Accelerating Incremental Floorplanning of Partially Reconfigurable Designs to Improve FPGA Productivity

Athira Chandrasekharan

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
Peter M. Athanas
Paul E. Plassmann

August 5, 2010
Blacksburg, Virginia

Keywords: FPGAs, Reconfigurable Computing, Incremental Floorplanning

Accelerating Incremental Floorplanning of Partially Reconfigurable Designs to Improve FPGA Productivity

Athira Chandrasekharan

ABSTRACT

FPGA implementation tool turnaround time has unfortunately not kept pace with FPGA density advances. It is difficult to parallelize place-and-route algorithms without sacrificing determinism or quality of results. We approach the problem in a different way for development environments in which some circuit speed and area optimization may be sacrificed for improved implementation and debug turnaround. The PATIS floorplanner enables dynamic modular design, which accelerates non-local changes to the physical layout arising from design exploration and the addition of debug circuitry. We focus in this work on incremental and speculative floorplanning in PATIS, to accommodate minor design changes and to proactively generate possible floorplan variants. Current floorplan topology is preserved to minimize ripple effects and maintain reasonable module aspect ratios. The design modules are run-time reconfigurable to enable concurrent module implementation by independent invocations of the standard FPGA tools running on separate cores or hosts.

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

Dedication

To Mom

(for showing me the magic of books and learning)

And Dad

(for teaching me how to dream high)

Acknowledgments

Where I am today would not have been possible without the help and support of so many people.

I would like to thank Dr. Cameron Patterson for giving me the opportunity to work in the *Configurable Computing Machines* lab and being my advisor for my thesis research. His insightful ideas and guidance have helped me achieve my goals.

I would also like to express my gratitude to Dr. Paul Plassmann and Dr. Peter Athanas for serving on my thesis committee and providing me with valuable comments on my research.

This thesis would not have been possible without the help of my colleagues in the FPGA productivity project — Sureshwar Rajagopalan, for his timely support that helped me complete my research and thesis; Tannous Frangieh, for the indispensable help he provided when I faced computer issues and also for his valuable inputs towards my research; Yousef Iskander and Stephen Craven, for their help and guidance in this project.

Thanks to *Bollo's Cafe and Bakery* for their cozy ambiance that helped me focus on writing my thesis and also the daily supply of delicious coffee without which I could not have managed hours of writing each day.

A zillion thanks to my friends for pulling me up each time I got in a rut and helping me through this rewarding journey, especially the hectic first semester. And finally, thanks to my family for supporting all my endeavors and constantly encouraging me to dream high and not settle for anything less.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Field-Programmable Gate Arrays (FPGAs) are being increasingly used in real-time streaming [1, 2] and data-processing applications [3], cryptography [4, 5], and high-speed communications [6, 7], in part due to their ability to be reprogrammed multiple times. Changes to FPGA designs can however result in long run-times or several iterations before the implementation tools meet the timing requirements of the design. There is no guarantee that timing closure will occur after extended runs spanning hours or days. This thesis explores the incremental floorplanning of large-scale FPGA designs through the use of parallelism, Run-Time Reconfiguration (RTR), and design change speculation to improve FPGA productivity through faster design implementation.

1.1 Motivation

Configurable logic devices such as FPGAs permit arbitrary reprogramming of the devices' hardware functionality, leading to lower cost and faster time-to-market advantages. Thus FPGAs allow customization of hardware at an economical price even in low quantities. In addition, Intellectual Property (IP) cores can be integrated into FPGA chips, tailoring them

for certain specialized applications, thus competing with Application-Specific Integrated Circuits (ASICs).

More complex application domains and larger design sizes, however, translate to longer implementation times. It is not uncommon for the implementation tools to run for hours or days. Worse, there is no assurance that the design will meet timing after extended runs. Additionally, changes made to a design in later stages of the development process are not easily incorporated, and may require complete re-implementation. Although incremental floorplanners do exist, the designs still undergo several iterations of synthesis, floorplanning and implementation, leading to noticeable delays, and sometimes failing to achieve timing closure.

Over the last 20 years, there have been several improvements made towards software Integrated Development Environments (IDEs). The hardware development platforms however have remained somewhat antiquated. Hardware environments do not enjoy the rich feature set that software environments do, such as integrated text editors, hyper-linked error messages, rapid-prototype tools, the ability to link libraries (some of them standardized) to integrate new functionality and build tools such as GNU Make [8] that discriminately rebuild only the code that has changed and its dependencies.

In the past, the demand for faster placement was met by using faster processors and memories, and enhanced FPGA routing architectures, preserving hierarchy to reuse unchanged block implementations, and performing continuous optimization of code and data structures. Parallelism has started appearing in vendor tools — mainly in the place-and-route (PAR) tools. However, vendors are slow to adopt parallelism because of the complications added to the algorithms, the need for reproducible results, and possible quality of results trade-offs.

This thesis takes a different approach to the goal of speeding up FPGA implementation, and focuses on the early and middle stages of development during which modules are still undergoing significant change, and connections to on-chip logic analyzers are frequently modified. Some area and speed optimizations are possibly traded off for improved observability

and faster implementation. Rather than focusing on parallel PAR algorithms, we use a divide-and-conquer strategy on existing tools to speed up implementation. This is referred to as *Dynamic Modular Design* (DMD), which is distinguished from the established (static) modular design flow suiting late-stage development where the floorplan is fixed, changes are localized, and timing closure is the main objective. A *Partial module-producing, Automatic, Timing-aware, Incremental, Speculative* (PATIS) floorplanner [9] implements the DMD flow since the standard Partial Reconfiguration (PR) flow is manual and not normally associated with enhanced productivity. RTR usually leads to low productivity, but PATIS uses a simplified version of the PR flow that does not have the design or debug complications of hot-swapped modules. Bus macros are automatically inserted and provide passive, readback-based observability of all communications between floorplanned modules.

1.2 Contributions

The work presented in this thesis is part of a project that developed a novel FPGA design flow with the aim of improving FPGA productivity. This thesis illustrates the work done in developing an incremental floorplanner to accommodate design changes. Incremental floorplanning — a core feature of PATIS — extends the Xilinx PR flow as a means to update design modifications. The ability to implement modules independently leads to a divide-and-conquer method of mapping, placing and routing designs.

The incremental floorplanner also speculates changes in a design and generates variants of the initial floorplan produced by the PATIS automatic floorplanner. The speculative behavior of PATIS makes available several feasible layout options for a potential change in the design, thus speeding the process of design modification. These floorplans are stored in a database, leading to better indexing and faster access. Based on several metrics introduced in Section 3.2, speculation happens discriminately, treating modules differently and independently. The absence of a suitable floorplan invokes incremental floorplanning to meet new

resource requirements.

To summarize, this thesis makes the following contributions to the project:

1. An incremental floorplanner that makes fast incremental updates to the layout when changes are made to the design Hardware Description Language (HDL) source.
2. Anticipating changes in a design depending on modular attributes such as its tendency to undergo changes and difficulty to meet timing.
3. Using RTR as a means rather than an end towards the big-picture objective of improving FPGA productivity.
4. Parallel application of implementation tools to speed up design reconfiguration through a divide-and-conquer strategy.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 presents the background information on FPGAs, reconfigurable computing and prior research in incremental floorplanning. An overview of the DMD flow is presented in Chapter 3. The initial experiments conducted as part of the proof-of-concept phase are also presented. Chapter 4 summarizes the incremental floorplanner flow, supporting tools, and algorithms. The process of gathering FPGA and design information, and communication with back-end tools are also described. Floorplanning results applied to the benchmark suite are analyzed in Chapter 5. Chapter 6 presents conclusions and discusses future work.

Chapter 2

Background

This chapter covers background topics and related work that pertain to this thesis. It begins with a general discussion of FPGAs, followed by an analysis of the standard FPGA design flow and its several enhancements. Reconfigurable computing is introduced and a survey of previous work in incremental floorplanning is presented.

2.1 FPGAs

FPGAs are semiconductor devices with programmable logic and interconnects. The programmable fabric consists of logical elements such as gates, memories, multipliers, and digital signal processing components, among several others. FPGAs are field-programmable devices, meaning that the user can program them, as opposed to the manufacturer. A common analogy of the FPGA is the electronic breadboard, on which a wide variety of electric circuits can be created by placing electronic components and connecting them through wires. Modifications to the circuits can be easily made by a rearrangement of the components and/or wires.

The programmability of an FPGA is its greatest advantage over ASICs, which are integrated

circuits that are not configurable. As a result, the same FPGA chip can be used for different purposes at different times. Moreover, updates can be applied to a design already configured on the FPGA, providing faster time-to-market and lower cost advantages. On the other hand, FPGAs are slower and less power efficient than ASIC designs.

FPGAs are useful in applications where complex logic is required and future changes to the design are possible. They are used in equipment for video and imaging, aerospace, military, data-intensive, and digital signal processing applications. Figure 2.1 shows the layout of a typical FPGA as islands of logic blocks connected by a sea of interconnects. Logic blocks can be programmed to implement Boolean functions, whose complexity depends upon the manufacturer and the FPGA series being used. The routing channels are connected to the logic blocks through electrically configurable switches.

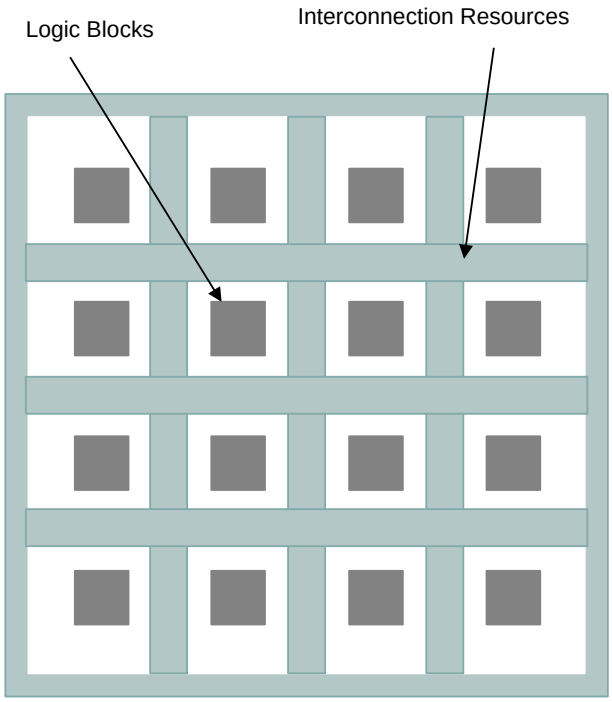

Figure 2.1: FPGA layout – logic blocks and interconnects

2.1.1 Resources

Figure 2.2 shows a detailed layout of a typical FPGA, using a portion of the Virtex-4 FX100 chip as an example. A diversity of resources constitutes the logic elements within an FPGA. The most common among these are Configurable Logic Blocks (CLBs), Block Random Access Memory (BRAM) slices and Digital Signal Processor (DSP) elements. Figure 2.2 illustrates columns of BRAM and DSP slices separated by blocks of CLBs. Different chips have different resources counts; for instance, chips oriented to signal processing applications have more DSP columns.

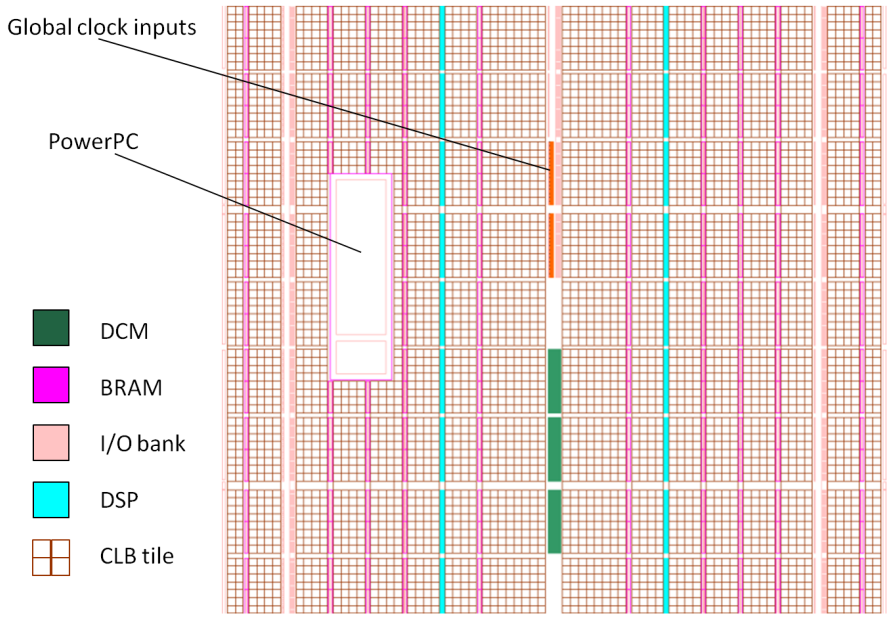

Figure 2.2: FPGA layout – resource details
(*Screenshot captured from PlanAhead tool*)

Configurable Logic Blocks are the most basic units of an FPGA. A Virtex-4 CLB consists of two `SLICEL` (a lookup table) and two `SLICEM` (a lookup table with Random Access Memory functionality) primitives. Every CLB has a configurable switch matrix with 4 or 6 inputs, some selection circuitry such as multiplexers, and flip-flops. The highly flexible switch matrix, which consists of pass transistors and contains no logic, can be configured to handle combinatorial logic, shift registers, or random access memories.

The Xilinx local or on-chip memory is made up of large embedded memory blocks called *Block RAMs*. Xilinx FPGA BRAM quantities differ by device. BRAMs are especially useful for embedded applications using a microprocessor.

Digital Signal Processing slices come in handy for applications that are compute-intensive. With the flexibility of creating custom hardware on FPGAs, the availability of DSP slices enhances the utility of these chips. These elements are useful for applications such as signal processing, filtering, compression and decompression.

In addition to these common resources, FPGAs have I/O banks, Digital Clock Management (DCM), Gigabit Transceivers and clock resources. Some FPGAs such as the one in Figure 2.2, have PowerPC processors, which are useful for embedded applications. FPGAs that do not have a processor core can be programmed to develop a soft-core processor in the FPGA fabric. Such a rich and varied organization of resources makes FPGAs extremely viable for several applications.

2.2 Current Design Methodologies

2.2.1 Standard FPGA Design Flow

The standard FPGA design flow shown in Figure 2.3 consists of three main stages: *design entry and synthesis*, *design implementation*, and *design verification* [10]. Design verification does not occur in a chronologically single step, but instead happens at different points of the flow. The Xilinx Integrated Software Environment (ISE) Design Suite supports all three phases of the flow. In the design entry phase, the Register Transfer Logic (RTL) description of the design is created using either text-based entry through popular HDLs such as Verilog and VHDL, or schematic entry, or both. The earliest of the design verification stages — functional simulation — is done during design entry to verify the functionality of the logic for expected operation. Once the design has been created, the functional information in

the HDL is synthesized to generate a structural netlist, which is usually a Native Generic Circuit (NGC) or Electronic Data Interchange Format (EDIF) file. The netlist file, which contains a logical description of the design in terms of the components required and the interconnections between them, is then translated by a tool called `NGDBuild` to generate a Native Generic Database (NGD) file. The constraints for the design such as timing or placement parameters are also specified during this stage.

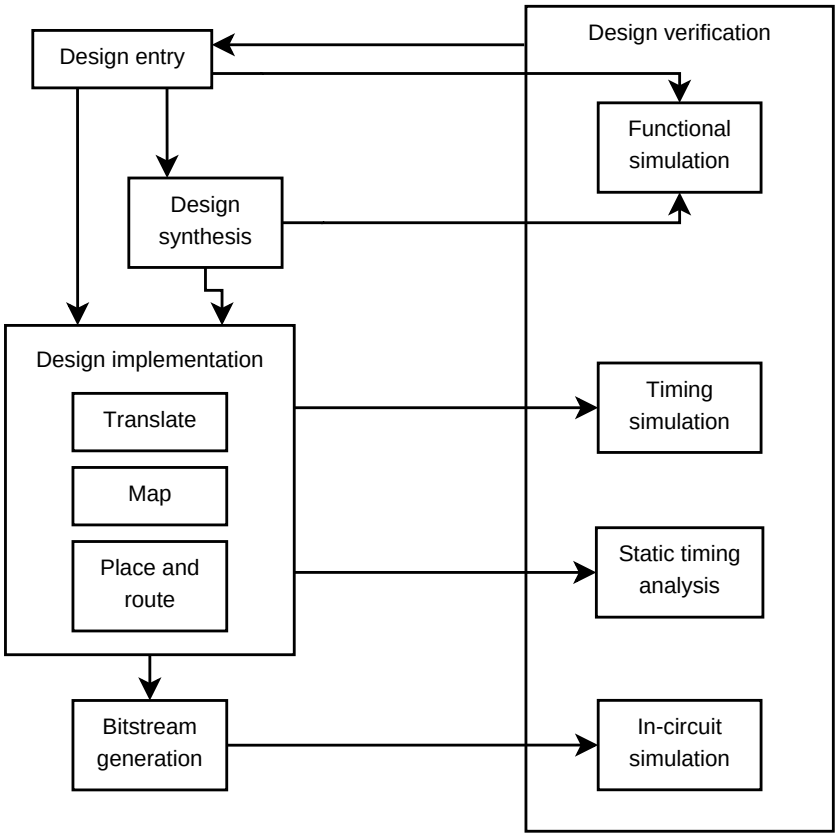

Figure 2.3: FPGA standard design flow

During the design implementation stage, the logic is mapped to the device primitives, producing a Native Circuit Description (NCD) file, which is a physical representation of the design. The Map-stage output NCD is placed and routed to produce an updated NCD file that includes placement and routing information. This can be followed by three design verification tests: timing simulation, static timing analysis, and in-circuit verification. *Timing simulation* verifies the speed of the design under worst conditions, and looks for setup and

hold violations. *Static timing analysis* is useful when quick timing checks are needed after a design is placed and routed. It also helps in finding any critical path delays in the design. Under *in-circuit verification*, different iterations of the design can be loaded to test in-circuit. A Xilinx utility called `BitGen` can then be used to produce a bitstream file (BIT) to download to the FPGA device from the placed and routed NCD file. The BIT file contains configuration information such as details about the logic components and their interconnects, and also device-specific information.

