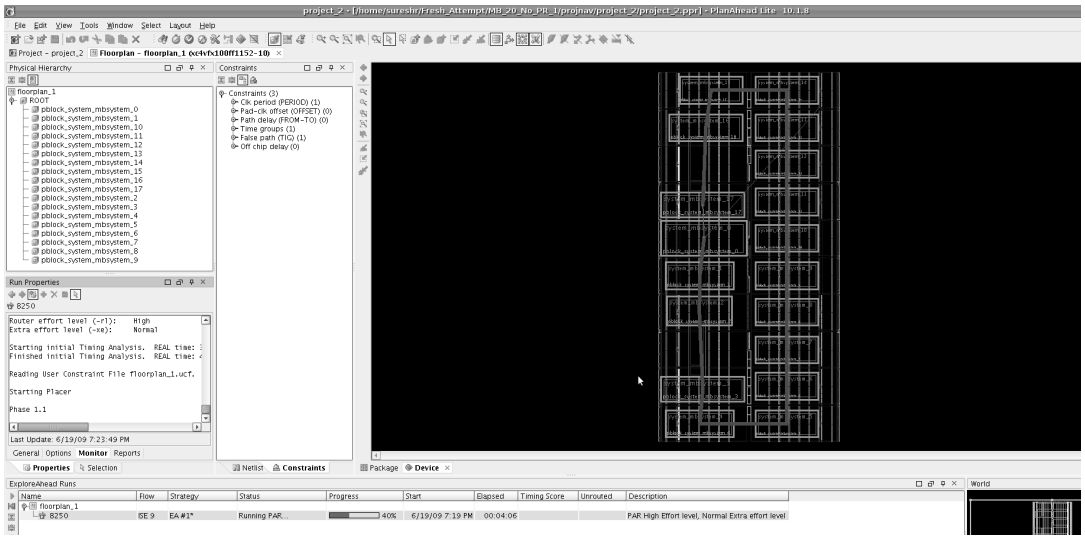


Figure 3.9: PlanAhead screenshot

Table 3.1: Place-and-Route times

Design / # modules	f_{clk} **(MHz)**	**Elapsed PAR time (minutes)**		
		Manual flat design	**Manual static FP**	**Manual PR FP**
CFFT 3	256.40	-	35	10
MB 5	127.40	330	80	20

Legend: '-' indicates the design did not meet timing in less than 24 hours. FP - FloorPlan

3.5 Experimental Verification — Proof of Concept

Before automating the DMD flow, it was necessary to show the implementation acceleration proposed is achievable and practical. For this purpose, a few static designs were run through the DMD flow with manual floorplanning and bus macro placement. Table 3.1 shows the run-times for example designs using different approaches.

The following observations were made from the results of the proof-of-concept experiments, which serve as evidence of the DMD approach advantages.

1. The tool run-times reduced considerably when using the PR flow with manual floorplanning and parallel module implementations. Floorplanning improves run-time by localizing component placement and routing. Since there is no overlap between the location constraints of the modules, they can be implemented in parallel. This offers tremendous potential to improve the run-times of tools by leveraging modern multi-core processors. As seen from Table 3.1, parallel implementation of modules is considerably faster and improves turnaround time.

2. Implementation tools fail or take excessive time on large designs when given severe timing constraints. Non-timing-driven placement and routing can be faster by an order of magnitude. Although flattening the design increases placement and routing flexibility, run-time escalates dramatically.

3. Implementation time is proportional to the design size. A large design will require more time than a smaller design with similar timing constraints, presumably because less densely packed devices reduce contention for preferred logic and routing resources.

4. With large designs and strict timing constraints, the normal design flow tends to take a long time, or does not meet timing. This is partly because the tools must repeatedly re-implement the entire design instead of just focusing on modules with timing deficiencies.

5. The location constraints for every module determine the area used by the module for logic implementation and routing. Module regions should not be excessively large so as to waste resources, nor excessively small such that local resource contention arises.

The PATIS flow seeks to reduce implementation time by leveraging PR. When a module is modified, there is no need to re-implement the entire design provided the floorplan has sufficient flexibility to minimize ripple effects across module boundaries. This flexibility is

analyzed by the incremental floorplanner which generates a variety of floorplans by modifying the module boundaries.

Chapter 4

Automatic Floorplanning

Automatic floorplanning is a key component of the PATIS flow as it aims to place interconnected modules together thereby minimizing inter-module routing delays. In the normal flow, floorplanning is a manual process and is done after hierarchical synthesis. However, manual floorplanning can be a very hard task with complex non-planar designs. Hence in the DMD flow, floorplan generation is automated to reduce the burden on the designer. With increasing FPGA resource heterogeneity, the number of resource types whose requirements are to be separately satisfied increases thereby making automatic floorplanning a hard problem. Another complicating factor is the difference in the density variations of the resources over the FPGA. This density difference can result in under-utilization of one resource type while satisfying the requirement of another. The above factors require consideration of FPGA heterogeneity and design non-planarity to generate a floorplan which meets all the timing constraints. Section 4.1 formally describes the floorplanning problem, which is followed by a description of the overall flow and the algorithms used for generating a floorplan.

4.1 FPGA Flooplanning Problem

The floorplanning problem can be formally stated as follows: Given a set of n modules indicated by M_i, $i = 1, 2, .., n$ with a resource requirement vector $\phi_i = \langle nc_i, nr_i, nd_i \rangle$, which specifies that the module M_i requires nc_i CLBs, nr_i RAMs, and nd_i DSPs, a floorplan F is represented by a set $F = \{P_1, P_2, ..., P_n\}$, defining a non-overlapping 2-dimensional placement P_i of all the modules. The placement of every module P_i is represented by a 4-tuple vector $\langle x_0, y_0, x_1, y_1 \rangle$, where (x_0, y_0) denotes the bottom-left coordinate and (x_1, y_1) the top-right coordinate. The target FPGA architecture is represented by a grid of resources with RPM cartesian coordinates such that the bottom-left corner is (0,0) and the top-right corner is (W,H) where W is width and H the height of the target architecture. The width is the maximum RPM x-coordinate, and the height is the maximum y-coordinate.

The floorplan of a partial reconfigurable design is feasible if:
(i) No two modules overlap in any configuration frame.
(ii) F satisfies the fixed-outline constraints of the device.
(iii) F satisfies the resource constraints for each module.
(iv) There exists a placement of bus macros as port interfaces which satisfies the timing constraints.

4.2 Proposed Overall Flow

The first step in the FPGA flow is design entry, generally using an HDL for the target application. The information necessary to automatically floorplan the design has to be extracted from either the HDL or the netlist file obtained by hierarchically synthesizing the design. The floorplanner requires three kinds of information:

1. *module list* — a list of module instances in the design,
2. *interconnection list* — a representation of the interconnections between modules,
3. *resource requirement vector* — count of resources required for each module.

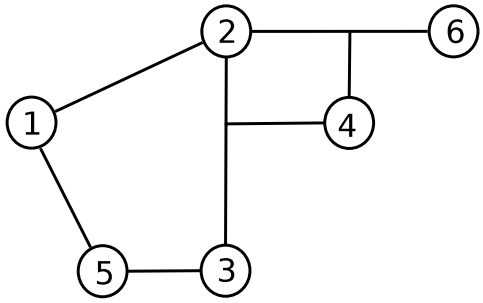


Figure 4.1: A hypergraph

The automatic floorplanner uses a hypergraph data structure to represent a design. A hypergraph is a generalization of a graph where an edge can connect any number of vertices, and is preferred over a normal graph structure because fanout(s) can be expressed in a natural way using hyperedges. Formally, a hypergraph is defined as $H = (V, E^H)$, where V is the set of vertices and E^H is a set of hyperedges. Each hyperedge is a subset of the vertex set, the size of which is defined by the number of elements in the subset. The module instances in the design correspond to the vertices in the hypergraph and the interconnections between the modules are modeled as hyperedges. Figure 4.1 shows an example hypergraph where numbers represent the modules in the design. The edge connecting modules 2, 4 and 6 is an example of a hyperedge connecting more than two vertices which physically translates to a wire connecting three modules.

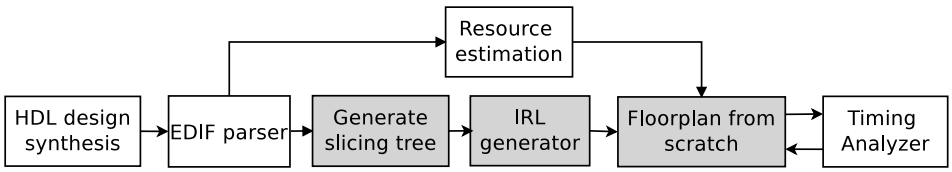


Figure 4.2: Automatic floorplanning flow

The first step in the automatic floorplanning flow, as shown in Figure 4.2, generates a hypergraph from the synthesized design with modules as vertices and interconnections as hyperedges. The hypergraph obtained is recursively bisected using the hMetis [20] tool to create a slicing tree structure, whose leaves correspond to the modules present in the design. The last step before assigning location constraints is to generate an IRL for every module. IRLs are lists of all possible implementations of the module on the target FPGA. Finally the floorplan for the design is generated by traversing the slicing tree in depth-first fashion and assigning location constraints to leaves of the tree which correspond to the modules in the design. The floorplan needs to be timing verified, which means the static routes need to satisfy all the timing constraints, before the floorplan can be deemed valid. Each of these stages — netlist extraction, slicing tree generation, IRL generation and floorplan generation — are described in Sections 4.3 through 4.6 along with the algorithms used.

4.3 Netlist Extraction

The automatic floorplanner, as explained in the Section 4.2, requires information about module interconnections and resource requirements, which is obtained from the netlist file generated by synthesizing the design. A netlist file is a collection of nets specifying the source and sinks of a signal. It is important to note that the netlist file needs to be hierarchical as opposed to a flat netlist because the latter does not contain module instance information which is essential for floorplanning. The floorplanner represents the hypergraph using an interconnect list, which is a list of hyperedges along with the vertices. The data structure capturing the interconnect list consists of two arrays: the first array, `eptr`, is of size `NumberOfHyperedges+1`, and is used to index the second array `eind` that stores the actual hyperedges. Each hyperedge is stored in `eind` as a sequence of the vertices spanned in consec-

utive locations. Specifically, the ith hyperedge is stored starting at location `eind[eptr[i]]` upto (but not including) `eind[eptr[i+1]]`. The size of the array `eind` depends on the number of hyperedges. The resource requirement vector, a vector of 3-tuples $(n_{CLB}, n_{RAM}, n_{DSP})$, is used to represent the resources needed by each module. The tuples of the vector indicate the number of CLBs, BRAMs and DSPs, in that order, required by each module.

The synthesized NGC file is a proprietary format and cannot be parsed. Hence the NGC file needs to be converted to EDIF, which is a vendor-neutral, human readable format to store netlists. The EDIF file contains all the necessary information about the interconnect list, including names of module instances, and the modules (vertices) spanned by a net (hyperedges). The NGC obtained after synthesis is converted to EDIF using the `ngc2edif` utility. A recursive descent parser was written by the author based on the EDIF grammar to extract the interconnect list with the module names and their interconnections indicating the connectivity of each module. After generation of the interconnect list, resource estimation determines the requirements of each module instance. Accurate estimation is essential because over-estimation tends to waste chip area whereas under-estimation increases the place and route times or might lead to timing challenges. The PATIS floorplanner uses the resource estimation tool described in [21], which is used by the Xilinx manual floorplanning tool PlanAhead.

The module names obtained from the EDIF parser are used to uniquely identify individual instances and extract the corresponding resource information. The first step in resource information extraction is to create a PlanAhead project with the NGC file. Tcl scripts are automatically generated for each module using the PlanAhead project created in the previous step. The output of each of the scripts is a text file with resource requirements (number of slices, BRAMs and DSPs) of the corresponding module instance. The resource estimator works in two modes: when the design is floorplanned for the first time and the

estimate is required for all modules, and when the design is incrementally floorplanned and the estimate is required for only for the modules modified.

4.4 Slicing Tree Generation

The primary aim of floorplanning is to place interconnected modules as close as possible to minimize routing delays. The placement has to consider all the modules simulataneously, as optimizing for one module can worsen the delays for other modules. This is undesirable since the routing delay problem involves minimizing the maximum delay because the frequency of operation depends on the longest delay path in the design. There are many approaches to solve this problem, including simulated annealing to randomly attempts many placements and select the best among them. Unfortunately this approach consumes a great deal of time as it takes many iterations to converge to a good solution. Another possibility is to convert the hypergraph into an adjacency graph and generate its rectangular dual [22]. A rectangular dual of a graph G has two properties: (i) every vertex in G corresponds to a distinct rectangle in the dual, and (ii) an edge in G translates to adjacency in the dual. This method can generate efficient floorplans, but a disadvantage is the conditions that need to be satisfied for a dual of a graph to exist [22]. Another approach, which is used in the PATIS floorplanner, partitions the hypergraph obtained from the EDIF parser into sub-hypergraphs and then recursively solves the smaller sub-problems. The recursively bisected hypergraph can be converted into a slicing tree structure where closely connected modules are adjacent leaf nodes in the tree. The slicing tree is then used to generate the floorplan. hMetis [20], a high quality hypergraph partitioning tool, is used for this purpose. The following subsections give a brief introduction to hMetis and its wrapper.

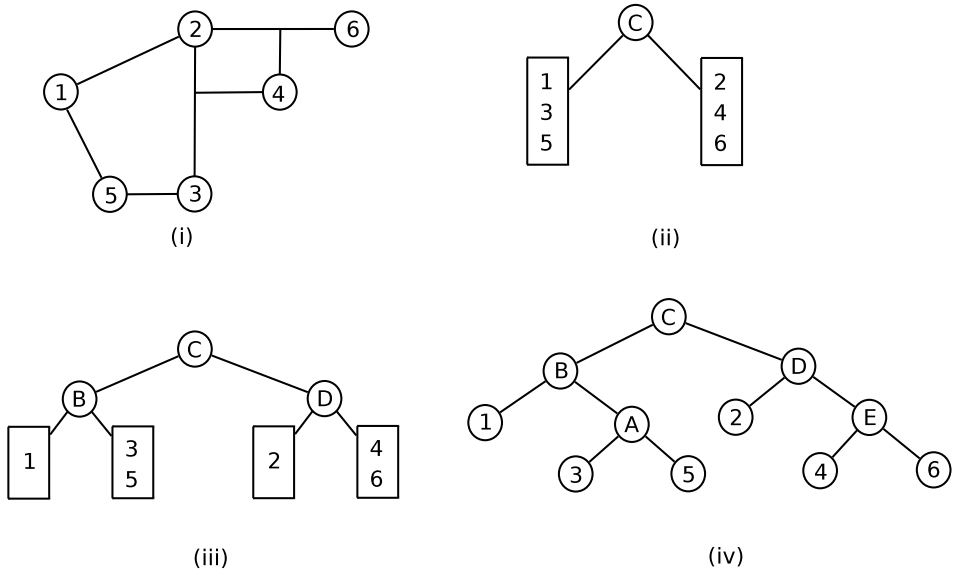
4.4.1 Overview of hMetis

hMetis is a software package developed to efficiently partition large hypergraphs. Traditional graph partitioning algorithms work on the original graph which can take a lot of time to generate an efficient partition. hMetis, on the other hand, takes a multilevel approach to partitioning, reducing the size of the hypergraph during the *coarsening* stage by collapsing the vertices and hyperedges, partitions the smaller hypergraph, and *uncoarsens* it to generate the partition for the original hypergraph. The objective of the coarsening phase is to reduce the size of the hypergraph in such a way that a good bisection on the smaller hypergraph is not significantly worse when compared to a bisection of the entire hypergraph. The refining (uncoarsening) phase focuses on the vertices at the partition boundary to improve the partition quality. The main advantage of using multilevel partitioning techniques is the reduced run-time of the tool compared to traditional partitioners. hMetis allows k-way partitioning of the hypergraph — partitioning the original hypergraph into k parts — through multilevel recursive bisection. hMetis associates each hyperedge with a weight. A higher weight implies that either the hyperedge is critical or the number of hyperedges between the connected modules are large. Weight-based partitioning aids in satisfying the timing constraints, as hyperedges with higher weights are given preference to remain uncut. Section 4.4.2 explains the generation of the slicing tree using hMetis.

4.4.2 hMetis Wrapper

PATIS uses two-way partitioning of the hypergraph to generate the slicing tree. The initial hypergraph is partitioned into two parts with roughly the same number of vertices, while minimizing the number of hyperedges cut. Each sub-hypergraph is bisected again. This continues until each sub-hypergraph has two or fewer vertices. However, bisection of the

hypergraph using hMetis has one limitation — the number of vertices in the sub-hypergraphs cannot be made equal. The difference in the number of vertices between the two sub-hypergraphs is defined by the imbalance factor. Equal partitioning helps to generate a uniform floorplan and hence the hMetis wrapper balances the number of vertices in each of the sub-hypergraphs. This is done by repeatedly selecting the least-connected vertex V in the sub-partition with more vertices and moving V over the boundary until the partitions are balanced.



(i) Input hypergraph (ii) after first bisection (iii) after second bisection
(iv) hMetis-generated slicing tree

Figure 4.3: Generation of slicing tree from hypergraph

Figure 4.3 shows the step-by-step process of generating a slicing tree from the hypergraph. The original hypergraph is shown in the Figure 4.3 (i). After the first bisection there are two sub-hypergraphs which are balanced with respect to the number of vertices. It is important to note here that if a hyperedge with k vertices is bisected with n vertices in one sub-hypergraph and $k - n$ in another, the n and $k - n$ vertices form a hyperedge in the respective sub-hypergraphs. After the second bisection, all the sub-hypergraphs have two or fewer vertices

and recursive bisection is stopped. The generation of a slicing tree from the hypergraph is straightforward, with every bisection point corresponding to a node in the tree. When the number of vertices in the sub-hypergraph is one, the vertex is automatically made a leaf. When the count is two, a node is instantiated with both the vertices as child leaf nodes.

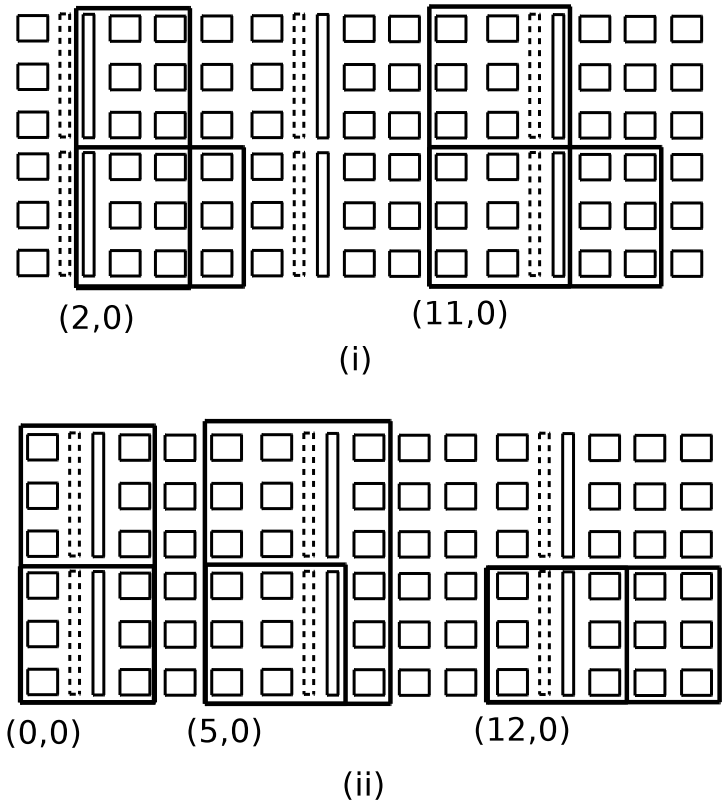
4.5 IRL Generation

4.5.1 Overview of IRLs

The slicing tree gives an indication of which modules should be placed close to each other. By traversing the slicing tree, the floorplanner assigns location constraints to each module. PATIS uses IRLs [12] to generate a list of all possible implementations of a module on the target FPGA. During the slicing tree traversal, this list is scanned to select the best possible implementation for every module. Each implementation is defined by a 4-element vector $\langle x_0, y_0, x_1, y_1 \rangle$, where (x_0, y_0) denotes the bottom-left coordinate and (x_1, y_1) the top-right coordinate. An implementation i is said to be irreducible if there exists no other implementation i' which has the same starting coordinate as that of i and is dominated by i. An implementation i, represented by $\langle x_0, y_0, x_1, y_1 \rangle$, dominates i', represented by $\langle x_0, y_0, x'_1, y'_1 \rangle$, if $x_1 - x_0 \geq x'_1 - x_0$ and $y_1 - y_0 \geq y'_1 - y_0$. This removes many redundant implementations thereby reducing the size of the list. IRL generation obtains module resource requirement information as explained in Section 4.3.