The standard FPGA flow is an iterative process of design entry, implementation and verification, until the design functions as expected. As a result, even small changes require a complete synthesis, placement and route of the design, consuming a considerable amount of design time. Therefore, this flow is not capable of making rapid updates in case of significant changes to a large-scale design. In the software domain, this is analogous to recompiling all libraries every time the application code changes. Another limitation of this flow is the possibility of timing failure in case of any design updates. Whenever a modification is made to a design, no matter how trivial, it may result in a timing failure or even increase the critical path delay of modules that previously met their timing, thereby requiring a full design re-implementation. The current FPGA tools still treat each design as a single entity, without enforcing any distinction between stable modules and modules that are under development.

2.2.2 Enhanced Design Flows

FPGA and Electronic Design Automation (EDA) tool vendors have come up with enhanced flows to address several limitations of the standard flow. For instance, Xilinx's Modular Design Flow [10] partitions a design into several modules, thus allowing a team of engineers to work independently on different modules and later merge them. This parallel development has the advantage that designers can isolate a module during modification, keeping other modules intact and allowing for independent timing closure on each module. Although this saves significant implementation time, the Modular Design Flow requires proper partitioning

of the design before the implementation phase and depends heavily on team member communication to ensure proper functioning of the final assembly phase. Although each module may meet timing independently, a combined implementation of all modules is still needed before generating the bitstream.

In May 2006, Synplicity and Xilinx formed the Ultra-High Capacity Task Force to introduce improved methodologies for large-scale designs, such as the Incremental Design Flow [11]. Employing hierarchical design practices, incremental design decreases run-times during design compiles by preserving unchanged logic stored as guide files [12]. This manner of reusing unchanged portions greatly reduces the time-to-market for large-scale designs.

SmartGuide [13] and Partitions [14] are two examples of incremental design techniques. *SmartGuide* looks for matching names between the current and previous netlists, in order to enable the tools to implement only the changes made to the design. This is especially useful at the end of the design cycle, in case of minor changes to a module, pin locations, timing constraints, or any such similar updates, after the overall design is stable and has already met timing. On the other hand, *partition* is suited to block-based projects, where each block is usually defined as a partition. Unmodified partitions from previous runs are preserved through a "cut-and-paste" approach, and only modified partitions are re-implemented. Although using partitions reduces tool run-times, it is essential to specify the partitions manually, early on in the design process. Altera introduced Incremental Compilation [15] that uses modular and hierarchical flows and preserves results of unchanged portions to implement designs incrementally.

Incremental flows however necessitate the use of incremental synthesis, which resynthesizes only the modified portions of a design while reusing unchanged logic. Changes at the functional level can be easily identified, disregarding irrelevant changes such as modified comments. Clearly, incremental synthesis reduces the input size of the problem since the amount of logic to be processed during each iteration reduces. However, it tends to optimize logic without any notion of placement or routing delay estimate, when in fact, routing is often the

main contributor to total delay. This can lead to the design not meeting timing and hence result in several more design iterations.

Such a design can benefit from the Physical Synthesis flow [16], taking advantage of the knowledge of the device's physical delays. In this flow, synthesis is aware of module placement and has an improved estimate of the routing delay, thus achieving timing more quickly. This improved visibility promotes faster implementation, and reduces design iterations. Figure 2.4 compares the Xilinx standard and physical design flows.

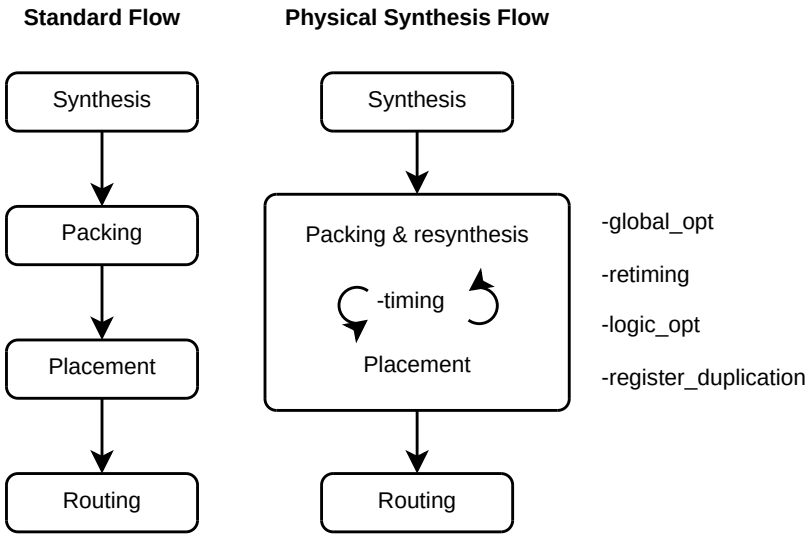

Figure 2.4: Xilinx standard and physical design flows

2.2.3 Reconfigurable Computing

Xilinx's PR flow allows a designer to designate regions in the FPGA fabric whose contents can be swapped while the remaining portion of the device continues normal, uninterrupted operation. There is tremendous potential for design size reduction as the reconfigurable region can be used to swap-in and swap-out modules on demand, rather than the traditional methodology of allocating static, permanent space to components of a design. This methodology can be used to implement future changes in modules. Although RTR is essentially a methodology used to implement volatile designs whose modules change and evolve over time,

DMD employs RTR as a strategy for fast implementation. Isolation of the modules ensures their parallel development and implementation without affecting the rest of the design.

By convention, most FPGA designs are static in their entirety without regard to a module's actual execution profile. However, this dynamic methodology has yet to achieve mainstream acceptance. Designers are reluctant to adopt RTR because of additional design complexity. 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.

Xilinx's PR flow [17] is shown in Figure 2.5. In the design entry phase, the static and PR regions are clearly demarcated using specified communication interfaces called bus macros. Following design entry, the timing and placement constraints for the design are specified, and each PR region is constrained within an area group or bounding box. This is followed by 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 logic has to be implemented first because some routing resources within the PR regions may be used to implement static routes and cannot be used by the PR modules. The routes within the PR regions used by the static logic are stored in the `arcs.exclude` file. Whenever the static region is re-implemented, the `arcs.exclude` file changes necessitating re-implementation of all the PR modules. The final step in the PR flow is merging the static and PR module implementations to generate a single bitstream.

DMD uses RTR for its ability to independently implement modules. Once the static logic is implemented, the PR modules can be compiled in parallel, thus saving significant implementation time. The timing analysis following the optional non-PR design implementation provides a preliminary report on the design's static timing behavior. This reduces the need

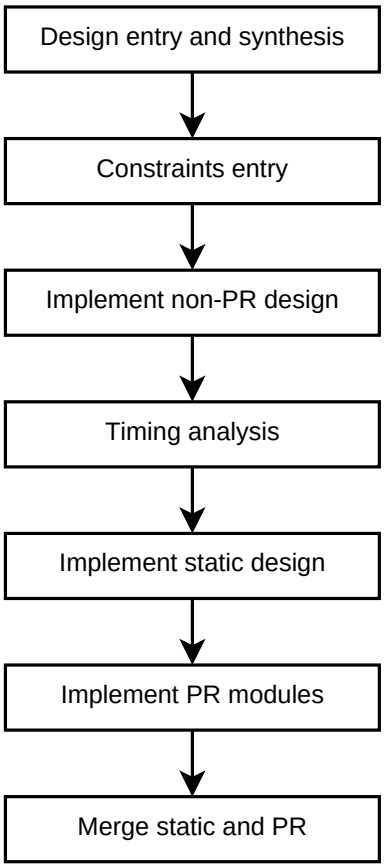

Figure 2.5: Xilinx's PR flow

for multiple iterations to meet a design's timing.

2.2.4 FPGA Incremental Floorplanners

Support for modifications on large-scale designs is fast becoming a necessary option in order to cope with the increasing complexity of FPGA designs. Although incremental flows have helped solve this problem to some extent, the slow tool response to design modifications increases the time needed to incorporate changes and route the design. Complete re-implementation and manual intervention are not preferred, especially for minor changes. Previous work in incremental floorplanning has been biased towards ASICs. Floorplanning algorithms are typically based on simulated annealing or genetic algorithms, using various

floorplan representations. Such algorithms typically need lengthy iterations to generate a feasible floorplan.

Area and wiring estimates are the metrics mainly used by incremental floorplanners to determine optimal placement modification of modules. Crenshaw et al. [18] proposed an incremental floorplanner that used a greedy method to apply local changes on a slicing floorplan tree. During incremental changes, affected nets were first removed, and pins updated, before reinserting these nets. Drastic changes to a module causes it to be labeled critical and re-floorplanned until no longer critical. This floorplanner uses an area metric to assess the need for a full floorplan revision. A productivity-driven methodology for incremental floorplanning was developed by Liao et al. [19], where a floorplan is computed using area estimates and the actual area required for each sub-circuit. These algorithms change the module areas without drastically altering their shapes or locations, and meet tight goals for area, performance and power. This methodology works with the physical design cycle to generate several polygon-representations of modules.

For incremental floorplanners to be effective, minimal changes should be made to the layout. Liu et al. [20] developed a floorplanner for non-slicing floorplans, using corner block list (CBL) representations for modules in the floorplan. CBL preserves the topological relations of the block, has a smaller upper bound on the number of possible configurations, and produces few redundancies. Menager et al. [21] developed a dynamic floorplanner for block placement, soft macro shaping, JTAG cell placement, power grid, and power rings design. Dedicated constraints are associated with priorities to guide the floorplanner during constraint overlap. Although this hierarchical flow captures the designer's intent through the use of relative references, a new floorplan description language is used to describe module location constraints. Liu et al. [22] created a tool that both generates a one-shot flooplan and handles incremental changes. Their floorplanner incorporates genetic algorithms to generate a feasible floorplan, and focuses on area and wirelength optimizations. Depending on which module is to be incrementally floorplanned, up to half the number of modules in the design could undergo re-implementation, which is unwanted.

2.3 Need for a New Design Flow

In contrast to the floorplanners discussed above, the DMD flow requires an incremental floorplanning methodology that can implement changes to multiple modules in parallel. Incremental modifications should be fast and local, and should not result in long runs for the implementation tools. It should also be possible to rapidly invoke the incremental floorplanner each time a potential design change is to be evaluated. The tool should be able to update the floorplan quickly because the exploration space involves small changes, usually local to the modified module and affecting only a portion of the design. An incremental floorplanner should not share tight coupling with a specific architecture or design tool. Although the methodology we propose uses Xilinx's PlanAhead to assist with placement decisions, any family of Xilinx FPGAs can be floorplanned. This is facilitated through extraction of FPGA data from architecture files and floorplanner interaction with back-end tools.

Chapter 3

System Overview

The focus of this chapter is to introduce the *Dynamic Modular Design* methodology which simplifies hardware development for large-scale FPGA designs. The tools that have been developed for this methodology speed up the floorplanning, placement and route phases of the FPGA design flow. In addition, they also simplify the process of incorporating design updates. The novel concepts introduced in this chapter include the use of partial reconfiguration to implement static designs, automatic floorplanning to generate a constraints file, automatic insertion of bus macros to enable intermodule communication, and speculative floorplanning to proactively accommodate design changes.

The complete approach involves estimating the resource utilization of a design followed by automatic floorplanning. Layout changes are anticipated through speculative floorplanning. Once a floorplan is generated, implementation follows, eventually generating a bitstream. The input to this flow is the netlist file obtained after HDL synthesis. In addition to the bitstream, which can be downloaded to the chip, the flow also generates several feasible constraints files. This chapter also illustrates the versatility of Xilinx's PlanAhead tool and lists the conclusions drawn from the initial experiments conducted as part of the proof-of-concept phase.

3.1 Dynamic Modular Design

DMD uses the Xilinx modular [10] and PR [17] flows to speed up implementation cycle time. As in the Xilinx modular flow, DMD partitions a design into several modules, which are implemented in specific partially reconfigurable development sandboxes. In addition, any glue logic between modules is considered part of the static logic. Through this concept, it is possible to dynamically adjust the boundaries between PR regions and static logic. Any change in a module may not require re-implementation of the complete design. Instead, only those modules whose placement has been changed will need to be re-implemented. A change in static logic will however necessitate a complete design update by the tools.

Figure 3.1 compares the DMD flow with the Xilinx standard flow in Figure 2.3, and shows how DMD fits into the overall FPGA design flow. As in the Xilinx standard flow, DMD starts with design entry. An RTL description of the design is generated either through text or schematic entry. The design is split into several independent self-contained units or modules, and this hierarchy is maintained henceforth during the rest of the flow. Next, functional simulation verifies the design logic for correct operation. The design is then synthesized to generate a netlist that contains the design's modules and interconnections.

The synthesized design is then floorplanned automatically. Floorplanning a design before the implementation phase has several advantages. It aims to minimize the chip utilization area, thereby reducing interconnect lengths and improving speed. This is done by identifying modules which can be placed close together, leading to less routing resources to connect them. Floorplanning thus helps to tackle the sometimes conflicting goals of area and speed. Floorplanning is typically a manual phase, but DMD has an automatic floorplanner that places modules efficiently.

Using the constraints file generated by the floorplanner, DMD implements the design and verifies timing, as in the Xilinx flow. The design is first translated using `NGDBuild`, then mapped, placed and routed to generate an NCD file. Finally, a bitstream is generated and

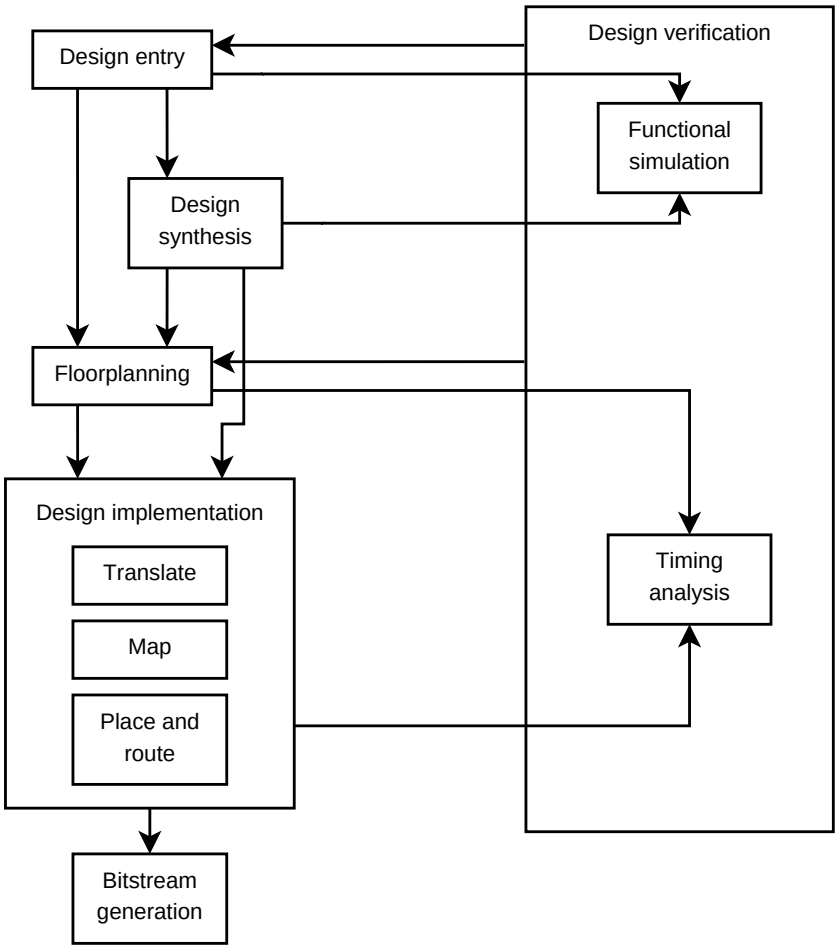

Figure 3.1: High-level DMD flow

downloaded into the chip.

DMD speeds up hardware development by using RTR as a design-time methodology, as opposed to a run-time strategy. During development, modules often change in size and resource requirements. Changes in such FPGA designs normally result in a long time to re-floorplan and re-implement the modules. By using PR, a modified module can be updated independent of other modules. Moreover, PR enables parallel implementation of modules, thus greatly improving development time. DMD adds a module-linkage step that helps reduce the synthesis and implementation time. This is done by anticipating and accommodating design changes to generate several possible module configurations.