Along with the resource requirement vector, a representation of the physical layout of the target FPGA is required. The representation of all the resources on the chip is based on the RPM coordinate system so that every resource on the chip can be uniquely identified. The resources corresponding to specific locations are mapped onto a `rpmgrid` containing all the

associated information available. It is also necessary to query the location of all the resources as the target FPGA can vary. The `xdlrc` file, which contains the routing information of the FPGA, is used to generate the `rpmgrid` which is a two-dimensional grid of the RPM coordinates. The resource at the RPM coordinate (x, y) is represented by `rpmgrid[x][y]`.



(i) IRLs for a module at two different coordinates (ii) dominating implementations

Figure 4.4: Dominating implementations

Figure 4.4 (i) shows the different irreducible implementations possible for a module given a starting coordinate. The chip layout in Figure 4.4 (i) does not correspond to an actual FPGA, but is used only for illustration purposes. The figure shows possible implementations of a module with two different initial coordinates, none of them dominating. Multiple implementations from the same coordinate have different aspect ratios, defined as the ratio of

width to height. This is useful in reducing the white space during floorplanning as explained in Section 4.6. Figure 4.4 (ii) shows different implementations dominating one another. The three ways an implementation can dominate another depending on the bounding box height and width:

1. The width of both the bounding boxes are same but one is taller than the other, as is shown at coordinate (0,0) in Figure 4.4.

2. The height of both the bounding boxes are same but one is wider than the other, as is shown at coordinate (12,0) in Figure 4.4.

3. Both the width and height of one bounding box is greater than those of the other, as is shown at coordinate (5,0) in Figure 4.4.

The IRL generation algorithm needs to identify all dominating bounding boxes and remove them from the list. Minimizing the size of the IRL reduces the complexity of the algorithm choosing the best implementation. Section 4.5.2 describes an algorithm to generate an optimal list of implementations for a given module on the target FPGA.

4.5.2 IRL Generation

As explained in Section 4.5.1, the generation of IRLs is the first optimzation step in the creation of a new floorplan. The list of possible implementations must be minimized to reduce the complexity of later floorplan generation stages. PATIS considers and satisfies three main resource types that are present on the latest generation FPGAs: CLBs, BRAMs and DSPs. The RPM coordinates used in the floorplanner point to atomic resources on the target FPGA; however, CLBs are not atomic but are comprised of slices. Hence slice count is used to represent logic needed for a module, which is advantageous because the resource estimator uses a similar represention.

PR implementation of a design introduces rules regarding location constraints of modules, the primary rule being no two modules can share a configuration frame. In Virtex-4, a configuration frame is 16 CLBs in height, making that the minimum height of any module. In theory a module can have height less than one configuration frame but the rest of the resources in that configuration frame cannot be used. Hence for effective utilization of the resources on the FPGA it is necessary to have integral multiples of the configuration frame as the height of any implementation.

Another important condition involves limiting the aspect ratio to a specific range in order to reduce the difference between the height and width of any implementation. Consider the implementation in Figure 4.5 and suppose a module in the design requires 24 CLBs. The layout shown in the figure assumes the height of the configuration frame is 3 CLBs. Two possible implementations are as shown in Figure 4.5: the width of the first implementation is that of the chip and large when compared to the height, while the width of the second implementation is comparable to the height. Generating a slicing floorplan using the first implementation is more difficult when compared to the second implementation because no white space is available on three of its sides. This restriction allows the floorplan to grow in only one direction.

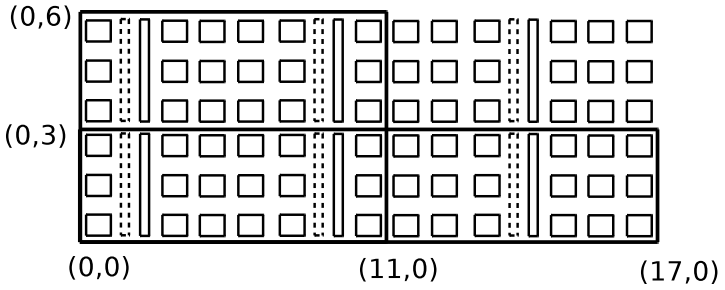


Figure 4.5: Implementations with different aspect ratios

Given an initial coordinate, the algorithm used to generate IRLs is shown in Figure 4.6. Important functions used in the algorithm are explained below:

```
1:  Procedure: GenerateIRLS()
2:
3:  height = 0;
4:  while (height < maxAllowedHeight) do
5:      primSite = rpmGrid[x0][y0];
6:      noImplPoss = 0;
7:      x0 = getNextValidSite(primSite);
8:      {reqSliceCount, reqBramCount, reqDspCount} = getResourceRequired();
        /* get reqd column count (height dependent)                          */
9:      numSliceCols = reqSliceCount/nslice_configframe * h;
10:     x1 = getResourceRight(sliceType, numSliceCols);
11:     grid = setBoundaries(x0, y0, x1, y0 + height);
12:     {sliceCount, bramCount, dspCount} = getResourceCount(grid);
13:     if (sliceCount < reqdSliceCount) then
14:         noImplPoss = 1;
15:         continue with new height;
16:     if (bramCount < reqdBramCount) then
17:         numBramCols = (reqBramCount − bramCount)/nslice_configframe * h;
18:         x1 = getResourceRight(bramType, numBramCol);
19:         grid = setBoundaries(x0, y0, x1, y0 + height);
20:         {sliceCount, bramCount, dspCount} = getResourceCount(grid);
21:         if (bramCnt < reqdBramCount) then
22:             noImplPoss = 1;
23:             continue with new height;
24:     if (dspCount < reqdDsp) then
25:         numDspCols = (reqDspCount − dspCount)/nslice_configframe * h;
26:         x1 = getResourceRight(dspType, numDspCols);
27:         grid = setGridBoundaries(x0, y0, x1, y0 + height);
28:         {sliceCount, bramCount, dspCount} = getResourceCount(grid);
29:         if (dspCount < reqdDspCount) then
30:             noImplPoss = 1;
31:             continue with new height;
32:     if (noImplPoss = 0) then
33:         insertImpl(grid);
34:     update(height);
```

Figure 4.6: IRL generation algorithm

`getNextValidSite`: This function takes a `primSite`, a coordinate on the RPM grid, and returns the nearest valid site in the specified direction. The `primSite` (line 5) is defined as a RPM coordinate. This is essential because the RPM grid is not continuous and pointing to an invalid site returns an exception.

`getResourceRight`: To satisfy a module's resource requirements, PATIS uses a resource column as the basic unit. The reason for this is two fold: the height of the implementation is fixed, and an FPGA column vector contains resources of the same type. A resource column is defined as a vector of resources of height h, which is the height of the implementation. Resource columns are defined differently for different resource types. In case of an implementation with height h, the number of resource columns required is defined as

$$ncols_{r1} = nreqd_{r1}/(nr1_{configframe} * h). \tag{4.1}$$

where $r1$ denotes a particular resource type, $nr1_{configframe}$ is the number of resources of type $r1$ which make up a configuration frame, and $nreqd_{r1}$ is the required resource count of type $r1$ obtained from the resource requirement vector. This function takes in a starting `primSite` and traverses the RPM grid by $ncols_{r1}$ columns of resource type $r1$.

Given an initial RPM coordinate `primSite`, the algorithm first checks for its validity. If `primSite` is deemed invalid, the `getNextValidSite` function is used to obtain the next valid site. The next step is to obtain the resource count from the resource requirement vector of that module, which is shown in line 8. The requirement for each type of resource obtained in the previous step is satisfied sequentially. The order in which the resource requirements are satisfied was decided based on the density of the resource on the FPGA, from most dense to least dense. In many implementations, it was seen that satisfying the slice count generally satisfies all the other resource requirements. As seen from line 12, the slice requirement is satisfied first, followed by BRAM (line 16) and DSP (line 24). For every resource type, the number of resource columns required is computed and satisfied using the `getResourceRight` function.

The bounding box is represented as a `grid` with bottom-left and top-right coordinates. The `grid` boundaries are updated whenever the resource requirement for any resource type is

satisfied. For example once the slice requirement is satisfied, the grid is updated as shown in line 19. If any resource requirement is not satisfied, module implementation is impossible at the given initial coordinate and for the given height. The IRL is updated with the current implementation if all the requirements are satisfied and no existing implementation in the IRL is dominated by the new one. The algorithm shown in Figure 4.6 generates an implementation given a starting coordinate and height. For every starting coordinate, the height is incremented to generate a new implementation as long as the aspect ratio is within an acceptable range.

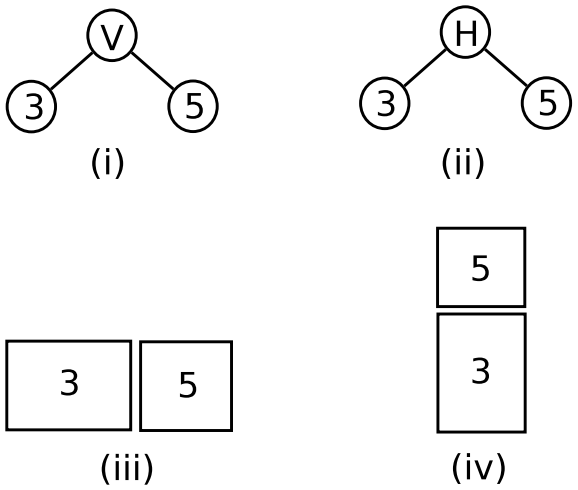
4.6 Floorplan Generation

The slicing tree and the IRLs generated are used in this step to assign location constraints to each of the modules. The slicing tree is traversed in depth-first order. Whenever a leaf node (which represents a module) is encountered, the IRL for the module is scanned to identify the best implementation given the starting coordinates. After traversing through all leaf nodes and constraining the respective modules, the validity of the floorplan is verified. Figure 4.7 shows the algorithm which details the steps involved in generating an optimized floorplan. This section explains the algorithm flow elaborating on the design decisions and optimizations.

As explained in Section 4.4, the slicing tree places interconnected modules as adjacent leaves. However, this does not give explicit information on the relative placement of the modules which is essential during floorplanning. This additional information is defined by specifying a cut direction at each node in the slicing tree. The cut direction on any parent module indicates how the child modules are separated. A vertical cut implies a vertical boundary between the child modules, whereas a horizontal cut implies a horizontal boundary. Figure 4.8 shows how the modules are separated in case of different cut directions.

```
1:  Procedure: Floorplanner(node)
2:
3:  x_0 = 0; y_0 = 0;
4:  nodeLeft = node → left;
5:  nodeRight = node → right;
6:  if (nodeLeft ≠ NULL) then
7:      Floorplanner(nodeLeft);
8:  if (nodeRight ≠ NULL) then
9:      Floorplanner(nodeRight);
10: if (state = 0) then                                  /* both child nodes are leaves */
11:     selectMode = 1 ;                                    /* unity aspect ratio mode */
12:     getIRL(nodeLeft, x_0, y_0, selectMode);
13:     if (cutDirection = horizontal) then
14:         update(y_0);
            /* horizontal cut mode, width mode                                          */
15:         selectMode = 2;
16:         getIRL(nodeRight, x_0, y_0, selectMode);
17:     if (cutDirection = vertical) then
18:         update(x_0);
            /* vertical cut mode, height mode                                           */
19:         selectMode = 3;
20:         getIRL(nodeRight, x_0, y_0, selectMode);
21: if (state = 1) then                                  /* only one child node leaf */
22:     if (nodeLeft = leaf) then
23:         if (cutDirection = horizontal) then
24:             selectMode = 2;
25:             getIRL(nodeLeft, x_0, y_0, selectMode);
26:         if (cutDirection = vertical) then
27:             selectMode = 3;
28:             getIRL(nodeLeft, x_0, y_0, selectMode);
29:     if (nodeRight = leaf) then
30:         if (cutDirection = horizontal) then
31:             selectMode = 2;
32:             getIRL(nodeRight, x_0, y_0, selectMode);
33:             if (cutDirection = vertical) then
34:                 selectMode = 3;
35:                 getIRL(nodeRight, x_0, y_0, selectMode);
36: if (state = 2) then                                   /* both are cluster nodes */
37:     verify validity of floorplan up to this node;
38: update(x_0, y_0);
```

Figure 4.7: Floorplan generation algorithm



(i) Part of slicing tree with a vertical cut (ii) part of slicing tree with a vertical cut (iii) vertical cut between two modules (iv) horizontal cut between two modules

Figure 4.8: Cut directions

Assigning cut directions for all the nodes in the slicing tree can be done in two ways: (i) trying different combinations of cut directions in a bottom-up approach or (ii) alternately assigning horizontal and vertical cut directions and modifying when required. The former is similar to simulated annealing which assigns cut directions to nodes and location constraints to modules in one step, while the latter is a direct approach. PATIS uses the second technique to assign cut directions mainly because IRLs generated for every module give an optimized placement, thereby removing the need for a stochastic approach. In addition, a uniform floorplan can be obtained by alternating horizontal and vertical cuts in the slicing tree. Figure 4.9 shows the slicing tree after assigning cut directions for the example from Figure 4.3.

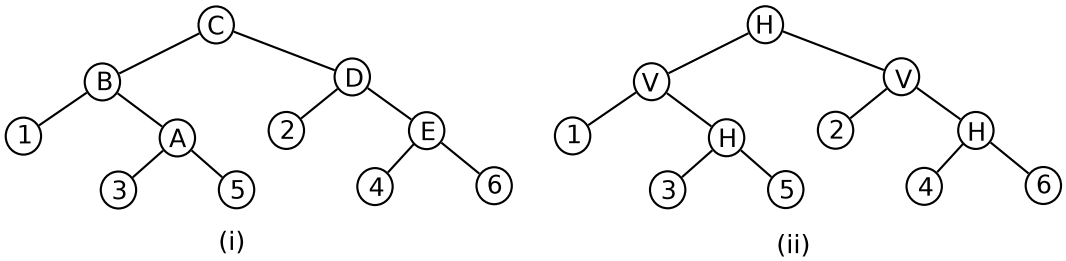
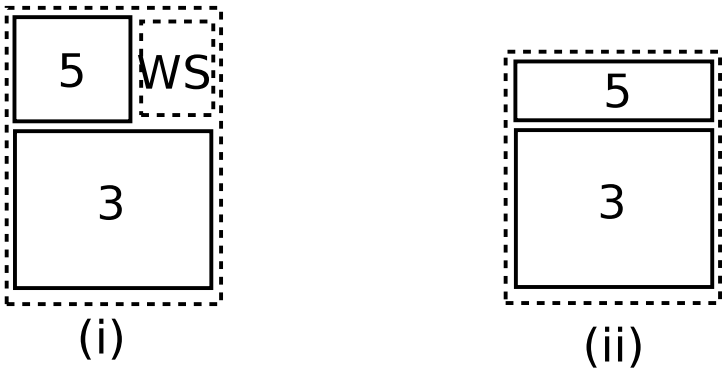


Figure 4.9: Slicing tree with alternate horizontal and vertical cut directions

The next step in floorplanning is to traverse the slicing tree with the cut directions assigned and place the module instances on the FPGA. There are two essential tasks in this step: (i) assign the location constraints for all the modules, and (ii) clustering the location constraints of the child nodes to generate the constraints of the parent node. In PATIS both tasks are done in a single depth-first traversal of the slicing tree. A depth-first traversal is one that progresses by visiting the first child node of the tree and visiting its first child node until a node that has no children is reached. The search then backtracks, returning to the most recent unvisited node. To simplify the implementation, the left child node of the slicing tree is always explored before the right.

There are two approaches to generate a slicing floorplan from a slicing tree: a top-down approach where the layout is partitioned initially and then realigned to satisfy the resource requirements; and a bottom-up approach which places the modules satisfying the resource constraints and clusters them to generate the complete floorplan. In PATIS, the IRLs generated are used to place modules in a bottom-up approach. Especially for slicing floorplans where adjacent leaves are placed based on the cut direction, it is possible to have white spaces which reduce the chip utilization factor. Figure 4.10 (i) shows an example of module placement with white spaces in case of a horizontal cut. Difference in width (in case of a horizontal cut) or height (in case of a vertical cut) of modules placed next to each other is the main reason for white space. In this example, the white space can be reduced by selecting an implementation of `B` whose width is closest to that of `A`. Figure 4.10 (ii) shows an implementation of the same modules `A` and `B` without any white spaces.

PATIS implements the above concept by generating constraints for both the child modules together at the parent node. As shown in Figure 4.11, three possibilities arise at every node while assigning location constraints: (i) both the child nodes are leaves (Figure 4.7, line 10), (ii) one is a leaf and the other is a cluster node (line 21), and (iii) both are cluster nodes (line



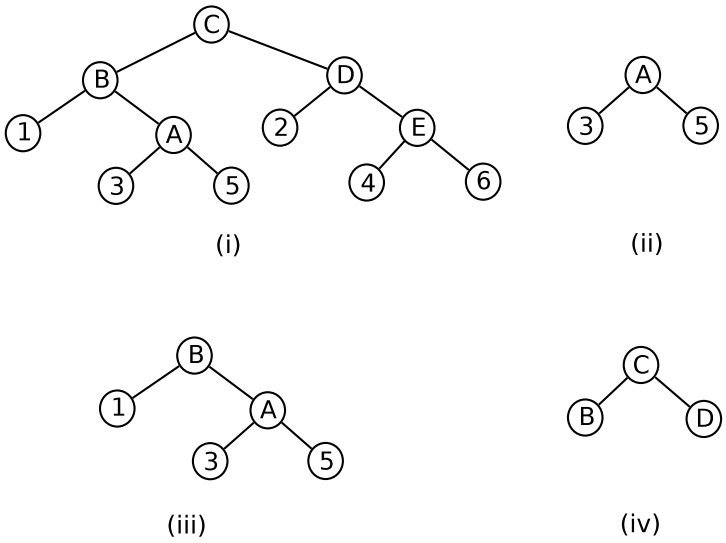
(i) White space (WS) in case of vertical cut (ii) implementation without white space

Figure 4.10: Implementation with and without white space

36). Any node which is not a leaf in the slicing tree is classified as a cluster node because the constraints for this node are obtained by clustering the location constraints of the two child nodes. In order to generate an optimal floorplan, each of the cases described above are handled separately by using different modes for selecting a suitable implementation. The unity aspect ratio mode chooses an implementation from the IRL with minimal difference between height and width. The width mode selects an implementation whose width is closest to the specified width, and the height mode selects an implementation whose height is closest to the specified height.

Case (i) : In this case, both the child nodes are modules and need to be assigned location constraints. The left child is placed first as shown in line 12, by selecting an implementation closest to unity aspect ratio. The placement of the second module is dependent on the direction of the cut; for a vertical cut (line 17) an implementation whose height is equal to the first module is chosen as shown in line 20. A similar decision based on width is made in case of a horizontal cut (line 13). The reason for making such a choice is to minimize the white space during clustering as shown in Figure 4.10.

Case (ii) : This case can be better understood from Figure 4.11 (i). For node `B`, the left child

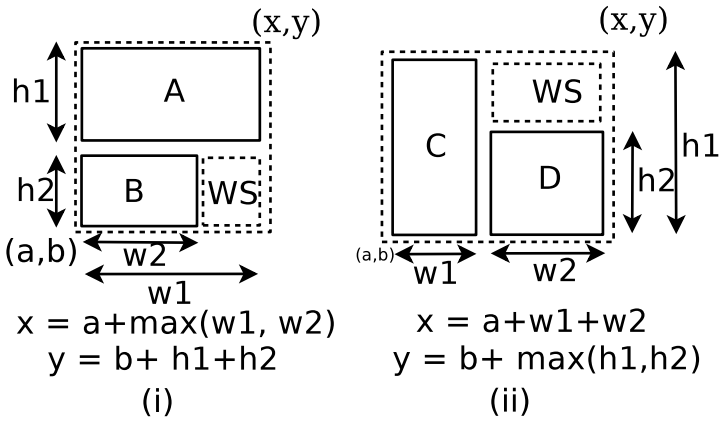


(i) Original slicing tree (ii) parent node with two child nodes both leaf nodes (modules) (iii) parent node with one child a leaf and other a cluster node (iv) parent node with both child nodes as cluster nodes

Figure 4.11: Parent nodes at different levels in slicing tree

is a module but the right side is a clustered node `A`. The location constraints for the clustered node `A` is obtained as explained in case (i) by clustering the constraints corresponding to modules `3` and `5`. The leaf node (module 1) needs to be placed and since the cut direction at node `B` is vertical, the height of the implementation is fixed as the height of the clustered node `A`.

Case (iii) : In this case the constraints of the child nodes are clustered to get the bounding box of the parent node. Clustering, in case of both vertical and horizontal cut directions, is as shown in Figure 4.12. The clustered boundary completely encompasses the contained modules, in this case modules `A` and `B`. In case of a vertical cut, as shown in Figure 4.12 (ii), the width is the sum of the individual widths of the modules while the height is the maximum of the individual heights of the modules. It is clear that the difference in height or width between modules placed together is the primary reason for white space in floorplans.