3.1.1 Overall Flow

A detailed flowchart illustrating DMD is shown in Figure 3.2. After design entry and synthesis, the flow checks that the tool is doing a first run on that design. If true, the full design is synthesized and then floorplanned automatically, followed by a speculative floorplanning of the design to generate several alternative floorplans. For non-first time runs of the tool, only the modified modules are synthesized, followed by incrementally floorplanning the design to accommodate the new changes. This is a two-pronged strategy. The designs speculatively generated are considered first, and any match is used as the new floorplan. If there is no match, incremental floorplanning generates a new modified floorplan. Once a floorplan is available, design implementation and bitstream generation follows.

The timing behavior of each generated floorplan is verified before it is placed and routed. In the case of first-time runs of the tool on a design, a failure to meet timing leads to a re-run of the automatic floorplanner. For modified designs, another floorplan is fetched from the database. If the incremental floorplanner fails to generate a feasible floorplan, the automatic floorplanner is called. A suitable floorplan, once obtained, is implemented and a bitstream generated.

3.2 DMD Decision Attributes

As in a typical FPGA design flow, there are several attributes that control the floorplanning and implementation stages. Some of these parameters or constraints decide which tool is triggered, some determine which modules are to be updated, floorplanned or implemented. Yet other parameters control their placement. These attributes help the DMD flow to produce efficient floorplans that depend on module characteristics. A summary of attributes and their definitions is provided in Table 3.1.

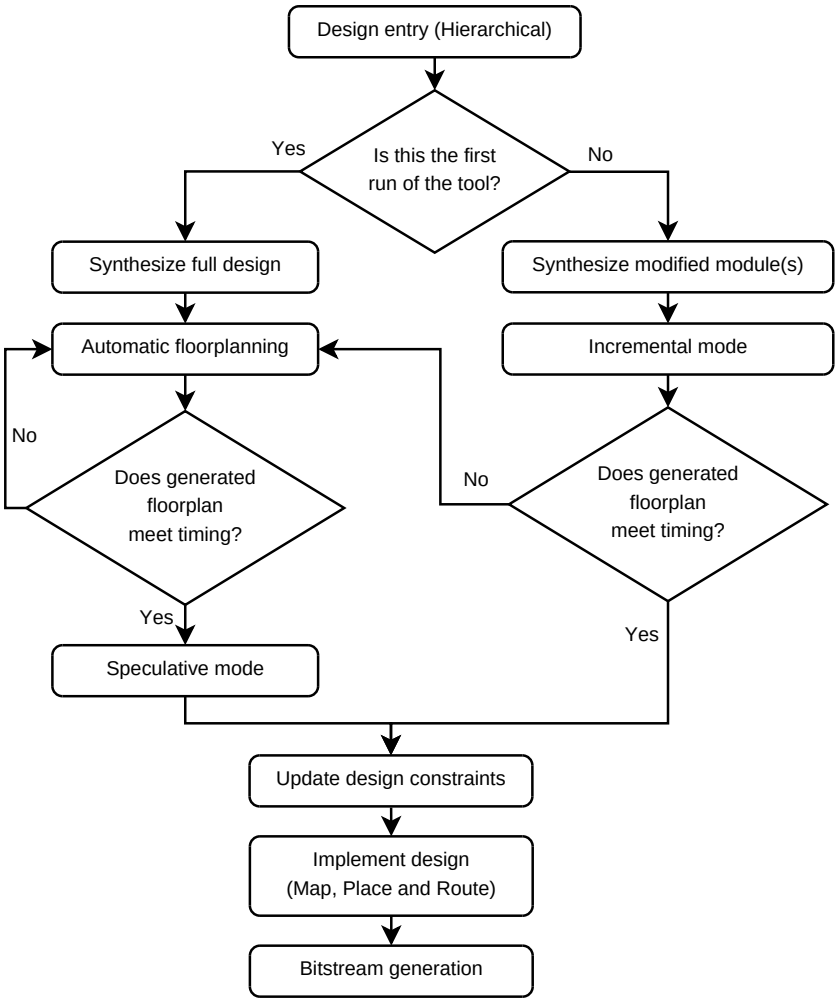

Figure 3.2: DMD functional flow

3.2.1 Module-Specific Attributes

The conventional approach to floorplanning treats all modules in a uniform manner, but DMD assigns to each module two new attributes: *fickleness* and *viscosity*. These metrics factor in a module's difficulty to meet timing during implementation and likelihood to change after implementation.

1. **Fickleness**: Each module has a fickleness that reflects its difficulty to meet the design timing constraints. Such modules may need to be implemented several times before the design finally satisfies timing. Altering the placement of such a module once timing

Table 3.1: DMD decision attributes

Attribute	**Definition**
Viscosity	Measure of a module's susceptibility to design changes
Fickleness	Measure of module's difficulty to meet timing constraints
Timing	Indication that a design operates at its designated frequency
Resource requirement vector	Enumeration and count of resources required by a design
Special resources	Need for resources present at limited locations on the chip
New design	Indicates whether a design is new or modified

is met may require several more implementation attempts before timing is satisfied again. Such modules are branded as fickle, and are not modified during speculative or incremental floorplanning. Their positions will be frozen in place.

2. **Viscosity**: A module's susceptibility to change is reflected by the viscosity attribute. A low viscosity value indicates a high likelihood of changes in a module, and vice versa. Low viscosity modules are assigned extra resources so as to minimize the ripple effect of changes on the overall layout. Aligning low viscous modules closer to each other has the added advantage that in case a module has to be modified, nearby modules will be able to shrink sufficiently to free up some resources.

3.2.2 Design-Specific Attributes

A design has several attributes that control its floorplanning. Two of them — *resource requirement vector* and *special resources* — play a role in deciding the layout of the design. The resources needed by a module determine its potential locations on the chip. *Timing* is the most important attribute of any design. A design has to be floorplanned such that it meets timing. The last attribute, *New design*, determines if a design is being floorplanned for the first time or incrementally.

1. **Timing**: This is the most important attribute of any FPGA design flow. A good floor-

planner generates a layout that meets timing. Different layouts can lead to different minimum design frequencies. Several aspects of a design — distance between modules, interconnect lengths, long concatenated chains of resources — influence the frequency of operation. Registering the ports of design modules also help to improve timing.

2. **Resource Requirement Vector**: Floorplanning aims to meet the resource requirements of a design. The module blocks should be placed so as to minimize area and maximize speed. Each design and module is associated with a resource requirement vector, which is an enumeration and a count of resources required to satisfy design logic and timing constraints. Module placement decisions depend on this vector. Moreover, incremental floorplanning will be done only if the current floorplan violates any of the elements of the vector, since not all design modifications trigger a need for more resources.

3. **Special Macros**: Certain resources, such as PowerPC processors and DCMs are present only at certain locations on the chip, unlike CLB and BRAM slices that are distributed more uniformly across. If these locations are occupied but not used by any module, they will not be available to modules that need them. Hence, prior to floorplanning, a list of these macros is needed to ensure satisfaction of the resource requirement vector.

4. **New design**: DMD makes a two-pronged decision each time a design is synthesized. New designs go through an automatic floorplanner that generates a floorplan and verifies timing. Next, changes in the design are anticipated based on the module viscosities, generating variants of the initial automatic floorplanner-generated layout. Modified designs, however, are incrementally floorplanned to meet the updated resource requirements. Typically this involves changes to very few modules. The incremental floorplanner first attempts to find a suitable floorplan among those generated speculatively. If unsuccessful, it modifies the current floorplan to accommodate the changes. This is demonstrated in Figure 3.2.

3.3 PATIS Flow

PATIS is a partial module-producing, automatic, timing-aware, incremental and speculative floorplanner that implements DMD by leveraging the Xilinx PlanAhead tool to generate feasible floorplans. When a module change affects a floorplan, PATIS accommodates the change by applying a minimum set of updates to the existing floorplan. Ripple effects are considered, and a completely new floorplan may be generated if incremental changes are inadequate. PATIS also speculatively generates multiple floorplans depending on module viscosities and resource requirements, resulting in faster re-implementation of a modified design.

The PATIS flow is shown in Figure 3.3. The design is synthesized using Xilinx Synthesis Technology (XST), generating an NGC netlist. This netlist is converted to EDIF and then parsed to generate design-specific information such as the list of modules and their interconnections. The EDIF file is also scanned for an estimate of the resource usage. Once the chip details are known, a map of the resource layout is created. PATIS then automatically generates a floorplan which is timing-verified. The availability of a floorplan triggers the incremental floorplanner to speculatively generate several feasible floorplans. All floorplans are subsequently verified for timing behavior, and if successful, indexed in the database.

Alternatively, during subsequent runs of this tool for the same design, the incremental floorplanner is triggered to meet the new resource requirements. It first scans the database for a suitable floorplan. If unsuccessful, it attempts to incrementally floorplan the design. If the incremental floorplanner fails, the modified design will be floorplanned from scratch. Since the modules are partially reconfigurable, bus macros are placed on their boundaries before implementation. PATIS has an automatic bus macro placer that uses the Xilinx PAR tool to determine location constraints for bus macros. The inter-module timing of the floorplan is also determined while inserting bus macros.

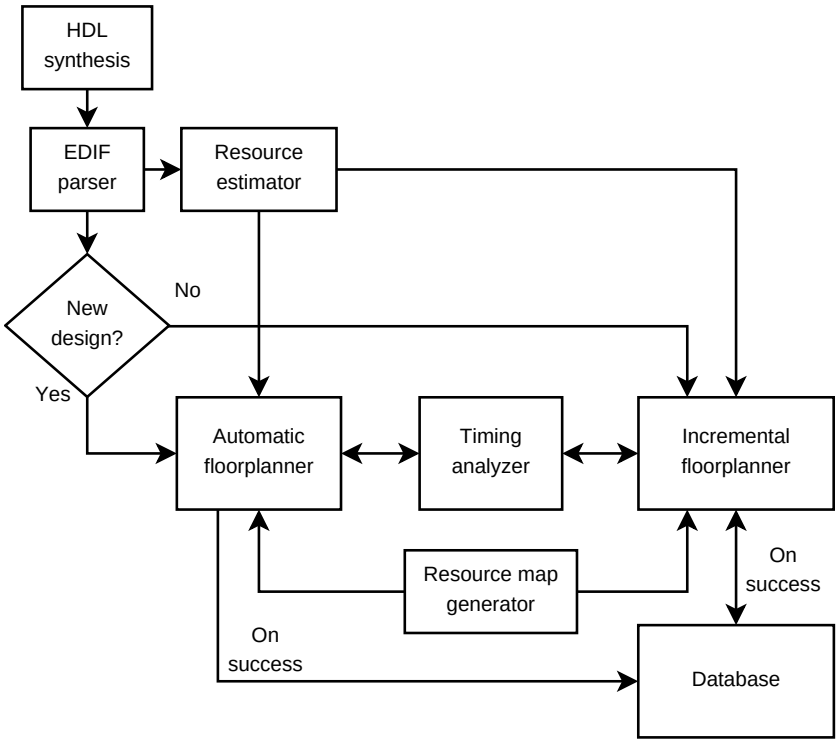

Figure 3.3: PATIS flow

3.4 Automatic Floorplanning

Newly synthesized designs are automatically floorplanned by PATIS. The automatic floorplanner assigns area constraints to module instances while minimizing the distance between interconnected modules. Floorplanning any design requires precise resource estimates for each module — this information is obtained from the synthesized netlist. 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 as explained in Section 3.6.

The first step in the automatic floorplanning flow, shown in Figure 3.4, generates a hyper-

graph from the synthesized design with modules as vertices and interconnections as hyper-edges. The hypergraph obtained is recursively bisected using hMetis [23] to create a slicing tree structure, whose leaves correspond to the design modules. The last step before assigning location constraints is to generate an *Irreducible Realization List* (IRL) for each 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 the tree leaves, which correspond to the modules in the design. The floorplan needs to be timing verified — static routes need to satisfy all the timing constraints — before it can be deemed valid. In case the floorplan fails timing, a different slicing tree representation is generated.

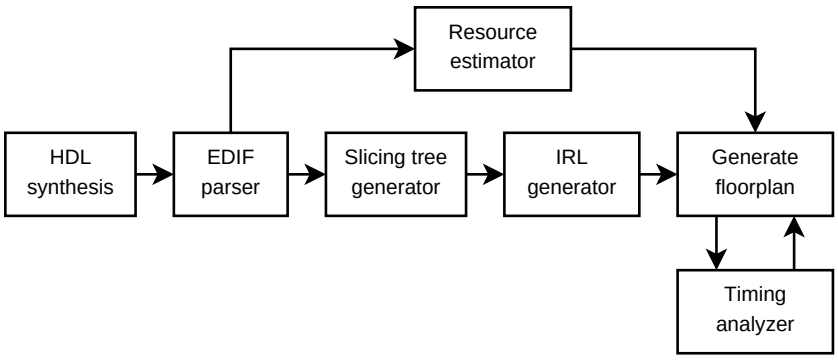

Figure 3.4: Automatic floorplanner flow

3.5 Speculative and Incremental Floorplanning

Modifications to a design trigger the incremental floorplanner to make local updates to modules to meet their new resource requirements. When a module changes, PATIS scans the database for a suitable floorplan. If the search is unsuccessful, the incremental floorplanner 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 around a module and only then attempts to modify neighboring modules.

Once a floorplan has been generated for a design, variants are speculatively added to the database in order of increasing module viscosities. Each module has a minimum resource constraint that cannot be violated. 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 rows or columns of resources 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. The incremental floorplanner is described in detail in Chapter 4.

3.6 Timing Analysis

After module placement, bus macros are inserted on module boundaries to facilitate intermodule communication. The `arcs.exclude` file created after the map, place and route of the static region contains the routing resources to be excluded during implementation of the modules. As a result, the static region can remain unaffected when the module is reconfigured. PR modules only use the routing resources contained within.

The conventional PR flow requires the bus macros to be instantiated and placed manually. It can be quite cumbersome to insert and place the large number of bus macros needed in any large-scale RTR FPGA design. Writing the HDL code is just as tedious. PATIS uses PAR to directly place the bus macros thus improving the tool run-time. The PATIS automatic bus macro placer is beneficial for designs constrained by delay on the intermodule connections. PATIS also automates the bus macro instantiation process thus requiring no additional steps from the designer as compared to a conventional PR design.

The overall flow for the bus macro insertion process is shown in Figure 3.5. The top-level HDL is parsed to obtain (i) the reconfigurable module instances, (ii) their input and output port information, and (iii) the nets connected to their input and output ports. The automatic bus macro placement tool creates a graph of the intermodule connections and places bus macros

at module boundaries using the graph. Nets between modules are instead routed through the bus macros. The module instances are updated and the bus macro instantiations are inserted in the top-level HDL. An NCD file is created using only the bus macro instances with the help of the `fpga_edline` tool, a script version of `fpga_editor`. Xilinx PAR is then used to generate an optimized placement of bus macros.

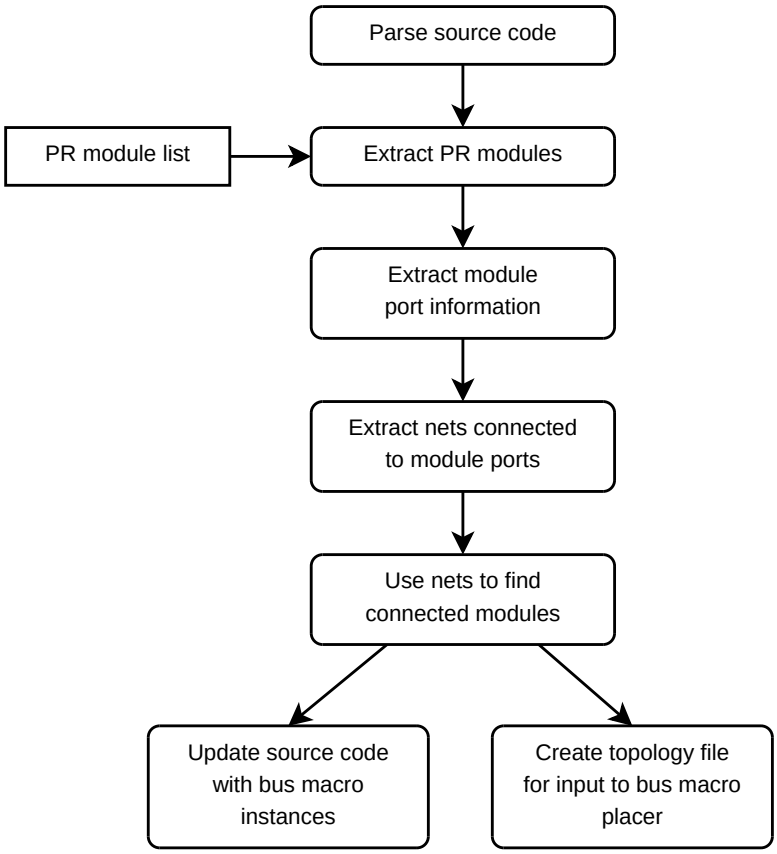

Figure 3.5: Bus macro insertion flow

3.7 Debug

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 reintegrated without altering the rest of the design. Register state is retrieved through the Internal Configuration Access Port (ICAP), a Xilinx proprietary interface allowing direct access to register bits as opposed to serially shifting the entire state as is done in Joint Test Action Group (JTAG) testing standard. LLD is demonstrated in Figure 3.6.