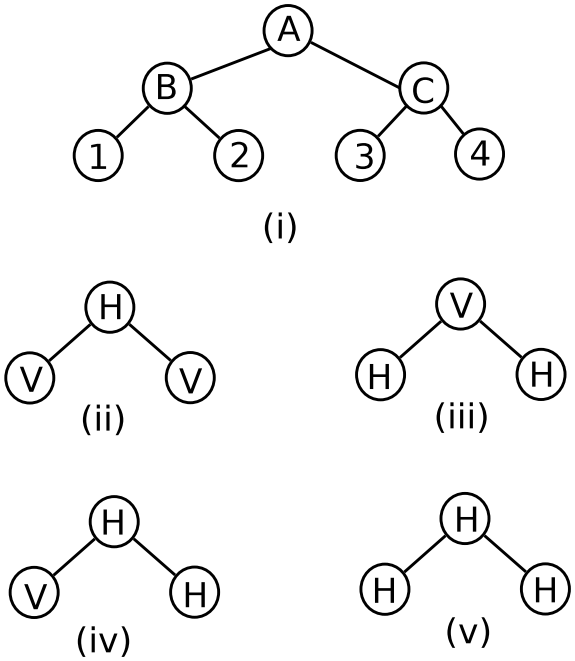


(i) White spaces (WS) in case of vertical cut (ii) white space in case of horizontal cut

Figure 4.12: White space in case of different cut directions

When assigning location constraints to modules, it is possible to encounter a situation where a module has no implementation given an RPM coordinate. PATIS tackles this problem by modifying the cut direction of the parent nodes and re-floorplanning the child modules. Cut directions of the parent nodes are modified in reverse order of the depth-first traversal until either the root node is reached or a valid floorplan is obtained. If no combination of the cut direction yields a valid floorplan, a new slicing tree is obtained from hMetis to generate a new floorplan.

Figure 4.13 shows the different modifications in cut direction possible for a parent node in case of an invalid floorplan. In order to locally optimize the floorplan, it is necessary to modify the cut direction of the child nodes along with the parent nodes. Though all the possible combinations are considered, those satisfying resource requirements with minimal asymmetry are preferred. For instance, consider a fragment of a slicing tree and the corresponding initial cut directions as shown in Figure 4.13 (i) and (ii). For illustration purposes, assume that module `3` has no implementation for the given RPM coordinates. PATIS tries to generate a valid floorplan by modifying the cut direction at node `C` followed by node `A` and so on. The modification order at a general node `A` in Figure 4.13 (i) is as follows:



(i) Slicing tree (ii) cut directions with parent node `A` (iii) flip all cut directions (iv) flip cut direction of failed child (v) flip cut directions of both child nodes

Figure 4.13: Modifications in case of floorplan failure

1. Flip the cut directions of both parent and child nodes as shown in Figure 4.13 (iii). This can be useful for sparse resources such as DSPs because a horizontal cut will allow the resource requirements to be satisfied easily.

2. Modify the cut direction of the child node that failed implementation (assumed to be the module `3`), as shown in 4.13 (iv). The cut direction of other nodes are initially unchanged but flipped individually if required.

3. The cut direction of both child nodes are flipped as shown in Figure 4.13 (v).

The last step in floorplan generation is to create a UCF file from the location constraints of the modules in the design. PlanAhead inputs the bounding constraints for each of the resource type and generates UCF output. The UCF file contains the module name and the

resources contained within the location constraints.

Static timing closure should be verified for every floorplan generated before it is used in the implementation stage. PATIS combines timing verification along with bus macro insertion. Section 3.2.3 explains the bus macro insertion methodology. Bus macros are placed using the PAR tool with all the timing constraints included in the UCF file. If the PAR tool generates a valid set of routes between the bus macros, then the floorplan generated is valid from a timing perspective.

Chapter 5

Results

A benchmark suite was developed to evaluate the performance of implementation tools using the DMD flow. To enable comparison with the standard flow, the tools were also run without using PR. To analyze PATIS behavior for any design type, benchmarks with varied resource requirements were chosen. This chapter first introduces the designs in the benchmark suite and their resource compositions. PATIS tool performance is then summarized followed by an analysis of implementation tool run-times.

5.1 Platform Description

Implementation experiments were conducted on a quad-core 2.66 GHz Core i7-920 CPU with at least 6GB of memory. The automatic floorplanning tool was run on a dual-core 1.86 GHz Xeon CPU with at least 3 GB of memory. The automatic floorplanning tool needed a 32-bit OS because the stable hMetis release works well on it. All benchmarks target a Virtex-4 FX100-ff1517 FPGA [23] using Version 9.2i of the ISE tools with the PR-12 patch. Programs were written in C++, and time durations for the tool runs were determined using the C++ utility `gettimeofday()`.

5.2 Benchmark Suite

The PATIS benchmark suite contains a variety of designs enumerated below, with different resource requirement vectors and chip utilizations. In these designs, the clock frequency constraints are chosen to make timing challenging but attainable.

1. **Cascaded complex FFTs**: A Fast Fourier Transform(FFT) [24] is an algorithm to compute the Discrete Fourier Transform (DFT) and its inverse, where the DFT converts from the time domain to the frequency domain. DFTs have extensive applications in spectral analysis and data compression. A hardware module implementing an FFT is compute-intensive and can be implemented using DSPs. However, the limited number of DSP slices on the Virtex-4 FX100 chip were quickly exhausted, and hence the FFT computations were implemented with logic slices leading to a high CLB utilization. This design consists of several complex FFT blocks cascaded in a ring topology, with two 16-bit buses between each pair. Alternate modules in the cascade compute the FFT and the inverse FFT of the input data, with a `ready` input line activating each module only when the input is ready. Two versions of this design were used for experimentation. One used 3 CFFT modules, while the other contained 6. The three-module design `CFFT 3`, shown in Figure 5.1, is the smallest in our benchmark suite with a chip area utilization of almost one-sixth, while the six-module design `CFFT 6` used about one-third of the FPGA, as illustrated in Figure 5.2.

2. **Multiple MicroBlaze processors**: The MicroBlaze is a soft processor core targeting Xilinx FPGAs [25]. As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric and is both CLB- and memory-intensive. The MicroBlaze may be programmed for custom applications. A MicroBlaze processor was generated using the Xilinx EDK with bidirectional Fast Simplex Link interfaces.

Several instances of the MicroBlaze were connected in a ring topology using Xilinx ISE. Different sizes of this design can be generated by instantiating different numbers of MicroBlaze processors. Figure 5.3 shows the five MicroBlaze system `MB 5` created with a chip utilization of 40%.

3. **A scalable Viterbi decoder**: A Viterbi decoder uses the Viterbi algorithm to decode a bitstream that has been encoded using forward error correction based on a convolutional code. There are several lower resource alternatives to a Viterbi decoder, but the benchmark chosen for this suite does the maximum likelihood decoding. The memory-intensive Viterbi decoder [26] is scalable according to the constraint length for the convolution code. Several modules of differing sizes were implemented by varying the constraint length, and then cascaded to form a ring. The `Viterbi 7` design was implemented with seven modules that occupy approximately 70% of the chip, as shown in Figure 5.4.

4. **FloPoCo implementations of arithmetic expressions**: Floating Point Cores (FloPoCo) [27] is an arithmetic core generator that can implement fixed-point, floating-point and integer expressions. Although FloPoCo supports basic arithmetic operations such as addition, subtraction, multiplication, division and square roots, more complex operations such as logarithms, trigonometry and polynomials are the focus. These operations are performed on DSP slices, with intermediate results stored in the BRAMs. Several data-intensive modules implementing different arithmetic operations were implemented in two variants of this application. One benchmark, `FloPoCo 8`, has eight modules — some large — with 85% chip area utilization. The other benchmark, `FloPoCo 10`, consists of ten modules — several medium-sized — occupying almost 90% of the chip area. The layouts of the `FloPoCo 8` and `FloPoCo 10` designs are shown in Figures 5.5 and 5.6, respectively.

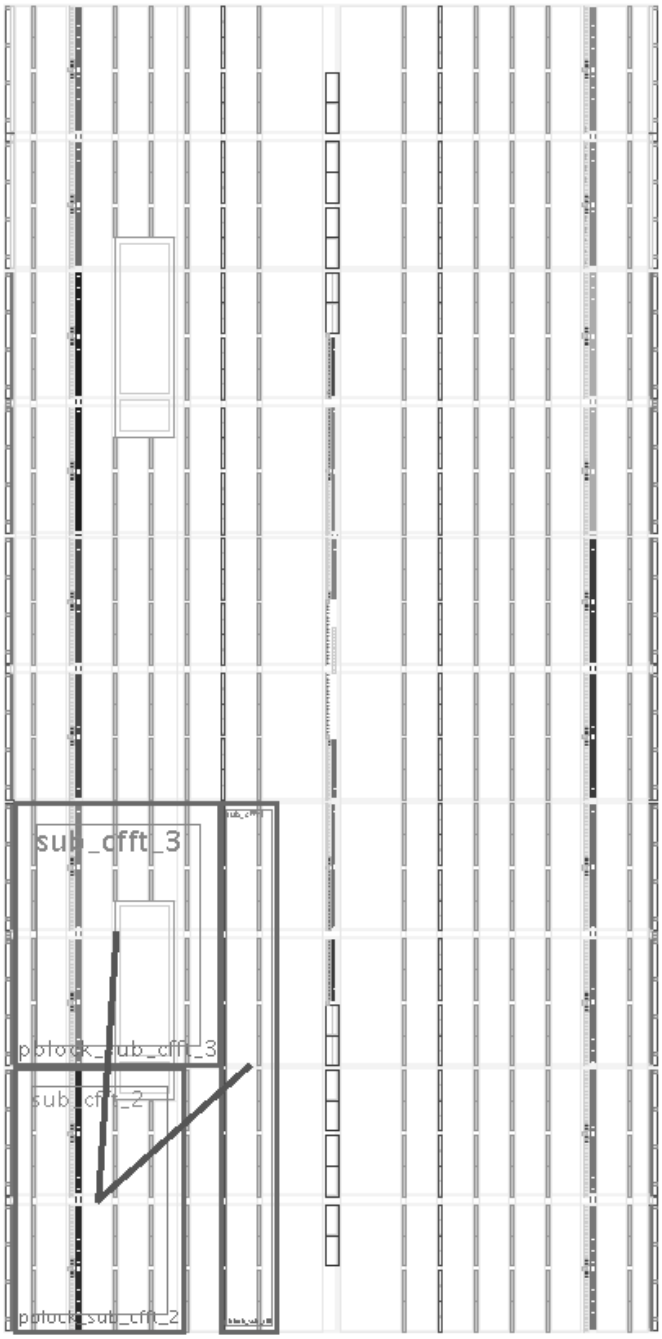


Figure 5.1: Design – `CFFT 3` (*Screenshot captured from PlanAhead tool*)