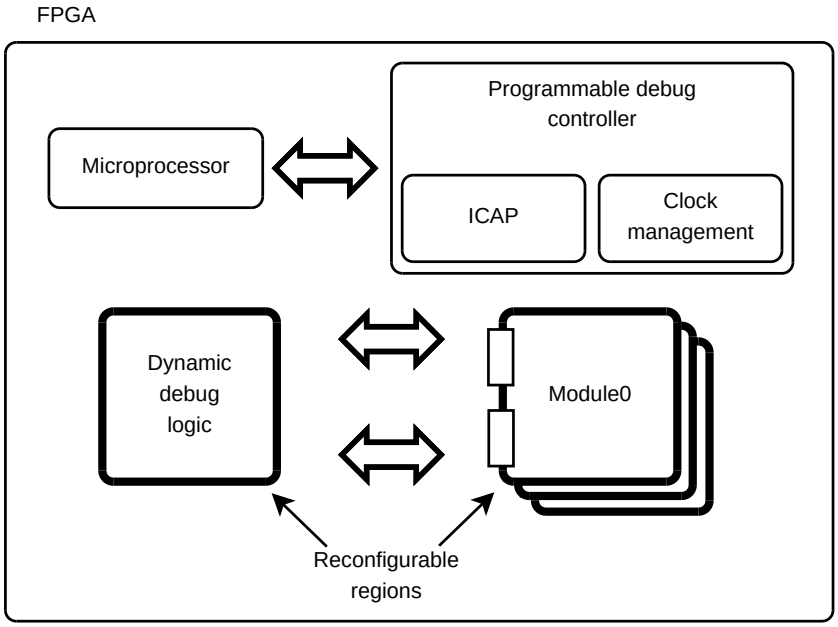

Figure 3.6: Low-level debugging in DMD

High-Level Validation, shown in Figure 3.7, abstracts away the 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 used in software unit-testing, where individual components are tested against known conditions.

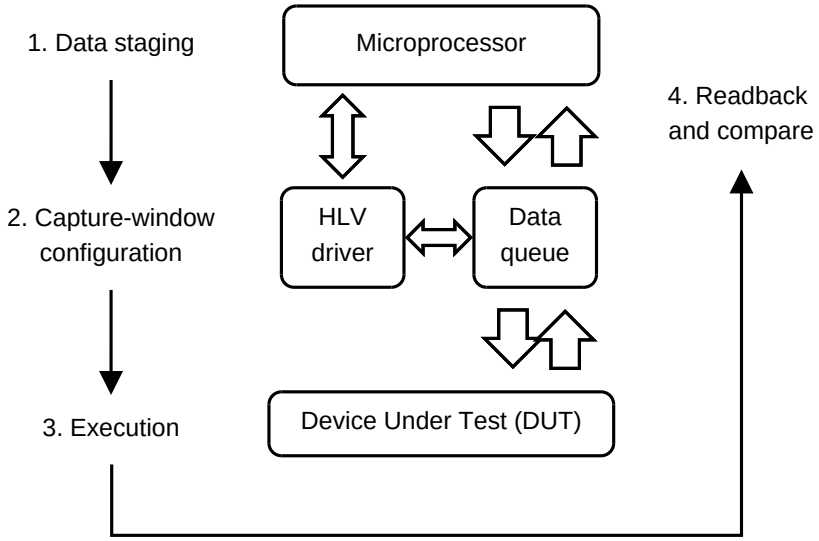

Figure 3.7: Overview of high-level validation

3.8 PlanAhead

Floorplanner Graphical User Interfaces (GUIs) help the designer allocate regions for design modules in order to improve performance and to make it more likely that achieving timing in one module does not degrade timing in other modules. PlanAhead is a versatile, user-friendly and manual floorplanning tool that was developed by Hier Design Inc, and later acquired by Xilinx to become the preferred graphical floorplanning method. In the 11.1i release, PlanAhead is fully integrated into Xilinx ISE, does not require a separate license, and replaces the original Xilinx floorplanner [24]. A screenshot of the PlanAhead GUI is shown in Figure 3.8.

Figure 3.9 shows the use of PlanAhead in the ISE design flow. After the RTL description of the design is synthesized using XST, the generated netlist can be used by PlanAhead to generate a User Constraints File (UCF), which contains the design timing and location constraints. Alternatively, a UCF can be imported by PlanAhead to load pre-existing design settings. Since the PlanAhead GUI greatly simplifies the floorplanning and implementation processes, DMD integrates PlanAhead into PATIS, thereby making use of the many helpful features enumerated below:

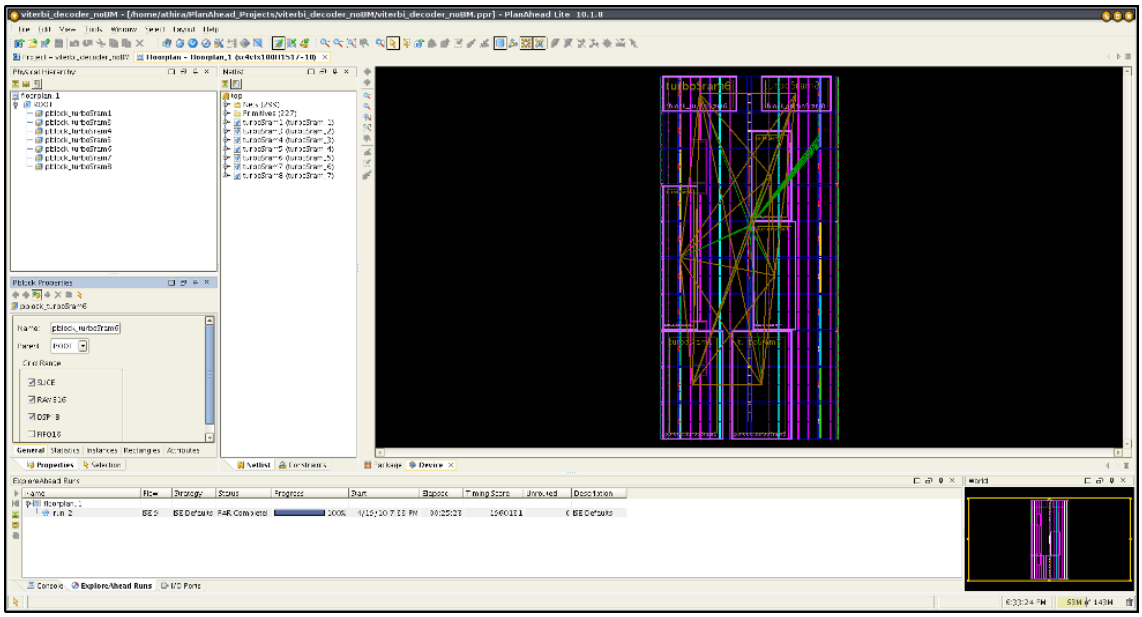

Figure 3.8: PlanAhead GUI

1. Although PlanAhead is used as a floorplanning tool, it also serves as a complete flow management tool from RTL development through bitstream generation [25]. In the PlanAhead 11 release, logic synthesis is included. In addition, PlanAhead can also be used for I/O pin planning, RTL netlist analysis, implementation result design analysis, floorplanning, or ChipScope tool core insertion and debugging.

2. PlanAhead supports a hierarchical, block-based and incremental design methodology. It can construct a design using multiple EDIF netlists that constitute a hierarchical design, or from a single hierarchical netlist. In addition, PlanAhead also supports PR [26], thus facilitating module modification and implementation. These features are necessary in DMD's reconfigurable design flow.

3. PlanAhead 10.1 can be used to invoke the ISE 9.2i PR design flow's implementation tools. In addition to the default ISE runs, PlanAhead also supports several enhanced runs with different strategies providing different effort levels for MAP and PAR. Each strategy tries to intelligently optimize various PAR options. This aims to provide broader implementation options for the designer.

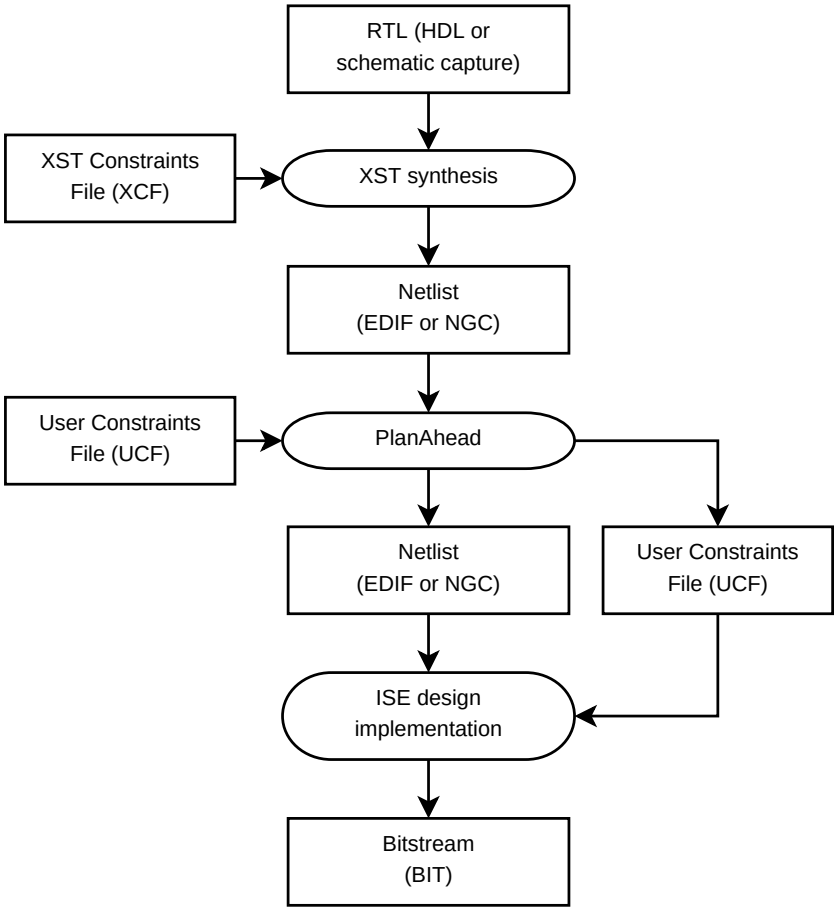

Figure 3.9: PlanAhead used in the ISE design flow

4. When implementing a reconfigurable design, PlanAhead 10.1 allows parallel PAR runs on different reconfigurable regions. This can be exploited by a multiprocessor system with adequate memory and memory bandwidth. Table 3.2 compares the time taken for serial and parallel implementations of a 10 MicroBlaze system at different frequencies. This ability to run parallel jobs during implementation is one of the main features of DMD.

5. 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.

Table 3.2: Serial versus parallel implementation times for a 10 MicroBlaze design

f_{clk} (MHz)	Serial Implementation	Parallel Implementation	Speedup $=Serial/Parallel$
122	33 minutes	12 minutes	2.75
125	45 minutes	16 minutes	2.64
128	1 hour 20 minutes	29 minutes	2.76

6. Each time a module is placed, moved or removed from a design layout, PlanAhead updates the UCF. Placement constraints can be generated from the four corner Relationally Placed Macro (RPM) coordinates of a module. This feature helps DMD to generate UCF files for any design whose corner coordinates have been determined.

7. When a design netlist is imported into PlanAhead, an estimate of its resource utilization is determined. The resource requirement vector is used by DMD to floorplan modules and allocate sufficient resources to each module.

3.9 Initial Experiments

Proof-of-concept experiments were conducted using two designs to demonstrate the speed advantages of DMD over a flattened implementation and a modular non-PR implementation. The first design consisted of multiple MicroBlaze processors connected in a ring topology through bidirectional Fast Simplex Link (FSL) interfaces. The second design was a series of cascaded complex Fast Fourier Transforms (FFTs). For floorplanning purposes, the former design can be represented by a simple planar graph, making it easier to floorplan than the latter design, which requires a non-planar graph for its representation.

Table 3.3 shows the ISE 9.2i place-and-route times in CPU minutes using a 2.66 GHz Core i7-920 processor, targeting Xilinx's Virtex-4 FX100-ff1517 device. Note that for both designs, the clock frequency was adjusted to make timing challenging but attainable. As evidence of

Table 3.3: PAR times for proof-of-concept designs with size and timing variations

Design	f_{clk} **(MHz)**	**Elapsed PAR time (minutes)**		
		Flat Design	**Manual FP**	**PR FP**
MB 5	127.4	330	80	20
MB 10	127.4	-	270	28
MB 20	125	-	-	105
CFFT 3	256.4	-	35	10
CFFT 6	250	75	-	24

Legend: MB – MicroBlaze, CFFT – Complex Fast Fourier Transform, PAR Time – Place-and-Route time in CPU minutes, FP – Floorplanning, A '-' indicates that the design did not meet timing in less than 30 hours.

the DMD approach advantages, the following observations were made:

1. Implementation time is a function of design size. Large designs can take a long time to complete implementation. In addition, PR flow required less time to implement the designs compared to the flattened and modular implementations. This is essentially because PR implementation of modules ensured no overlap of module boundaries and facilitated their parallel implementation.

2. Flattened implementation of large designs either failed to meet timing or took several hours, sometimes days, to achieve timing closure. For some designs, the tools ran for several days without meeting timing. The DMD approach was however able to achieve timing closure even where the flattened implementation failed.

3. The modular implementations for some designs did better than the flattened implementations, although in some cases, even the modular implementation failed to meet timing. This is because although the modules are independent, they are implemented together, and hence timing challenges in one module may spill over to other modules, or be aggravated by other modules' resource requirements. Moreover, the complete design has to be re-implemented even if only one module is failing timing. The effect of this is more pronounced under severe timing constraints.

These observations indicate that a PR implementation can be used to reduce tool run-times. The independent module implementation leads to faster bitstream generation. Moreover, PATIS is able to meet timing under conditions that are unattainable in the normal flow. PATIS overcomes the resource contentions among modules in the modular flow by separately placing and routing each top-level module. For incremental designs, PATIS reduces the input size of the problem solved by the place-and-route tools and ensures that only modified modules are re-implemented. Finally, by implementing modules separately, PATIS facilitates the use of multi-cores to concurrently run the tools on different modules. The lack of optimization across module boundaries and increased area are usually deemed acceptable in development environments.

Chapter 4

Incremental Floorplanner

The previous chapter gave a detailed introduction to the principles of DMD and the PATIS tool. The actual implementations of the PATIS incremental floorplanner and its auxiliary tools are the focus of this chapter. While the generation of the netlist files and the design implementation after floorplanning are handled by commercial tools such as Xilinx ISE, PATIS is responsible for floorplan generation and modification, and bus macro placement.

This chapter begins with an overview of the functional flow and discusses the various steps in the incremental floorplanner implementation. Next is an analysis of the strategies that form the basis of the PATIS incremental floorplanner algorithms, followed by the algorithms themselves. The input and output file formats of each tool are also described. Several PlanAhead functionalities are integrated with the PATIS floorplanner in the complete flow.

4.1 Functional Flow

The PATIS incremental floorplanner operates in two modes, as shown in Figure 4.1. 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.

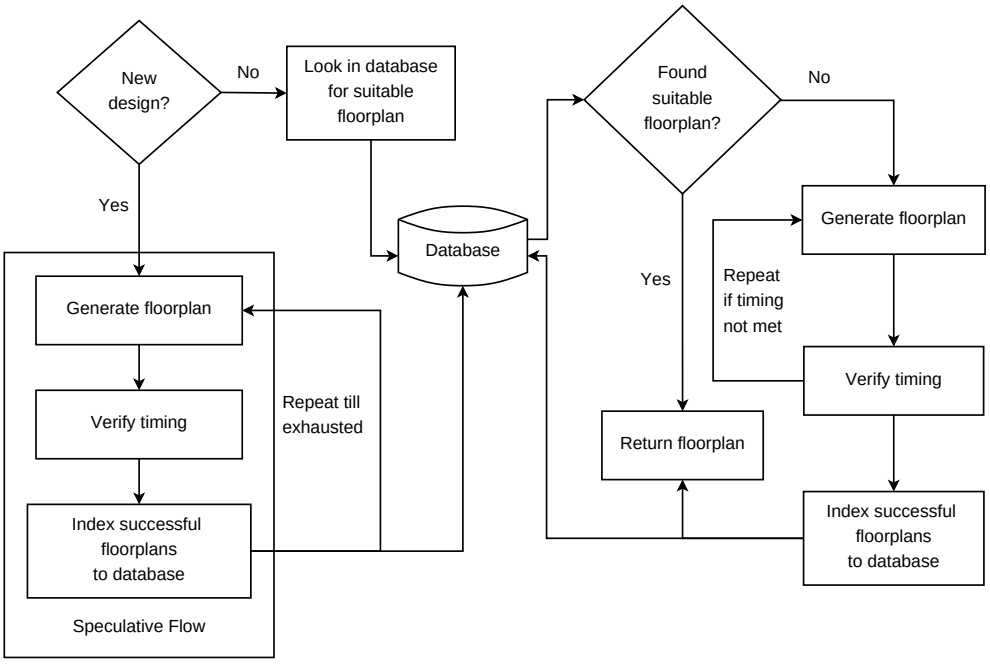

Figure 4.1: Incremental and speculative modes

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. 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 are generated by shifting each vertical edge of a module until prevented by the minimum resource constraint of the adjacent modules. Once all possible aspect ratios are explored, a horizontal edge is moved and the process repeats.