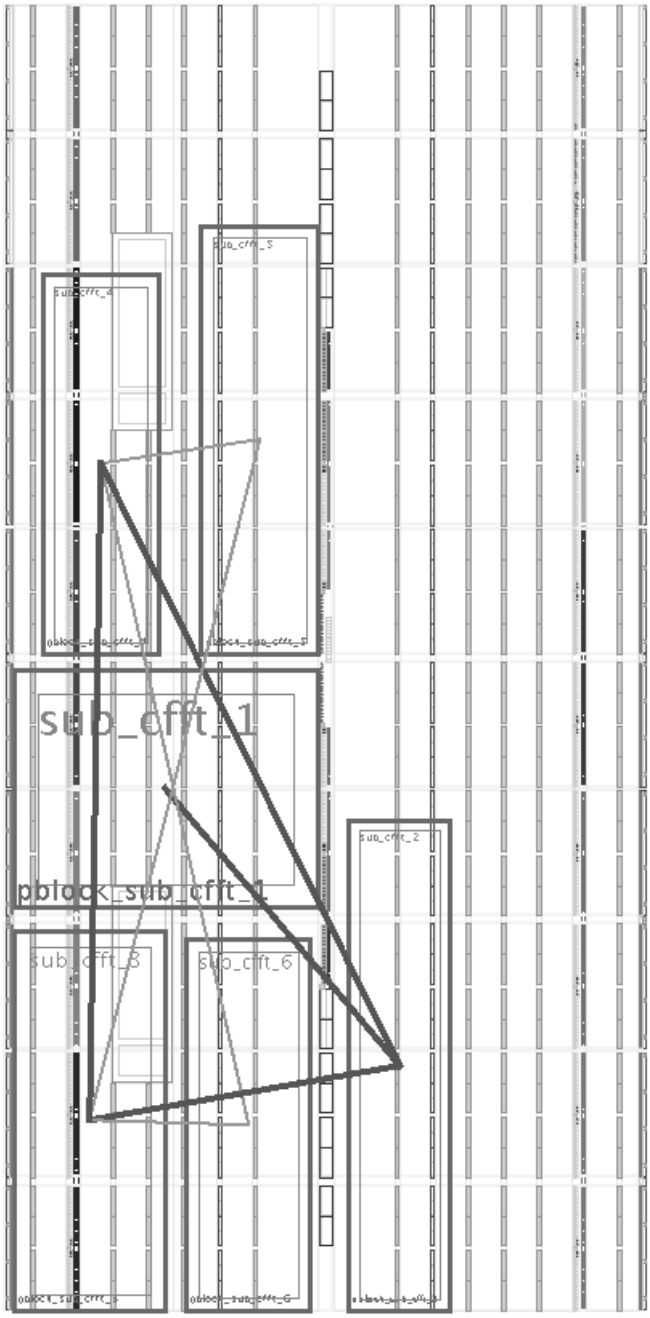


Figure 5.2: Design – `CFFT 6` (*Screenshot captured from PlanAhead tool*)

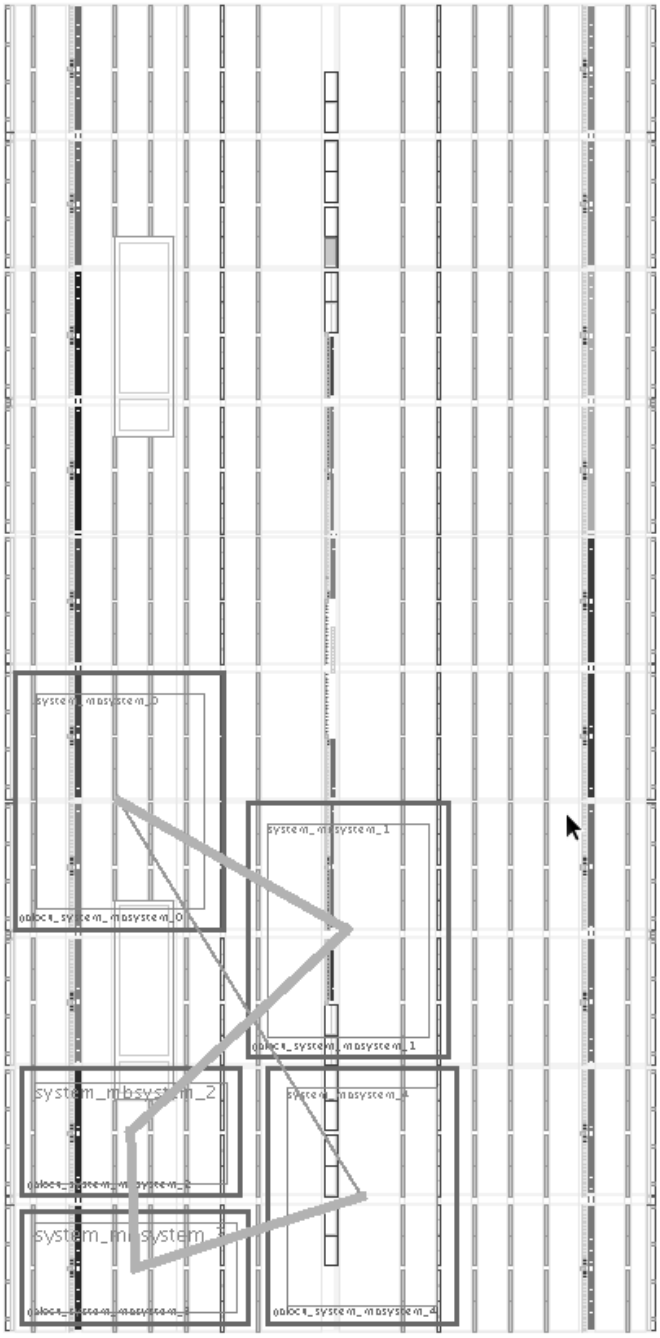


Figure 5.3: Design – `MB 5` (*Screenshot captured from PlanAhead tool*)

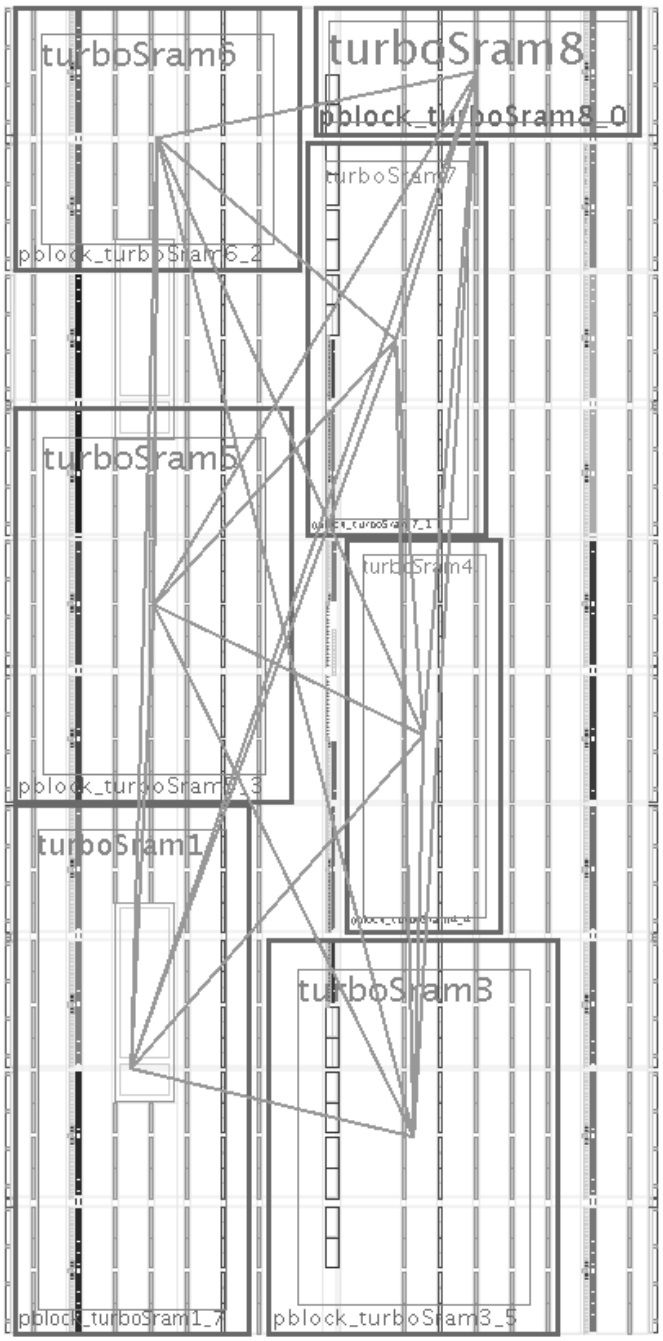


Figure 5.4: Design – `Viterbi 7` (*Screenshot captured from PlanAhead tool*)