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 behavior.

Incremental floorplanning of a module, both speculative and otherwise, is iteratively performed by two algorithms: *white space occupation* and *neighbor displacement*. 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.

4.2 Implementation Flow

The implementation flow of the PATIS incremental floorplanner is shown in Figure 4.2. The design is synthesized using XST with the hierarchy retained. Setting the `Keep Hierarchy` switch in the XST Synthesis options preserves the design hierarchy during synthesis. This option avoids the lengthy and cumbersome manner of synthesizing each module in an individual project, as done in the conventional PR flow.

An EDIF generator and parser tool converts the XST-generated NGC netlist file into the EDIF format. Once the EDIF is available, this tool parses the file to extract module- and interconnect-specific information. The resource estimator then obtains the resource usage for the design and for each module.

A resource map generator helps PATIS to customize its floorplan to the chip layout. This

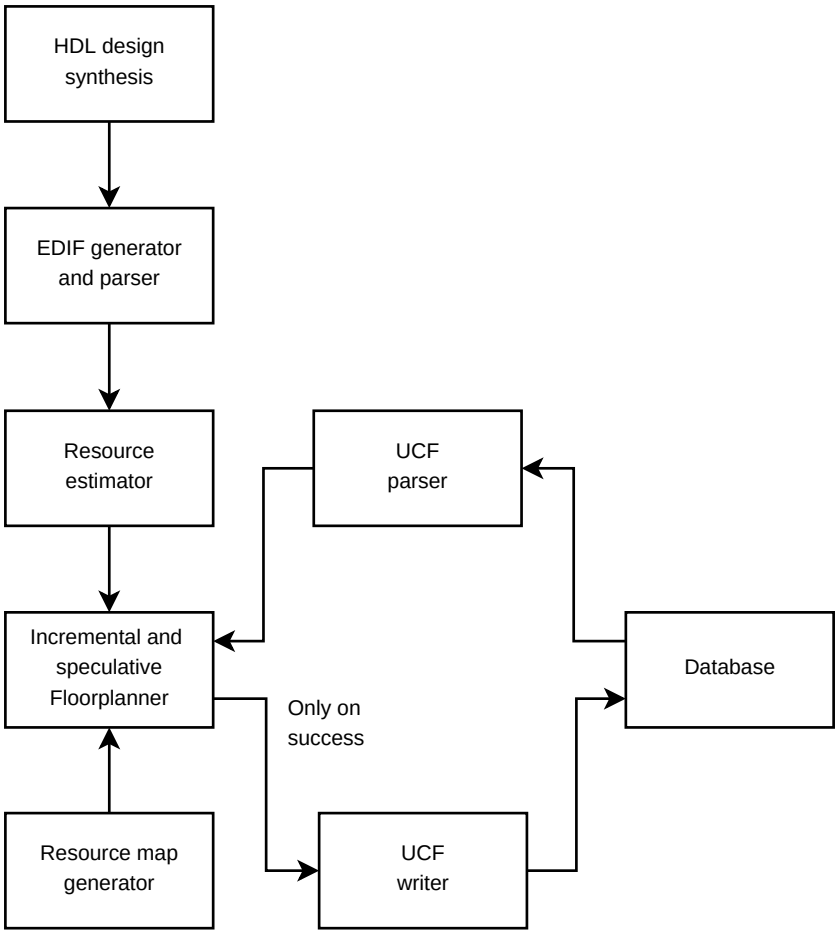

Figure 4.2: Incremental floorplanner implementation

tool generates a map of the FPGA chip with information about the name, number, type and RPM coordinates of resources such as CLB, BRAM and DSP slices, and later, the names of the modules possibly using each resource. The PATIS incremental floorplanner reads in UCF files when the speculative floorplanner needs an initial floorplan and when a design change necessitates a search for a suitable floorplan from the speculative database. For this purpose, a UCF parser extracts the module names and placement information from a UCF file.

The UCF files are picked from the database containing the UCF generated by the automatic and incremental floorplanners. To generate a UCF file from floorplans created by the PATIS incremental floorplanner, a UCF writer tool performs the reverse operation of the

UCF parser. Using the information generated by the floorplanner, the UCF writer writes a complete UCF file with names of the modules, area groups, and placement details. All generated UCF files have their timing verified.

4.2.1 EDIF Generator and Parser

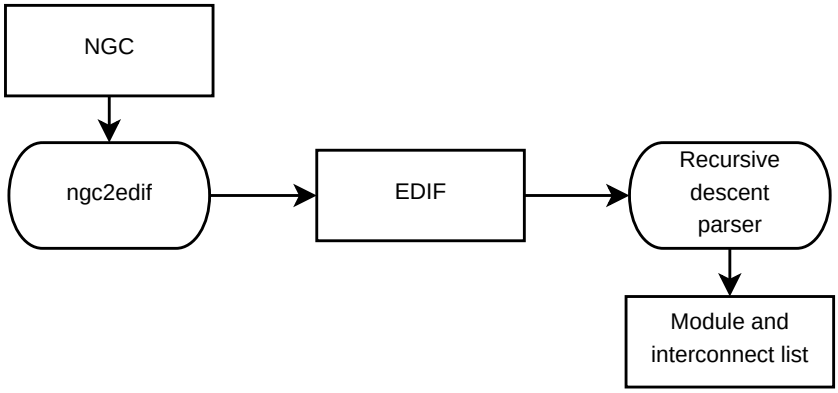

Figure 4.3: EDIF generator and parser

PlanAhead works with EDIF netlist files. The Xilinx ISE tools however generate netlists in NGC file format. A PATIS team member developed an EDIF generator and parser tool to convert the NGC files to EDIF and also to extract relevant information from the EDIF files, as shown in Figure 4.3. The first step of this tool is to convert the generated NGC file into EDIF using `ngc2edif`, whose command-line Backus-Naur Form (BNF) [27] and example are shown in Figure 4.4. `ngc2edif` is a utility created by Xilinx to convert their proprietary NGC netlist format into EDIF.

EDIF is a vendor-neutral format that can be used across EDA systems. It contains information about the chip name, I/O pins, and instance names. A typical EDIF file is shown in Figure 4.5, illustrating a module instantiation and an inter-module net. For module instances, the more important information contained within the EDIF are the port names and widths. Also note the value of the `Keep Hierarchy` flag that is essential for modular and PR flows. The net `inputbusy1` connects port `inputbusy` of module `sub_cfft1` to port `start` of

```
BNF
---
<ngc2edif-command> ::= "ngc2edif" <ngc> <edif>
<ngc> ::= <path> "/" <ngc-name> ".ngc"
<edif> ::= <ngc-name> ".edf"

Example
-------
ngc2edif C://xilinx/sample_project/samplengc.ngc samplengc.edf
```

Figure 4.4: Command-line BNF and example of Xilinx `ngc2edif` utility

module `sub_cfft_2`.

```
//module instance in EDIF format
(instance sub_cfft1
     (viewRef view_1 (cellRef sub_cfft1_sub_cfft1 (libraryRef top_lib)))
     (property BUS_INFO (string "14:OUTPUT:Iout<13:0>") (owner "Xilinx"))
     (property BUS_INFO (string "14:OUTPUT:Qout<13:0>") (owner "Xilinx"))
     (property BUS_INFO (string "12:INPUT:Iin<11:0>") (owner "Xilinx"))
     (property BUS_INFO (string "12:INPUT:Qin<11:0>") (owner "Xilinx"))
     (property KEEP_HIERARCHY (string "TRUE") (owner "Xilinx"))
     (property NLW_UNIQUE_ID (integer 0) (owner "Xilinx"))
     (property NLW_MACRO_TAG (integer 1) (owner "Xilinx"))
     (property NLW_MACRO_ALIAS (string "sub_cfft1_sub_cfft1") (owner "Xilinx"))
)

//NET instance, this net connects sub_cfft1 & sub_cfft_2)
(net inputbusy1
     (joined
          (portRef inputbusy (instanceRef sub_cfft1))
          (portRef start (instanceRef sub_cfft_2))
     )
)
```

Figure 4.5: Module and net instances in a sample EDIF file

EDIF files are versatile and can be read by text browsers, unlike NGC. Once the EDIF file is generated, the EDIF generator and parser extracts module and interconnect information such as number of modules, their names, and the interconnects between them. EDIF is parsed in a recursive descent manner to generate a list of modules and nets. Using this list, the parser generates a hypergraph whose vertices represent the design modules and

hyperedges indicate the inter-module nets.

4.2.2 Resource Estimator

Once the EDIF is generated, the resource estimator tool proceeds to use the EDIF in conjunction with PlanAhead to determine the resource usage of the design. PATIS floorplans in accordance with the resource utilization results that this tool provides. Over-estimation can result in larger module size than necessary, leading to area wastage. Under-estimation can result in overly constrained modules, or modules that do not satisfy resource or timing constraints, thereby increasing the place-and-route times or the possibility of unroutes.

Resource allocation makes use of PlanAhead's resource estimation functionality [28] for accurate estimation. The module information within the EDIF file is used by PlanAhead to determine the top-level resource usage as well as the individual module resource requirements. To facilitate this, PlanAhead, which is invoked using Tool Command Language (TCL) scripts, first creates a custom project using just the top-level NGC file. Next the resource estimator creates the TCL scripts to extract and write each module's resource utilization to an individual text file.

This text file contains module-specific information such as number and type of each required resource, number of carry chains, and number of each type of logical resource assigned to a module. A sample text file is shown in Figure 4.6. PATIS chiefly uses the `Physical Resources Estimates` section to get the names and counts of the resources required by each module. Once it is floorplanned, the `Available` column lists the number of resources allocated. The `% Util` column helps to get a measure of what percent of resources is being used by a module.

```
Physical Resources Estimates
============================

Site Type  Available  Required  % Util
----------------------------------------
LUT        0          7239      No Sites
FF         0          6516      No Sites
SLICEL     0          4416      No Sites
RAMB16     0          12        No Sites


Carry Statistics
================

Type
--------------------------------------------------------------------------------
Number of carry chains    330
Longest chain             pblock_module0/pblock_module26/sub_cfft_2/cfft1/
  amulfactor/u1/u1/gen_pipe[10].Pipe/Maddsub_Xresult_cy<0>
Carry height utilization  N/A (5 CLBs)


Clock Report
============

Domain (Module)    Resource  Instances
----------------------------------------
lclk_BUFGP( top )  Global    6540


Net Statistics
==============

Boundary-crossing Nets
----------------------
52
```

Figure 4.6: A sample text file generated by the resource estimator

4.2.3 Resource Map Generator

Before a tool can commence floorplanning, the architecture of the FPGA chip and the resource distribution must be known. The resource map generator was created by another team member to read in the name of the chip, and obtain the architecture Extensible Markup Language (XML) and Xilinx Design Language (XDL) files from PlanAhead. The XML file

contains an FPGA chip's site names and their RPM placements. An RPM [29] is a collection of logic resources constituting a group. The location of the resources within a group, relative to the other resources of the same group, is controlled by Relative Location Constraints (RLOCs). By using relative constraints, designers need not specify the absolute locations of primitives on a chip. A portion of the XML file for the Virtex-4 FX100-ff1517 part is shown in Figure 4.7.

```
<Sites Part="4vfx100ff1517">
   ...
   <Site Name="AW22" IOB="IPAD_X0Y2" X="2" Y="16"/>
   <Site Name="AW21" IOB="IPAD_X0Y3" X="2" Y="17"/>
   <Site Name="AW25" IOB="OPAD_X0Y0" X="3" Y="16"/>
   <Site Name="AW24" IOB="OPAD_X0Y1" X="3" Y="17"/>
   ...
   <Site Name="SLICE_X0Y0" X="5" Y="0"/>
   <Site Name="SLICE_X0Y1" X="5" Y="1"/>
   <Site Name="SLICE_X0Y2" X="5" Y="2"/>
   <Site Name="SLICE_X0Y3" X="5" Y="3"/>
   ...
   <Site Name="RAMB16_X0Y0" X="13" Y="0"/>
   <Site Name="FIFO16_X0Y0" X="13" Y="1"/>
   <Site Name="RAMB16_X0Y1" X="13" Y="8"/>
   <Site Name="FIFO16_X0Y1" X="13" Y="9"/>
   ...
   <Site Name="DSP48_X0Y0" X="113" Y="0"/>
   <Site Name="DSP48_X0Y1" X="113" Y="1"/>
   <Site Name="DSP48_X0Y2" X="113" Y="8"/>
   <Site Name="DSP48_X0Y3" X="113" Y="9"/>
   ...
</Sites>
```

Figure 4.7: A sample XML file showing site information for Virtex-4 FX100-ff1517

XDL is a fully-featured physical design language that is useful for users to create their own implementation tools [30]. XDL describes the physical layout of designs, site names and locations, and hard macros. A snippet of the XDL is shown in Figure 4.8. Tile details, such the logic elements that compose a tile, and pin information and connection details of primitives such as BUFG and BSCAN, are contained within the XDL.

Using the XDL and XML files, the map generator creates a class representation of the chip.

```
(tiles 181 177
    ...
    (tile 1 4 CLB_X1Y159 CLB 4
        (primitive_site SLICE_X0Y318 SLICEM internal 34)
        (primitive_site SLICE_X0Y319 SLICEM internal 34)
        (primitive_site SLICE_X1Y318 SLICEL internal 27)
        (primitive_site SLICE_X1Y319 SLICEL internal 27)
    )
    (tile 1 5 INT_X2Y159 INT 1
        (primitive_site TIEOFF_X2Y159 TIEOFF internal 3)
    )
    (tile 1 6 CLB_X2Y159 CLB 4
        (primitive_site SLICE_X2Y318 SLICEM internal 34)
        (primitive_site SLICE_X2Y319 SLICEM internal 34)
        (primitive_site SLICE_X3Y318 SLICEL internal 27)
        (primitive_site SLICE_X3Y319 SLICEL internal 27)
    )
    ...
)
(primitive_defs 41
    ...
    (primitive_def BUFG 2 3
        (pin I0 I0 input)
        (pin O O output)
        (element O 1
            (pin O input)
            (conn O O <== GCLK_BUFFER OUT)
        )
        (element GCLK_BUFFER 2 # BEL
            (pin IN input)
            (pin OUT output)
            (conn GCLK_BUFFER OUT ==> O O)
            (conn GCLK_BUFFER IN <== I0 I0)
        )
        (element I0 1
            (pin I0 output)
            (conn I0 I0 ==> GCLK_BUFFER IN)
        )
    )
    ...
)
```

Figure 4.8: A sample XDL file showing tile information for Virtex-4 FX100-ff1517

PATIS uses RPM coordinates to map a module to a particular location. This tool supports certain utility functions such as obtaining nearest RPM coordinates containing a particular resource type, and determining the occupancy of a particular site.

4.2.4 UCF Parser

Each time the incremental floorplanner is called, a UCF file has to be parsed before the design can be re-floorplanned. This requires first getting a list of modules from the UCF file and extracting their names and location details. Once these details are obtained, PATIS instantiates objects for each module added to a floorplan class for that design.

The BNF for the area group statements in the UCF is shown in Figure 4.9. `RANGE`, `MODE` and `AREA_GROUP` are the main components of this user constraint. A typical UCF entry for a module looks as in Figure 4.10. This UCF entry corresponds to the placement of module `sub_cfft_3` in Figure 4.11.

```
<pr-module-loc> ::= <range> <mode> <instance>
<range> ::= "AREA_GROUP" <group-name> "RANGE=" <resource-range> ";" <EOL>
<mode> ::= "AREA_GROUP" <group-name> "MODE=RECONFIG;" <EOL>
<instance> ::= "INST" <instance-name> "AREA_GROUP = " <group-name> ";" <EOL>
```

Figure 4.9: BNF of a UCF file

Each module is associated with an `AREA_GROUP`, which is in turn linked with the `RANGE`. The associated area group for each module parsed from the UCF file is determined. This is followed by a verification that the module is reconfigurable; if it is, then the placement coordinates of the module are obtained. The coordinates thus obtained are used to create the `RpmGrid` module objects and added to the `RpmGridFloorplan` class.

```
AREA_GROUP "pblock_sub_cfft_3" RANGE=SLICE_X0Y104:SLICE_X43Y127,
                                     SLICE_X34Y64:SLICE_X43Y103,
                                     SLICE_X0Y64:SLICE_X19Y103;
AREA_GROUP "pblock_sub_cfft_3" RANGE=RAMB16_X0Y13:RAMB16_X3Y15,
                                     RAMB16_X3Y8:RAMB16_X3Y12,
                                     RAMB16_X0Y8:RAMB16_X0Y12;
AREA_GROUP "pblock_sub_cfft_3" MODE=RECONFIG;
INST "sub_cfft_3" AREA_GROUP = "pblock_sub_cfft_3";
```

Figure 4.10: An entry in a UCF file

The UCF file records the module boundaries in terms of resource coordinates. PATIS however

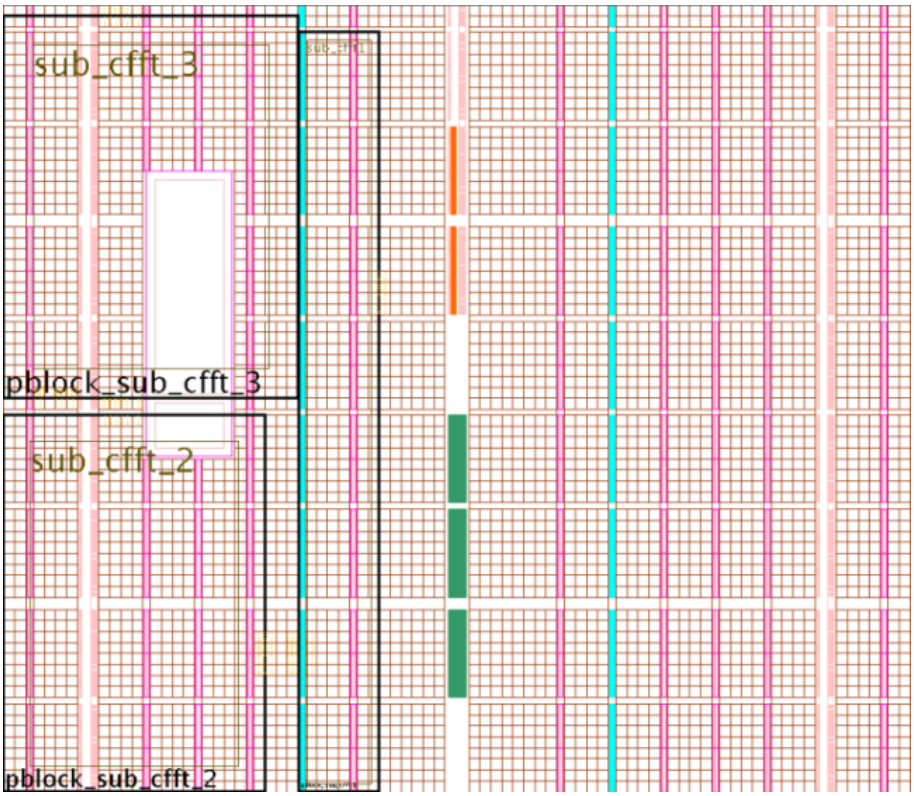

Figure 4.11: Sample floorplan
(*Screenshot captured from PlanAhead tool*)

uses RPM coordinates, and thus the resource map generator determines the equivalent RPM coordinates corresponding to each corner coordinate. The overall module boundary will be rectangular although PR allows modules of any shape [31].

4.2.5 Floorplan Database

The speculative floorplanner generates multiple floorplans by modifying the current floorplan to produce variants with different resource counts. To better organize these designs and to have easy access to various features of a design (such as information about the floorplan currently used and the number of floorplans generated for a project), an indexing of these designs is required. In addition, such an indexing mechanism ensures faster access for the PATIS tool when a design change has occurred. The database is used to store the UCF file

locations instead of the file contents to avoid large data transfer to and from the database.

Since the PATIS tool only requires an indexing mechanism to store certain details about each project, an SQLite3 database [32] is used. While the floorplans themselves will be stored in a local directory, the PATIS tool, specifically the incremental floorplanner, fetches these designs after querying their address from the database.

The PATIS database is a lightweight SQLite3 database that uses very little memory and is easy to use. SQLite3 was chosen for the following reasons:

- It requires no setup or administration, thus ensuring that the HDL designer does not have to worry about the technicalities of the database.
- It is written in ANSI-C. Since the PATIS tool is also being developed in C++, the interfacing between the database and the PATIS tool is simplified.
- The database itself is a single file, which can also be used across several platforms.
- It is server-less, thus ensuring that there is no overhead caused by any communications with a server to fetch details of a design. This is a significant advantage for the PATIS tool since the database stores minimal design information. The use of a more powerful database is not justified.

The PATIS database indexes all projects and their designs in two tables: `Project_List` and `Floorplan_List`. The `Project_List` table enumerates all the projects created by the tool, assigning each a unique ID. The decision process of whether to call the automatic floorplanner or incremental floorplanner for a particular design is based on whether it has already been indexed in the `Project_List` table. The `Floorplan_List` table contains floorplans created for each project and stores their locations, thus facilitating faster access to the floorplans. The structure of the two tables is shown in Tables 4.1 and 4.2.

In addition to these two tables, the PATIS database has another table type called the `Resource_List`, shown in Table 4.3. One `Resource_List` table is created for each de-

Table 4.1: Project_List table

Project Name	**Project ID**	**Author**	**Current Floorplan Version**
FFT	1	ccm	3
Viterbi Decoder	2	ccm	2
Video Filter	3	ccm	6
MicroBlaze Ring	4	ccm	4

Table 4.2: Floorplan_List table

Project ID	**Floorplan Name**	**Floorplan Path**	**Floorplan Version**
1	1_1.ucf	/1/floorplans/	1
1	1_2.ucf	/1/floorplans/	2
1	1_3.ucf	/1/floorplans/	3
2	2_1.ucf	/2/floorplans/	1
2	2_2.ucf	/2/floorplans/	2
...	...	...	...

sign, with one column for each module. This table tracks the CLB slice resource counts of each module in the corresponding design. Addition or deletion of modules causes the table columns to be updated as well. This is done by an initial scrutiny step that matches the column names with module names to ensure a match.

The `Resource_List` table is scanned during incremental floorplanning to choose floorplans that meet the new resource requirements. This strategy proves to be especially helpful when there are several floorplans generated per design. In such a scenario, it is not practical to scan through every design while looking for a suitable match. This columnar indexing of resource counts helps PATIS choose those designs that meet the new slice requirements and then analyze the chosen designs to check the counts of the other resources such as DSP and BRAM slices. In the `Resource_List` table, only CLB slice counts are recorded because they are the most commonly available resources in an FPGA chip and a design. Once the search space is narrowed, the PATIS tool verifies the counts of the other resources.

Table 4.3: Resource_List table

Floorplan Name	**Module One**	**Module Two**	**Module Three**
1_1.ucf	800	1200	800
1_2.ucf	800	1800	800
1_3.ucf	1400	1200	1200

The PATIS database API supports several utility functions such as adding new entries, assigning a unique ID to each project, and assigning version numbers incrementally to each floorplan of a design. The version of the currently used floorplan is also indexed in the `Project_List` table. At setup, one-time functions are called to create the database and the tables. Whenever the PATIS tool creates a timing-verified floorplan, it is indexed into the `Floorplan_List` table shown in Table 4.2. If this is a new design, a `Resource_List` table is created. One column is added in the `Resource_List` table for each module. For old designs, the previously created `Resource_List` table is used. For each module, the CLB resource counts are recorded in this table.

4.2.6 UCF Writer

The UCF writer performs the reverse function of the UCF parser. PATIS uses PlanAhead to produce the UCF files for the generated floorplans after supplying a module's bounding box details. A script is then run calling PlanAhead to generate the corresponding UCF file for the design. PlanAhead's knowledge of the architecture of many Xilinx FPGAs makes it easy to map floorplans to FPGA chips even if the floorplanning is done external to PlanAhead. The UCF file generated contains the range of all resources, including but not restricted to CLB, DSP and BRAM slices. The location address of this UCF file is then stored in the database, described in Section 4.2.5. Figure 4.10 shows the UCF entry generated corresponding to the `sub_cfft_3` module in Figure 4.11.

4.3 Strategies

PATIS algorithms are based on several strategies that make the process of incremental floorplanning easier and also result in valid feasible floorplans. These strategies, listed below, were derived from the proof-of-concept experiments described in Section 3.9.

1. A configuration frame in a Virtex-4 FPGA has a height of 16 CLBs (4 slices of BRAMs, 8 slices of DSPs, and 16 CLB slices) and a width of 1 CLB. In a Partial Reconfigurable design, a frame cannot be occupied by more than one module. Hence, even if one CLB in a frame is allotted to a module, another module cannot share the same frame. This implies that allocating a whole frame to a module ensures that the entire frame is fully utilized. Moreover, if a module requires very few resources, filling out a whole frame height could sometimes make readily available extra resources for potential changes. This is demonstrated in Figure 4.12, where (i) shows the illegal layout in which a few configuration frames are shared between `pblock_1` and `pblock_2`. Instead, the blocks should be placed as in (ii), which show them on non-overlapping sets of configuration frames.

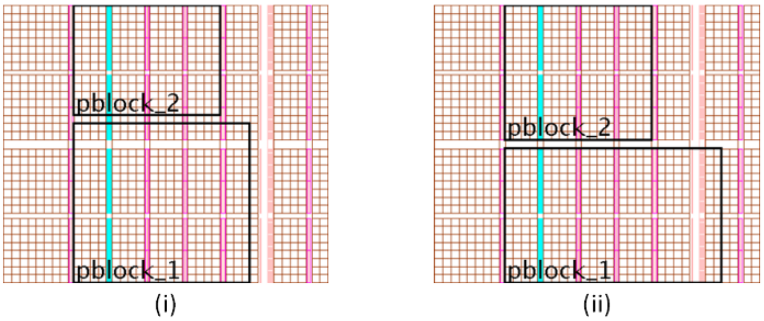

Figure 4.12: Sharing of configuration frames
(*Screenshot captured from PlanAhead tool*)

2. Re-floorplanning a design mainly requires widening a module, and possibly narrowing an adjacent module. To allocate more resources to a module, a module can be stretched along any of its four sides. However, since a whole frame has to be allocated

to a module, stretching it along its top or bottom sides will require the module to occupy an additional block of 16 CLBs (in Virtex-4). Hence, to ensure better resource management, the module should first be stretched along its right or left sides and then along its top and bottom sides, as illustrated in Figure 4.13.

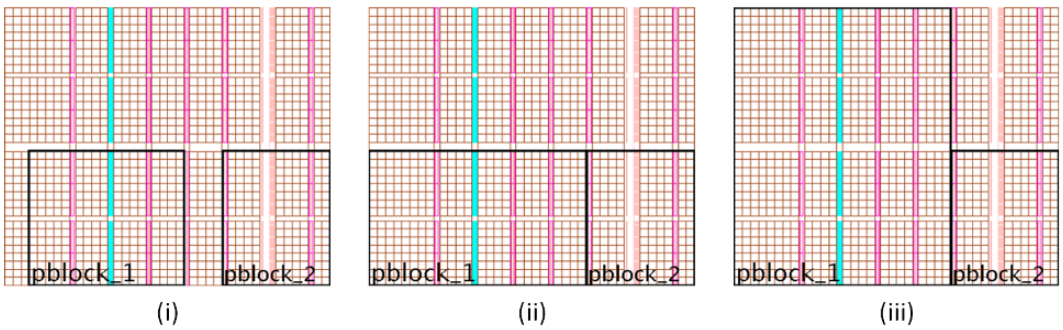

Figure 4.13: Strategies for module edge movement
(*Screenshot captured from PlanAhead tool*)

3. The time-consuming task in a PR design is the placement of bus macros. Sometimes, re-floorplanning can necessitate changing the location of the bus macros. This can be avoided by placing bus macros on those sides of the module that are least likely to be modified by incremental floorplanning. Since a whole configuration frame can be assigned to a module, and re-floorplanning will initially involve stretching or contracting the module along its sides, the bus macros can be placed on the top and bottom of the module. This is shown in Figure 4.14.

4. Modules may need to be re-floorplanned for any number of reasons, the most common among them is the need for more resources. Each time a module's placement is modified, any floorplanning tool has to ensure that none of its resource requirements are violated. In addition, knowledge of the layout of the target can help the designer customize the floorplanner algorithms. Typically, DSP and BRAM resources are located in columns. Modules that require more DSP / BRAM slices should grow longitudinally where possible. A lateral growth can end up assigning too many CLB slices to the module, leading to area wastage. As a corollary, modules that require both CLB

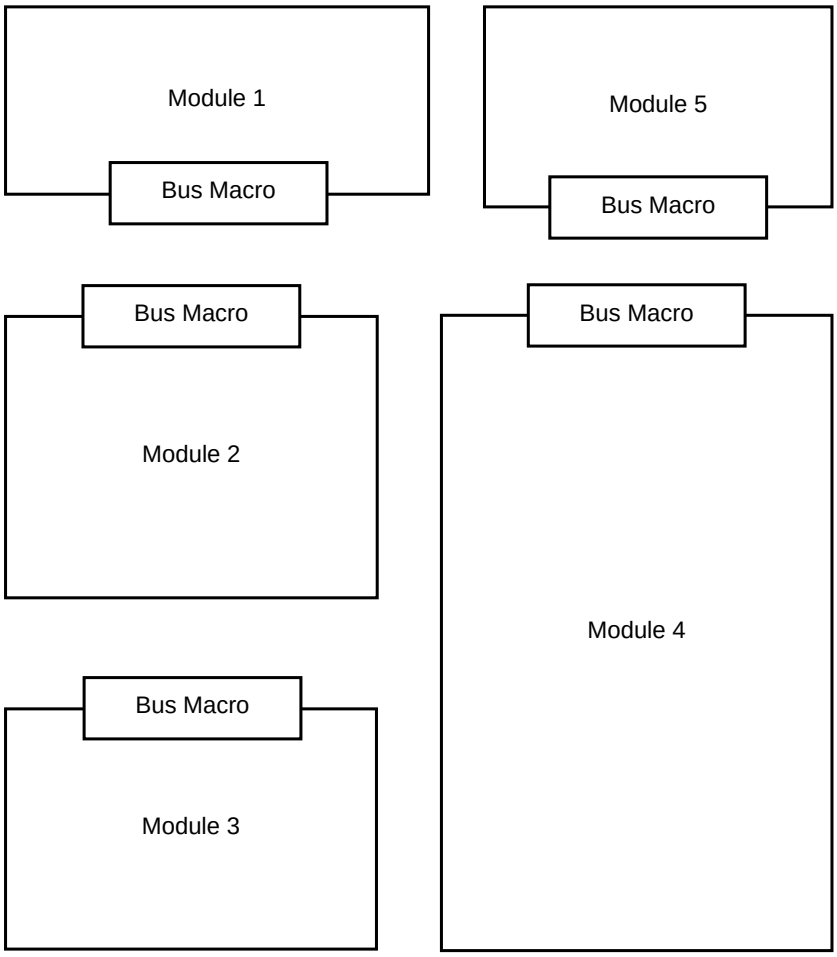

Figure 4.14: Bus macro placement on module boundaries

and DSP/BRAM slices can be allocated more slices first, since it is possible that extra DSP/BRAM slices may be added to the module in the process.

5. Another important decision deals with the quantity of extra resources that can be allocated to a PR module. Viscosity of a module is a measure of how frequently a module may be modified, but not how widely a module can change in terms of the resource requirement vector. Speculative floorplanning helps in this situation by iteratively allocating up to a certain percent of extra resources to a module, thus making available a large number of variants.

4.4 Algorithms

The PATIS incremental floorplanner works on the principle of minimum modification to the floorplan layout by updating modules locally. When any module is to be changed, PATIS looks at the region around the module for unoccupied resources and neighboring modules. PATIS then focuses its attention to this limited region so as to avoid ripple effects of large floorplan changes.

PATIS floorplans incrementally or speculatively through two algorithms — *White Space Occupation* and *Neighbor Displacement*. Both algorithms focus on locally modifying a floorplan rather than disturbing the whole floorplan, thereby avoiding any drastic changes. The floorplanner looks at the region immediately around the module to be re-floorplanned. Any white space available is first allocated to that module before attempting to modify any neighbors.

4.4.1 White Space Occupation

As described, the PATIS incremental floorplanner first looks for white space 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. The white space algorithm, shown in Figure 4.15, generates the largest possible module configuration that meets the new resource requirements. This is done subject to the constraint that the module's width-to-height aspect ratio stays within 0.5 and 2.0, which prevents excessive module growth in one direction. However, this restriction is relaxed when a module has still not met its resource requirements and there are available resources in the vicinity.

The white space algorithm in Figure 4.15 executes until the new resource requirements are met or the tool has explored all possible aspect ratios (line 5). There are two possible growth functions. In lines 8–11, PATIS executes the vertical edge movement function, `doVertEdgeMvt()`, moving either the left or right edges, while keeping the module aspect ratio between 0.5 and 2.0. This function first determines the number of slice columns re-

```
1:  Procedure: WhiteSpaceMovement (grid, resourceReqt)
2:
3:  done = false;
4:  exhausted = false;
5:  while (!done & !exhausted) do
6:      conflict = false ;                                   /* Vertical movement */
7:      aspRatio = getWidth(grid) / getHeight(grid);
8:      while (0.5 < aspRatio < 2 & !conflict & !done) do
9:          grid = doVertEdgeMvt(&aspRatio, &conflict);
10:         if (getResourceReqt(grid) < getResourceCount(grid)) then
11:             done = true;
12:     if (conflict) then
13:         noMoreVertMvt = true;
14:     conflict = false ;                                 /* Horizontal movement */
15:     aspRatio = getWidth(grid) / getHeight(grid);
16:     while (0.5 < aspRatio < 2 & !conflict & !done) do
17:         grid = doHorizEdgeMvt(&aspRatio, &conflict);
18:         if (getResourceReqt(grid) < getResourceCount(grid)) then
19:             done = true;
20:     if (conflict) then
21:         noMoreHorizMvt = true;
22:     exhausted = noMoreVertMvt & noMoreHorizMvt;
```

Figure 4.15: White space occupation algorithm

quired for the module. Should there be as many free columns available to the module, PATIS enlarges the module by that number. If unsuccessful, the module is enlarged by as many free columns as there are beside the module. If at any point the current module overlaps with another module or a configuration frame is being shared, thus setting the `conflict` flag in line 9, PATIS rolls back the last change.