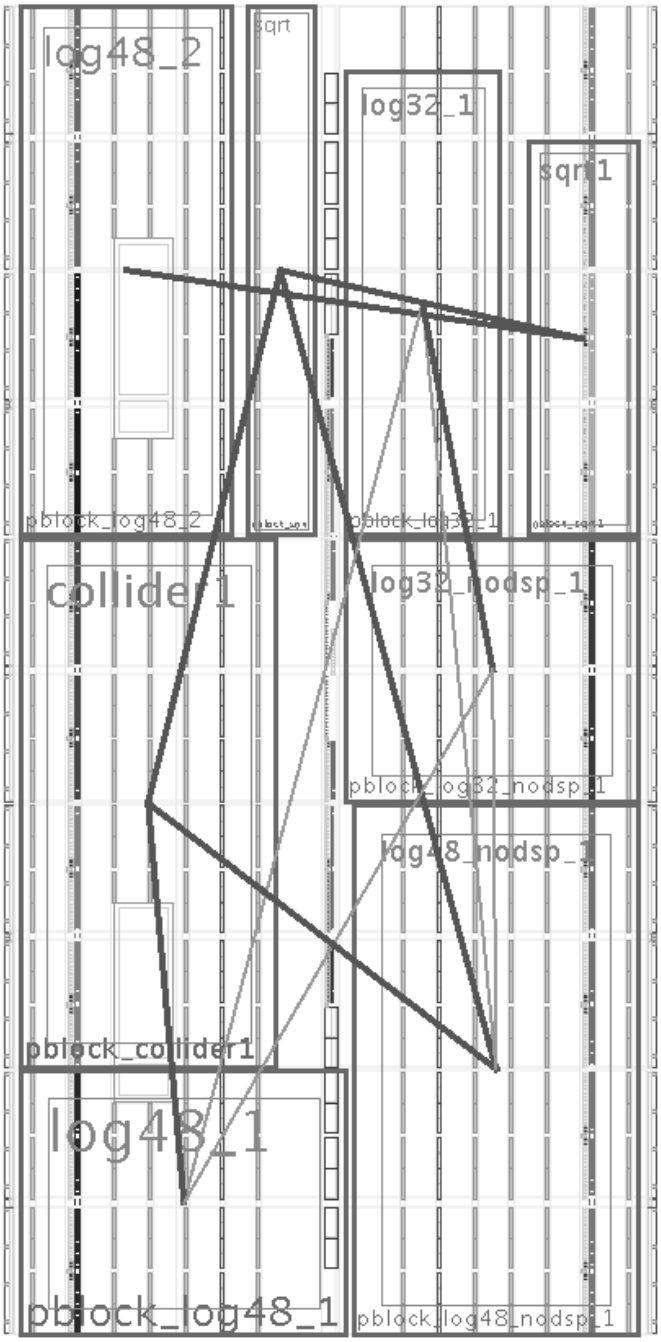


Figure 5.5: Design – `FloPoCo 8` (*Screenshot captured from PlanAhead tool*)

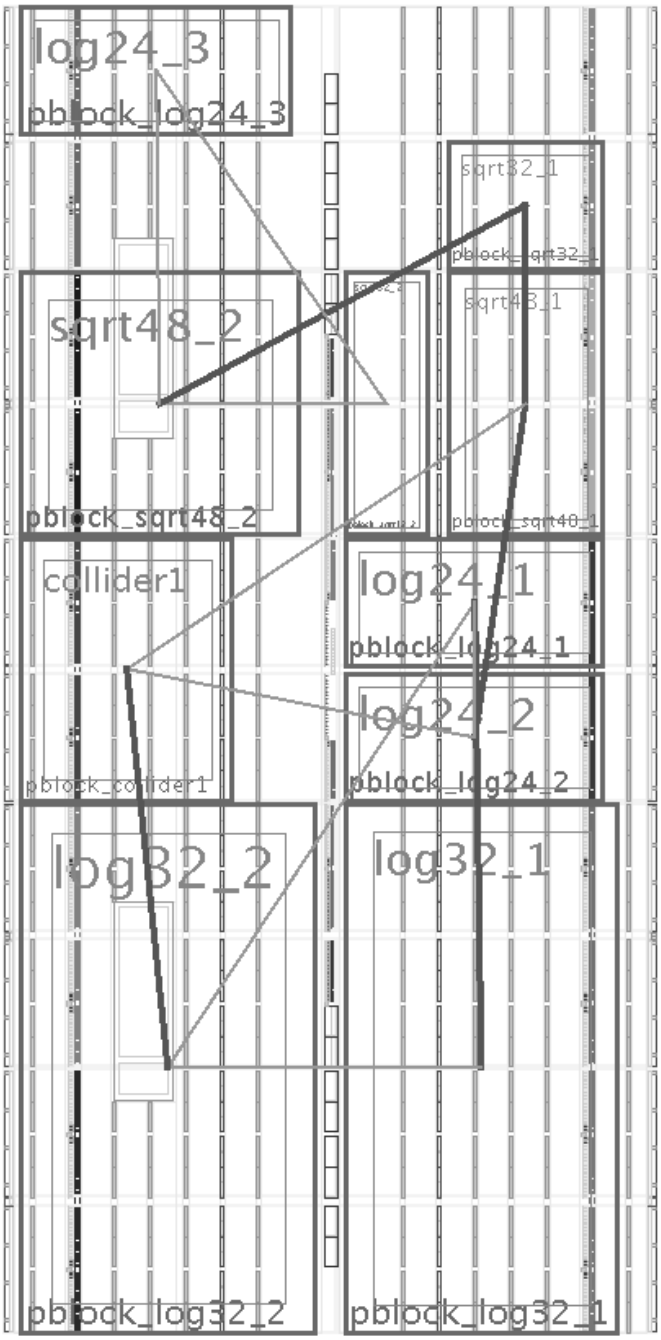


Figure 5.6: Design – `FloPoCo 10` (*Screenshot captured from PlanAhead tool*)

5.3 Automatic Floorplanner Performance

PATIS floorplans a design from scratch when it is implemented for the first time. Automatic floorplanning consists of two steps: the preprocessing phase and the floorplanning phase. The preprocessing phase generates the hypergraph representation of the design and estimates the resources required for every module. Run-time of the preprocessing phase depends on the number of modules (for resource estimation) and the design size (for hypergraph generation). PlanAhead, used for resource estimation, introduces additional overhead in this stage.

The floorplanning phase converts the hypergraph into a slicing tree, generates IRLs for every module, and assigns location constraints. Run-time for IRL generation depends on the module count, module size and size of the target FPGA. The non-intuitive parameter is the module size — the greater the size of the module, the less the number of realizations possible on the chip and hence the time required to generate IRLs is reduced. Depth-first traversal to assign location constraints depends on the number of modules (which decides the number of nodes on the slicing tree) and the number of realizations in the IRLs. The size of the realization list determines the time taken to search for the best module implementation.

Table 5.1: Automatic floorplanner - tool run times (on a 1.86 GHz Xeon CPU)

Design	**Module count**	**Run-time (CPU seconds)**
CFFT 3	3	97.42
MB 5	5	56.12
CFFT 6	6	191.92
Viterbi 7	7	246.45
FloPoCo 8	8	220.75
FloPoCo 10	10	223.52

Table 5.1 shows the run-times of the PATIS automatic floorplanner for each design in the benchmark suite. Pre-processing steps such as EDIF parsing and resource estimation are not

included in the run-time calculation. The time taken for PATIS increases with the number of modules and the size of the device. The former is intuitive because the time-consuming step in PATIS is IRL generation and IRLs are computed for each module. The latter implies an increase in the logic present on the FPGA, causing the number of realizations per module to increase thereby lengthening the run-time of the tool.

5.4 DMD Flow Performance

To evaluate run-time improvements, designs in the benchmark suite were implemented using two flows: (i) the DMD flow, and (ii) the non-PR floorplanned flow. The improvements are computed for both automatic and incremental PATIS floorplanners. Each design is implemented using different flows under the same timing constraints.

Table 5.2: Place-and-Route times after floorplanning

Design	f_{clk} **(MHz)**	**Elapsed PAR time (minutes)**		
		Manual floorplan	**PATIS floorplan**	**PATIS incremental (modules changed)**
CFFT 3	256.40	35	10	5 (2)
MB 5	127.40	80	17	7 (2)
CFFT 6	170.20	175	16	5 (3)
Viterbi 7	44.40	380	18	4 (1)
FloPoCo 8	74.10	120	11	5 (2)
FloPoCo 10	80.32	124	12	4 (1)

Table 5.3: Minimum frequency comparison for manual and PATIS flows

Design	f_{clk} **of manual floorplan (MHz)**	f_{clk} **of PATIS floorplan (MHz)**
Viterbi 7	44.4	71.4
FloPoCo 8	74.1	121.4

The run-times of the tools are as shown in Table 5.2. As explained previously, the frequency of operation is unchanged during the different implementation runs. In the manual floorplanning flow, all modules are assigned area constraints and place-and-route is performed on the design as a whole. PATIS implementation acceleration comes from the parallel reduction of a large global optimization problem to a set of smaller independent problems. Placing and routing an entire design is reduced to optimizing the placement and routing of each module within the area constraints specified. Module logic can be placed and routed in parallel as the respective area constraints are non-intersecting. We generally accept optimization restrictions across module boundaries for the sake of design productivity and timing closure. Although the PATIS/PR flow adds bus macro overheads to inter-module routing, pipelining the interface may improve the system clock frequencies as shown in Table 5.3. When the critical path of the design straddles module boundaries, registering the ports improves the system clock frequency. PATIS takes advantage of this by automatically registering the ports through bus macro interfaces. In two designs, `Viterbi 7` and `FloPoCo 8`, the frequency improvement is significant.

As the design size increases, the run-time of the tools increases drastically in the manual design flow, assuming similar strictness in timing constraints. The `Viterbi 7` design occupies 70% of the chip area takes around 380 minutes to be placed and routed, whereas the `CFFT 3` design occupies 30% takes only 35 minutes. However the difference in the run-times using the DMD flow for the same pair of designs is just 8 minutes. This illustrates the effectiveness of using a divide-and-conquer approach. Figure 5.7 shows the speedup when implementing the design using the DMD flow over the non-PR modular flow.

Speedup obtained using the DMD flow is also dependent on the target system and the available memory bandwidth. The implementation speedup is directly proportional to the number of cores available because concurrent implementation of PR modules is an embar-

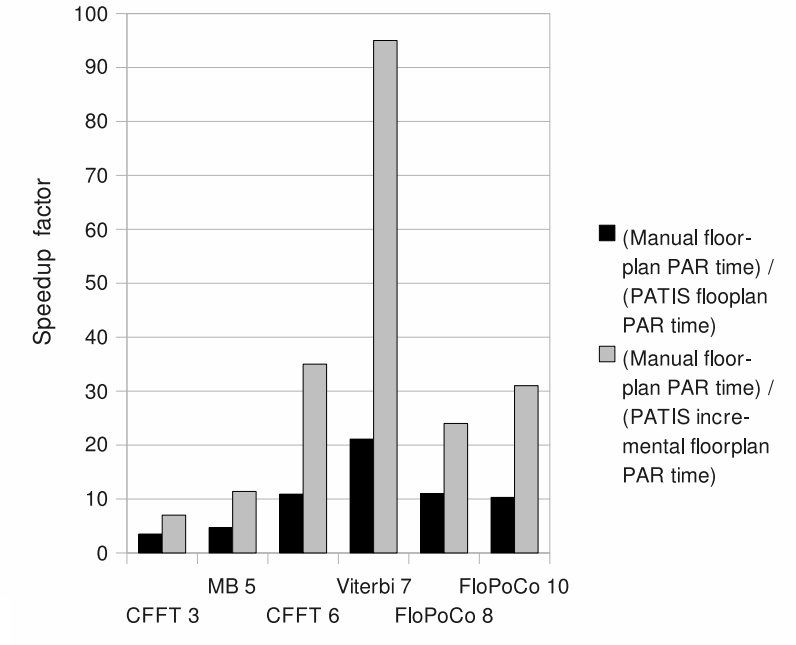


Figure 5.7: PATIS run-time improvements

rassingly parallel problem. Consider an n module design and an m core target system. When $n \leq m$, there is no contention for cores thereby making memory bandwidth the only factor affecting speedup. When $n > m$, the number of independent tasks to be executed exceed the number of cores of available. The resource contention allows only m independent tasks to be performed in parallel. Load balancing can be used to improve the performance in this case, but is not implemented in the DMD flow. Also PAR algorithms use large graph data structures which are memory intensive. Hence, running multiple PAR jobs in parallel requires considerable memory bandwidth for best performance.