In a similar manner, lines 16–19 show PATIS executing the horizontal edge movement function, `doHorizEdgeMvt()`, which moves either the top or bottom edges. At the same time, aspect ratio is maintained at between 0.5 and 2.0. During horizontal edge movements, PATIS makes sure that configuration frames are not shared. It extends the module by a whole configuration frame height, and looks for potential conflicts in line 17. In case of conflicts either in the `doVertEdgeMvt()` (lines 12–13) or the `doHorizEdgeMvt()` function (lines 20–21), the flag `noMoreVertMvt` or `noMoreHorizMvt` will be set. When both flags are set,

PATIS has exhausted all options, and will proceed to execute the Neighbor Displacement algorithm. Satisfaction of the new resource requirements causes PATIS to exit the white space algorithm.

The white space algorithm is demonstrated in Figure 4.16, where a Virtex-4 FX100-ff1517 is used as the base device. In the initial floorplan shown in (i), `pblock_1` is the module to which more resources are to be allocated. The first step is to execute the vertical edge movement function, thus assigning nearby free resources to `pblock_1` as shown in Figure 4.16 (ii). Since no more free resources are available along `pblock_1` module's left and right edges, the horizontal edge movement function will be triggered. This will cause the top edge of the module to be moved above by one configuration height. The final floorplan is shown in (iii).

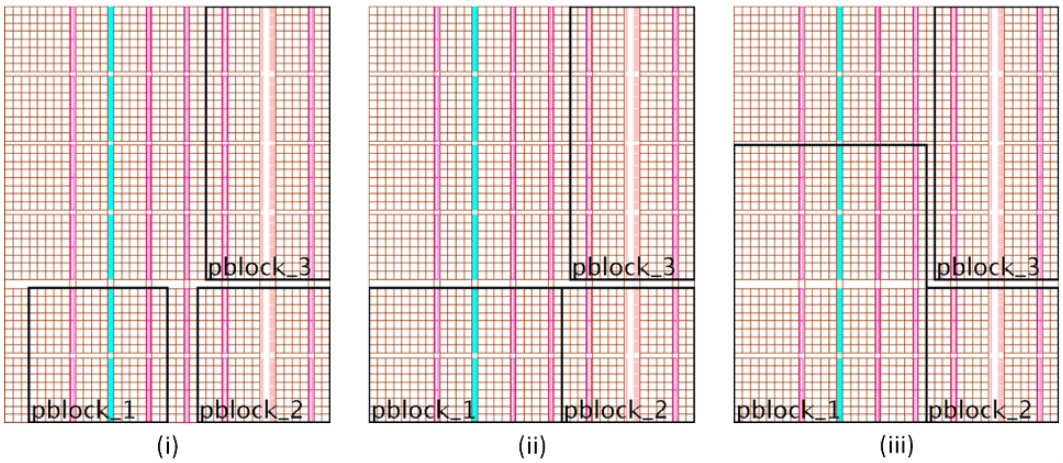

Figure 4.16: White Space occupation algorithm illustration
(*Screenshot captured from PlanAhead tool*)

4.4.2 Neighbor Displacement

The second phase of the incremental floorplanner modifies neighboring modules in addition to the current module, as shown in Figure 4.17. Adjacent module edges are shifted away from the module being floorplanned while still respecting their resource requirements. This continues until no further modification to the floorplan is possible, excluding any drastic changes to the topology. The neighbor displacement algorithm performs translation and

shrink operations.

```
1:  Procedure: NeighborDisplacement(grid, resourceReqt)
2:
3:  done = false;
4:  exhausted = false;
5:  while (!done & !exhausted) do                                   /* translate */
6:      conflict = false;
7:      neighbor = getConfictingGrid(grid);
8:      if (neighbor ≠ NULL) then
9:          newneighbor = translate(neighbor, &conflict);
10:         if (!conflict & getResourceReqt(neighbor) < getResourceCount(newneighbor)) then
11:             neighbor = newneighbor;
12:             grid = updateGrid(grid, neighbor);
13:         else
14:             exhausted = true;
15:     else
16:         exhausted = true;
17:     if (getResourceReqt(grid) < getResourceCount(grid)) then
18:         done = true;
19: while (!done & !exhausted) do                                    /* shrink */
20:     neighbor = getConfictingGrid(grid);
21:     if (neighbor ≠ NULL) then
22:         neighborReqt = getResourceCount(neighbor);
23:         newneighbor = shrink(neighbor);
24:         if (getResourceReqt(neighbor) < getResourceCount(newneighbor)) then
25:             neighbor = newneighbor;
26:             grid = updateGrid(grid, neighbor);
27:         else
28:             exhausted = true;
29:     else
30:         exhausted = true;
31:     if (getResourceReqt(grid) < getResourceCount(grid)) then
32:         done = true;
```

Figure 4.17: Neighbor displacement algorithm

Module Translation

A neighbor module can be translated by a certain number of configuration frames while still satisfying its resource requirements. This is done when there are free resources on the side of the neighbor module in the direction of the translation. The translation frees up resources

between the module being floorplanned and the neighbor module. Translation requires that there be no conflicts for the neighbor module in the direction of movement.

Translation does not affect the neighbor modules' aspect ratio, but may change their resource count. This could happen because translating a module typically changes the resource organization. The resources whose counts are most likely affected are DSPs and BRAMs, since they are located along limited columns in the chip. Should a negative resource count change occur, the tool rolls back the last translation applied to the affected module. Translation in the top and bottom directions move the neighbor module by a height equal to one configuration frame. This usually proves fruitful for the topmost or the bottommost modules, since for other modules, there is a chance of conflict due to modules stacked below or above.

Lines 5–19 in Figure 4.17 show the translation procedure. PATIS first gets a list of neighbors (line 7). In the absence of any immediate neighbor, PATIS cannot do neighbor displacement (lines 16–17). For each available neighbor, translation is done (line 10), such that the neighbor does not overlap with other modules and also still satisfies its resource count. Success causes the current module and the neighbor boundaries to be updated (lines 11–13), followed by a check of its resource requirements in lines 18–19.

The translation methodology is shown in Figure 4.18 on a Virtex-4 FX100-ff1517 FPGA chip, where (i) shows the initial floorplan. The first run of the translate function causes `pblock_2` to move to the far right, and the freed up resources are allocated to `pblock_1`, as shown in Figure 4.18 (ii). The next run of the translate function moves `pblock_3` up by a height of one configuration frame. Extending the top edge of `pblock_1` upwards, leads to the floorplan in (iii).

Module Shrink

In the event that a neighbor has nearby modules that prevent translation, a shrink operation is applied. The shrink shifts inward the edge of the neighbor module nearer the module being floorplanned, thereby constraining its resource count further. This frees up resources between

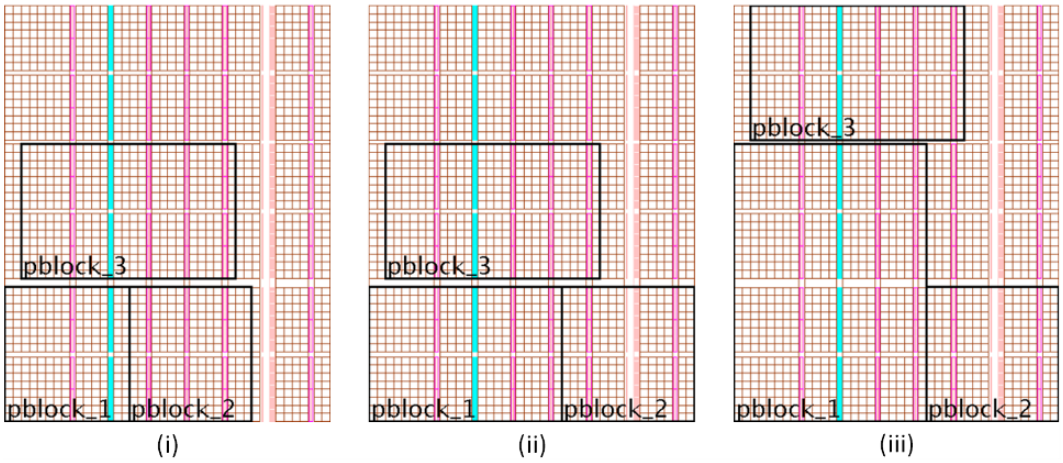

Figure 4.18: Neighbor displacement algorithm – translate illustration
(*Screenshot captured from PlanAhead tool*)

the module being floorplanned and the neighbor module.

Shrinking a module can reduce its resource count by a significant number depending on its height. For right and left edges, PATIS reduces the size of the neighbor module column by column, and subsequently expands the module being floorplanned. For top and bottom neighbors, shrinking by one row may not prove to be useful since in that case the same configuration frame may be shared. Instead, a neighbor will be shrunk by a height of one configuration frame. This typically involves a reduction of a large amount of resources, and hence has a very low success ratio. Rather than shrinking a top or bottom neighbor upwards or downwards, they are shrunk in the left or right direction — whichever results in the least resource shrinkage. A neighbor module can be shrunk until its resource requirements are no longer satisfied.

As with the translate procedure, the shrink method shown in Figure 4.17 (lines 20–33) starts with a list of neighbors to the module being floorplanned. Absence of neighbors causes the neighbor displacement algorithm to stop (line 30–31), while each available neighbor is shrunk iteratively by one configuration frame column, as shown in line 24. Each time a shrink function is executed, PATIS ensures that the neighbor module still satisfies its resource requirements. The procedure ends when the neighbor can no longer be shrunk or

the current module meets its resource requirements.

The shrink operation is shown in Figure 4.19 on the same Virtex-4 FX100-ff1517 FPGA chip. Figure 4.19 (i) shows the initial floorplan. Since the translate function cannot be applied here, the shrink function executes, reducing the resource count of `pblock_2`. This leads to the floorplan in (ii). `pblock_3` has a conflict above it and hence will be shrunk after allocating its free resources to `pblock_1`, as shown in (iii). Note that during these shrink operations, PATIS makes sure that no module resource requirements are violated.

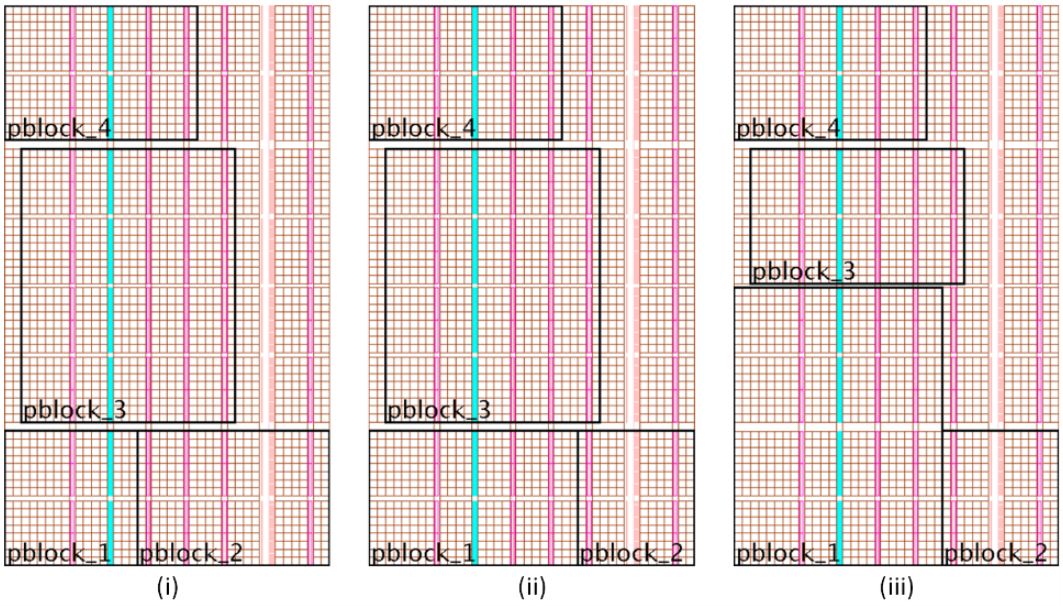

Figure 4.19: Neighbor displacement algorithm – shrink illustration
(*Screenshot captured from PlanAhead tool*)

The greedy incremental floorplanner algorithms are mainly intended to deal with simple but common cases of minor changes to one or two modules. Significant changes to the layout or inter-module timing can possibly result in major ripple effects, affecting other parts of the design. Therefore, such changes are dealt with by the non-greedy automatic floorplanning and bus macro insertion algorithms.

4.5 Data Structures and API functions

Some common data structures and API calls, along with the functions and data structures used in the algorithms are listed below.

4.5.1 EDIF Parser and Resource Estimation

Figure 4.20 shows the data structure used to represent the resource estimates for a module. Encapsulated in this object are its name (`module_name`) and a count of resources of type CLB (`number_of_CLBs`), BRAM (`number_of_BRAMs`), and DSP (`number_of_DSPs`).

```
class instanceMap {
    char* module_name;
    int number_of_CLB;
    int number_of_BRAM;
    int number_of_DSP;
};
```

Figure 4.20: Module list and resource estimation

4.5.2 Resource Map

Each primitive in the resource map is represented in a data structure, shown in Figure 4.21, specifying its name, type and RPM coordinates.

```
class PrimitiveSite {
    std::string name;
    std::string type;
    int rpmX, rpmY;
};
```

Figure 4.21: Details of a primitive

(x_0, y_0, x_1, y_1)

Bottom left corner x- and y- coordinates, and top right corner x- and y-coordinates of a module.

grid

Layout information of a particular module.

`getLeftNeighbor(`*primitive*`)`, `getRightNeighbor(`*primitive*`)`

Gets the primitive to the left or right of a particular *primitive*.

`getUpNeighbor(`*primitive*`)`, `getDownNeighbor(`*primitive*`)`

Gets the primitive above or below a particular *primitive*.

`getResourcesLeft(`*resource_type*`, `*count*`)`, `getResourcesRight(`*resource_type*`, `*count*`)`

Looks for a *count* number of a particular *resource_type* to the left or right of the current primitive.

`getResourcesUp(`*resource_type*`, `*count*`)`, `getResourcesDown(`*resource_type*`, `*count*`)`

Looks for a *count* number of a particular *resource_type* above or below the current primitive.

`createGridInFloorplan(`*module_name*`, `x_0`, `y_0`, `x_1`, `y_1`)`

Creates a grid for a module named *module_name* and defined corner coordinates (x_0, y_0, x_1, y_1).

`deleteGridFromFloorplan(`*module_name*`)`

Deletes a grid with the specified *module_name*.

`getConfictingGrid(`*grid*`)`

Returns any grid that is overlapping with the current module or *grid*.

`isPrimitiveInFloorplan(`*primitive*`)`

Checks if a particular *primitive* is free or occupied.

`getResourceCount(`*grid*`)`

Gets the resource count of a module or *grid*. It returns the values of `number_of_CLBs`, `number_of_BRAMs`, and `number_of_DSPs`.

4.5.3 Floorplan Database

version

Each floorplan indexed into the database has a *version* associated with it, much like a unique ID.

`is_project_new (`*project_name*`)`

Determines if the details of a project named *project_name* already exist in the database.

`add_record_project_list (`*project_name*`, `*id*`, `*version*`)`

Adds to the `Project_List` table a new entry corresponding to a project with name *project_name* and identification *id* and current floorplan *version*.

`add_record_floorplan_list (`*id*`, `*floorplan_name*`, `*floorplan_path*`, `*version*`)`

Adds to the `Floorplan_List` table a new entry for project *id*, corresponding to a floorplan named *floorplan_name* located at *floorplan_path* and having a particular *version*.

`get_project_details (`*project_name*`)`

Fetches details from the database corresponding to a project named *project_name*.

`get_floorplan_details (`*project_name*`, `*version*`)`

Fetches details from the database of a floorplan *version* corresponding to a project named *project_name*.

`get_floorplan_path (`*project_name*`, `*version*`)`

Gets the location address of a UCF file *version* corresponding to a project named *project_name*.

4.5.4 Incremental Floorplanner

done

Indicates if a module's resource requirements are satisfied.

exhausted

Indicates if no more design exploration using an algorithm is possible.

conflict

Indicates whether there is overlap between two modules.

aspRatio

Aspect ratio of the module.

noMoreVertMvt

Indicates that no more vertical edge movement can be done.

noMoreHorizMvt

Indicates that no more horizontal edge movement can be done.

`getWidth(`*grid*`)`

Gets the width of a module or *grid*.

`getHeight(`*grid*`)`

Gets the height of a module or *grid*.

`doVertEdgeMvt(&`*aspRatio*`, &`*conflict*`)`

Executes vertical edge movement on a module and updates *aspRatio* and *conflict*.

`doHorizEdgeMvt(&`*aspRatio*`, &`*conflict*`)`

Executes horizontal edge movement on a module and updates *aspRatio* and *conflict*.

`translate(`*neighbor*`, &`*conflict*`)`

Executes the translate function of the neighbor displacement algorithm, so long as the *neighbor* does not *conflict* with any module.

`shrink(`*neighbor*`)`

Executes the shrink function of the neighbor displacement algorithm with the module *neighbor* as parameter.

`updateGrid(...)`

Updates the coordinates of a module.