The use of PR normally decreases systems performance compared to a non-PR implementation [28]. This was largely attributed to the addition of nets to put PR modules into known states after loading their partial bitstreams since the global set/reset (GSR) signal cannot be asserted. However, PATIS avoids these critical path expansions because RTR is not needed; the entire FPGA configuration is reloaded when a module is updated. In

addition, the PR flow allows static connections between non-neighboring modules to transit PR regions without going through bus macros [29]. This direct routing avoids the delays added to non-neighbor pass-through signals in other RTR systems such as [30].

Incremental floorplanning re-implements only the changed modules, leading to shorter place-and-route times, shown under PATIS incremental in Table 5.2, for the number of modified modules given in parentheses. In the manual non-PR flow, any change in the design requires re-implementation of the entire design. In DMD, the incremental floorplanner tries to modify the existing floorplan which may reduce the run-time by re-implementing only modified modules. The incremental floorplanner tries to reduce the number of modified modules by reducing the ripple effects.

Chapter 6

Conclusions and Future Work

This thesis presents a novel approach to improve the productivity of large-scale, static FPGA designs. With increasing design size and stricter timing constraints, the implementation tools run-times increase. During initial development, any small change in the design requires complete re-implementation. To reduce this overhead, it is essential to re-implement only the modified logic in the design.

Dynamic Modular Design aims to improve the productivity of static FPGA designs in the early stages of development in two ways: reducing the run-time of the tools when implementing the entire design by placing and routing different modules in parallel, and re-implementing only the modified modules in case of localized changes. DMD tags each module as partially reconfigurable. PATIS automatically floorplans the modules in the design and anticipates changes, thus generating a suite of floorplan variants with modified resource usage. Whenever the requirements of a module changes, PATIS tries to choose a floorplan from the suite. In case a valid floorplan is not found, the incremental floorplanner tries to generate one with minimal ripple effects. All the modules are implemented in parallel which dramatically improves the tool run-times.

The contributions of the DMD flow are:

1. Development of the DMD flow to improve the productivity of static designs during the initial stages.

2. Design and implementation of an automatic floorplanner to assign area constraints for modules in a PR design.

3. Design and implementation of a speculative floorplanner which anticipates changes in a design and generates a suite of floorplans.

4. Incremental floorplanning to re-implement only modified modules in case of localized changes.

5. Automatic bus macro placement for interconnections between Partially Reconfigurable Modules.

6. Implementation of a debug module which passively monitors the module interfaces.

7. Parallel implementation of the design using a divide-and-conquer approach.

8. Automation of the design flow, including synthesis, place-and-route, and bitstream generation.

The primary focus of this thesis was the design and implementation of the PATIS automatic floorplanner. PATIS is used by DMD under two conditions: when the design is implemented for the first time, and when the incremental floorplanner cannot generate a floorplan for the modified design. Manual floorplanning of a complex, non-planar design is time-consuming and inefficient. PATIS reduces the burden on the designer by automating the process of floorplan generation. An average 10× speedup is observed in the implementation time when DMD flow is used for the entire design. When only modified modules are re-implemented, an average speedup of 35× is observed.

6.1 Future Work

Although significant speedups are possible with the current version of PATIS, it should be possible to further improve performance and usability in several ways:

1. **Portability**: The current version of the tool targets Virtex-4 FPGAs, although it is desirable to support newer families of Xilinx FPGAs. The resource map and the resource estimator are device-independent. However, there are a few device-dependent parameters including configuration frame characteristics and bus macro implementations. Future versions of the tool need to handle FPGA chips with additional resource types.

2. **Multiple Initial Floorplans**: PATIS currently generates one floorplan that satisfies all the resource requirements and timing constraints. Floorplan generation completes once a valid floorplan is obtained. To improve the quality of the floorplan, a large number of floorplans could be generated and timing verified. A floorplan can then be chosen based on parameters such as worst-case delay and overall resource utilization.

3. **Automate Entire Flow**: The current version of the DMD flow automates floorplan generation, bus macro generation, and parallel module implementation. However, some user intervention is currently required to run the separate tools. Complete automation is desired to give a push-button like flow.

Bibliography

[1] FPGA design flow overview. [Online]. Available: http://www.xilinx.com/itp/xilinx4/pdf/docs/dev/dev.pdf

[2] H. J. Kahn and R. F. Goldman, "The electronic design interchange format EDIF: present and future," in *DAC '92: Proceedings of the 29th ACM/IEEE Design Automation Conference*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 666–671.

[3] Xilinx 5.1i incremental design flow. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp418.pdf

[4] Ultra high-capacity task force. [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/design_entry/floorplanner.htm

[5] Using SmartGuide. [Online]. Available: http://www.xilinx.com/itp/xilinx10/isehelp/ise_p_using_smartguide.htm

[6] Incremental design reuse with partitions. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp918.pdf

[7] D. Zacher. (2009, April) Incremental synthesis: achieving shorter design cycles without quality trade-offs. [Online]. Available: http://www.fpgajournal.com/articles_2009/20090414_mentor.htm

[8] Physical synthesis and optimization with ISE 9.1i. [Online]. Available: http://www.xilinx.com/support/documentation/white_papers/wp230.pdf

[9] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Inf. Control*, vol. 57, no. 2-3, pp. 91–101, 1983.

[10] M. Wang, A. Ranjan, and S. Raje, "Multi-million gate FPGA physical design challenges," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided Design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 891.

[11] P. A. Sastry S., "The complexity of the two-dimensional compaction of VLSI layout." in *IEEE International Conference on Circuits and Computers*, 1982, pp. 402–406.

[12] L. Cheng and M. D. F. Wong, "Floorplan design for multi-million gate FPGAs," in *ICCAD'04: Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*. IEEE Computer Society, 2004, pp. 292–299.

[13] J. Yuan, S. Dong, X. Hong, and Y. Wu, "VLSI floorplan based on less flexibility first principle and linear programming," in *ASIC, 2005. ASICON 2005. 6th International Conference On*, vol. 2, 24-27 2005, pp. 832 – 835.

[14] Y. Feng and D. P. Mehta, "Heterogeneous floorplanning for FPGAs," in *VLSID '06: Proceedings of the 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 257–262.

[15] P. Banerjee, S. Sur-Kolay, and A. Bishnu, "Floorplanning in modern FPGAs," in *VLSID'07: Proceedings of the 20th International Conference on VLSI Design*. IEEE Computer Society, 2007, pp. 893–898.

[16] P. Banerjee, M. Sangtani, and S. Sur-Kolay, "Floorplanning for partial reconfiguration in FPGAs," in *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 125–130.

[17] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 28-30 2006, pp. 1–8.

[18] Partial reconfiguration user's guide. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/ug208_92.pdf

[19] J. M. Carver, R. N. Pittman, and A. Forin, "Automatic bus macro placement for partially reconfigurable FPGA designs," in *FPGA'09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2009, pp. 269–272.

[20] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *DAC '97: Proceedings of the 34th Annual Design Automation Conference*. ACM, 1997, pp. 526–529.

[21] P. Schumacher and P. Jha, "Fast and accurate resource estimation of RTL-based designs targeting FPGAs," in *FPL 2008, International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 59–64.

[22] K. Kozminski and E. Kinnen, "An algorithm for finding a rectangular dual of a planar graph for use in area planning for VLSI integrated circuits." in *DAC '84: Proceedings of the 21st Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 1984, pp. 655–656.

[23] Alpha Data ADM-XRC-4FX. [Online]. Available: http://www.alpha-data.com/products.php?product=ADM-XRC-4FX

[24] Fast Fourier transform. [Online]. Available: http://mathworld.wolfram.com/FastFourierTransform.html

[25] MicroBlaze processor reference guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[26] Adaptive soft output Viterbi algorithm (ASOVA) turbo decoder. [Online]. Available: http://www.ecs.umass.edu/ece/tessier/rcg/benchmarks/asova.html

[27] FloPoCo. [Online]. Available: http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

[28] R. Hymel, A. George, and H. Lam, "Evaluating partial reconfiguration for embedded FPGA applications," in *Proc. High-Performance Embedded Computing Workshop (HPEC)*, MIT Lincoln Lab, Lexington, MA, 2007.

[29] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *FPL 2006, International Conference on Field Programmable Logic and Applications*. IEEE, 2006, pp. 1–6.

[30] C. Patterson, P. Athanas, M. Shelburne, J. Bowen, J. Surís, T. Dunham, and J. Rice, "Slotless module-based reconfiguration of embedded FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 1–26, 2009.

Nomenclature

ASIC Application-Specific Integrated Circuit, page 2

BRAM Block Random Access Memory, page 5

BSP Binary Space Partition, page 13

CLB Configurable Logic Block, page 5

DFT Discrete Fourier Transform, page 57

DMD Dynamic Modular Design, page 2

DSP Digital Signal Processor, page 5

EDA Electronic Design Automation, page 12

EDIF Electronic Design Interchange Format, page 8

FFT Fast Fourier Transform, page 57

FloPoCo Floating Point Cores, page 58

FPGA Field-Programmable Gate Array, page 1

GUI Graphical User Interface, page 28

HDL Hardware Description Language, page 1

HLV High-Level Validation, page 27

HPWL Half-Perimeter Wire Length, page 13

ICAP Internal Configuration Access Port, page 27

IRL Irreducible Reduction List, page 13

LLD Low-Level Debugging, page 27

LUT LookUp Table, page 5

MDF Modular Design Flow, page 10

NCD Native Circuit Description, page 9

NGC Native Generic Circuit, page 8

NGD Native Generic Database, page 8

PAR Place And Route, page 1

PATIS Partial module-producing, Automatic, Timing-aware, Incremental, Speculative floorplanner, page 3

PR Partial Reconfiguration, page 2

PRM Partially Reconfigurable Module, page 17

PRR Partially Reconfigurable Region, page 17

RPM Relationally Placed Macro, page 29

RTR Run-Time Reconfiguration, page 18

UCF User Constraints File, page 22