4.6 Heterogenous Layouts

Heterogeneous designs have different resource combinations for different modules. Typically, CLBs, BRAMs and DSPs are the most common resources, and PATIS focuses on these resources while floorplanning. The PATIS incremental floorplanner approaches this problem through a multi-pronged strategy.

PATIS first determines the list of modules requiring significantly more DSP or BRAM resources and attempts to meet their requirement first. Because DSP and BRAM resources are arranged in columns, allocating more of these resources causes a module to grow longitudinally until obstructed by a neighboring module or if it runs out of that resource in the direction of growth. If longitudinal growth is not possible, then PATIS attempts to allocate resources through left and right edge movements.

Modules which require both DSP/BRAM resources and CLB slices are initially allocated more slices. Meeting CLB slice requirement before considering the DSP and BRAM requirements generally succeeds. The reason for this ordering is that slices are the most commonly available resources and satisfying the slice requirement generally meets the other resource requirements.

Chapter 5

Results

The DMD flow was executed on a suite of large-scale FPGA designs to compare the run-time advantages of the PATIS flow with a modular non-PR flow. The varied sizes and resource compositions of the designs resulted in several observations about the DMD flow. While improvements were noticed in all designs during implementation, certain properties of the design influenced their behavior.

This chapter begins with an introduction to the applications constituting the benchmark suite and their resource compositions. The timing performance of the designs and speedups obtained are then analyzed. Finally, a study of the PATIS tool behavior and a discussion of the results follow.

5.1 Experimental Setup

5.1.1 Platform Specifications

We conducted our experiments on a four-core 2.66 GHz Core i7-920 CPU with at least 6GB of memory. All benchmarks target a Virtex-4 FX100-ff1517 FPGA [33] using Version 9.2i of the ISE tools with the PR-12 patch. All programs were written in C++, and time durations for the tool runs were determined using the C++ utility `gettimeofday()`.

5.1.2 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. These frequencies have been calculated to the first decimal digit from the clock periods observed on PlanAhead.

1. **Cascaded complex FFTs**: A Fast Fourier Transform [34] is an algorithm to compute the Discrete Fourier Transform (DFT) and its inverse, where 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. The limited number of DSP slices on the Virtex-4 FX100 chip however were quickly exhausted, and hence the FFT computations were implemented on the CLB slices leading to a high CLB utilization. This design consists of several complex FFT (CFFT) 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 three CFFT modules, while the other contained six. The three-module design `CFFT 3`, shown in Figure 5.1, is the smallest in the 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 designed by Xilinx for Xilinx FPGAs [35]. As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of Xilinx FPGAs. The MicroBlaze, which is both CLB- and memory-intensive, can then be programmed for custom applications. A MicroBlaze processor was generated using Xilinx Embedded Development Kit (EDK) with bidirectional FSL interfaces. Several instances of the MicroBlaze were then 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 [36] 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 design, `Viterbi 7`, 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) [37] 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, it focuses more on complex operations such as logarithms, trigonometry and polynomials. 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.

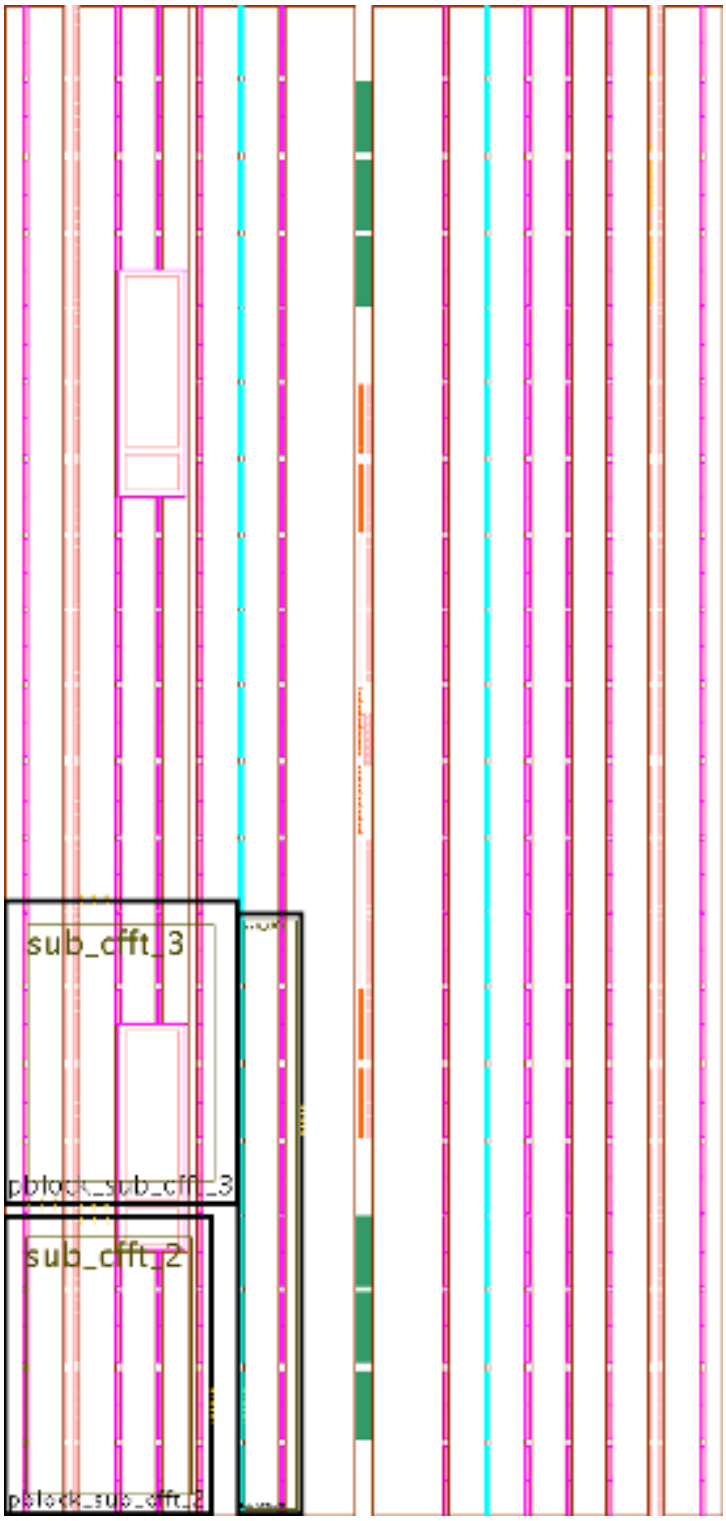

Figure 5.1: Design – `CFFT 3`
(*Screenshot captured from PlanAhead tool*)

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

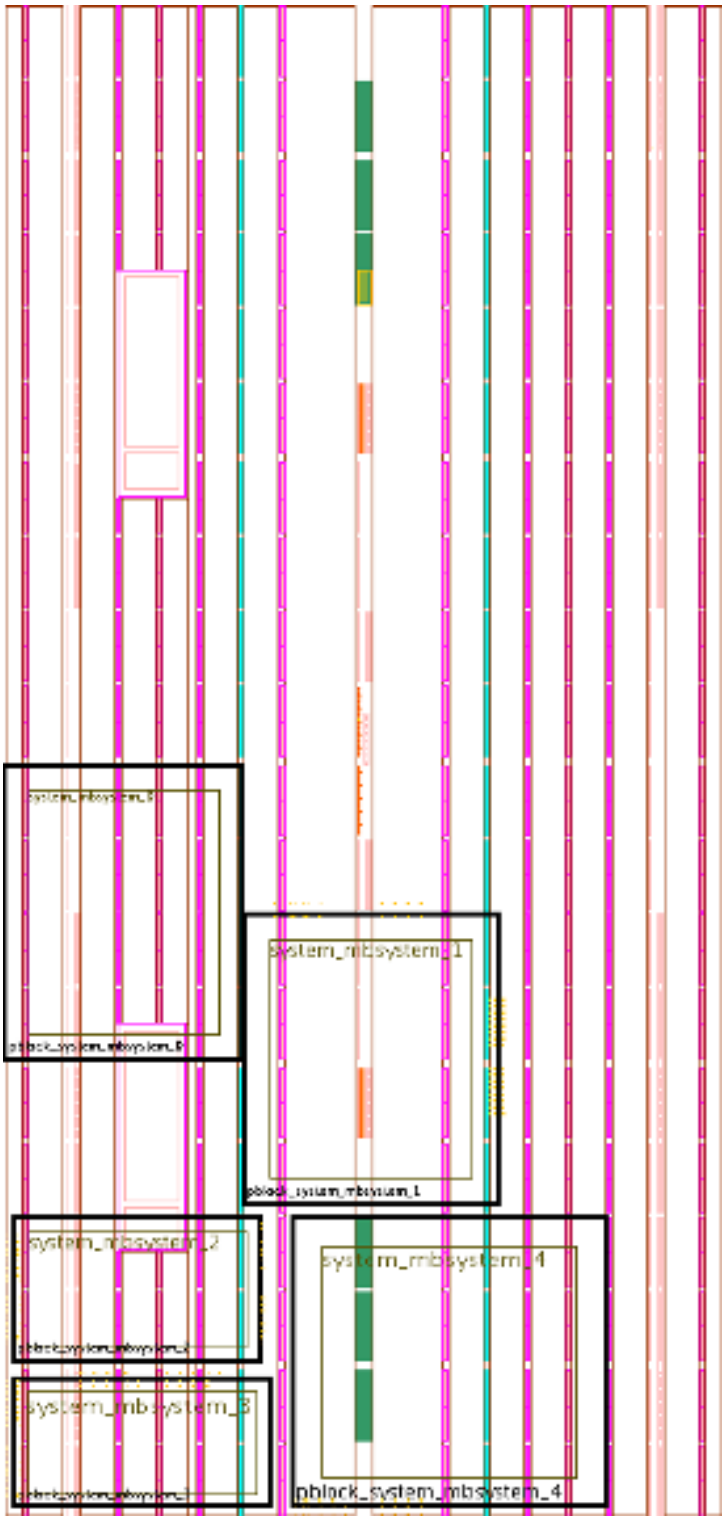

Figure 5.3: Design – `MB 5`
(*Screenshot captured from PlanAhead tool*)

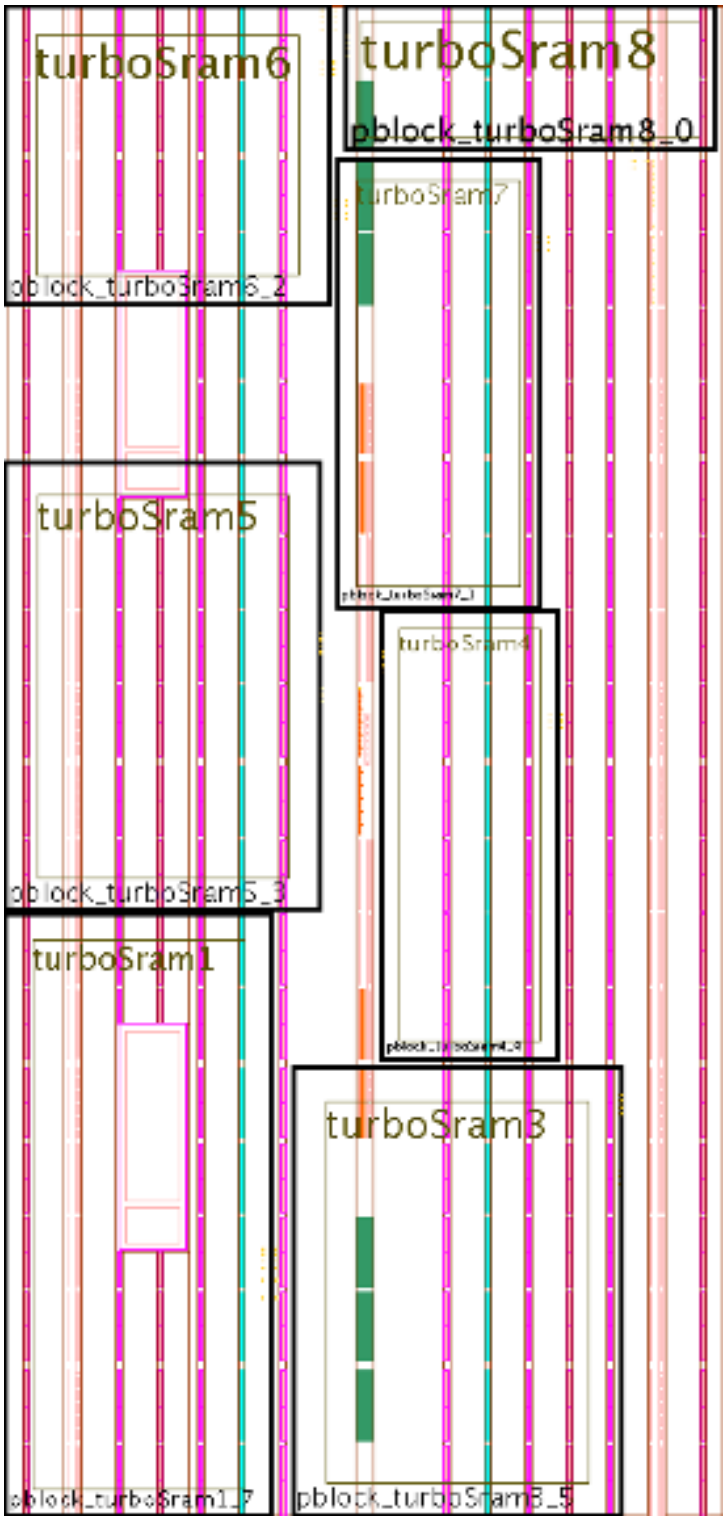

Figure 5.4: Design – `Viterbi 7`
(*Screenshot captured from PlanAhead tool*)

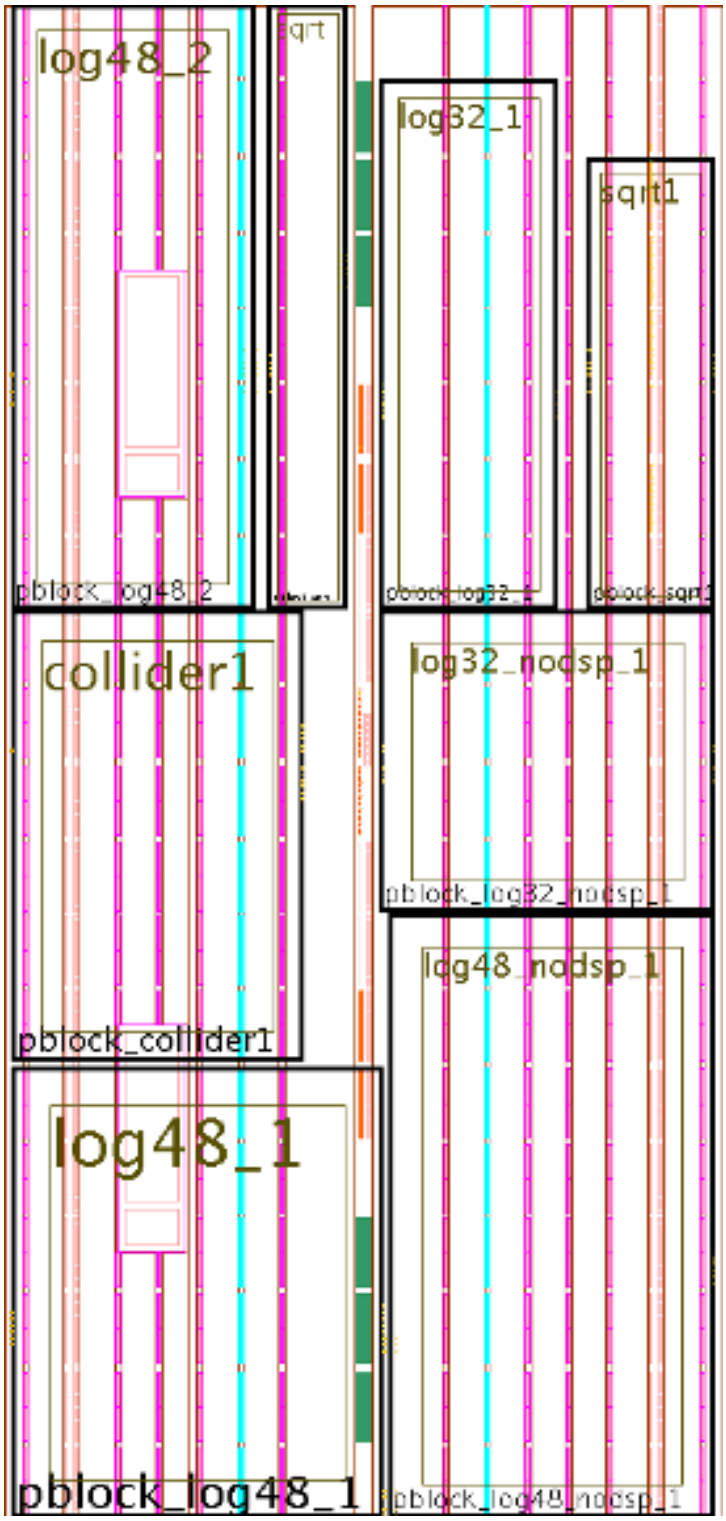

Figure 5.5: Design – `FloPoCo 8`
(*Screenshot captured from PlanAhead tool*)

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

5.2 Tool Performance

5.2.1 Speculative Flow

New designs go through a speculative mode of the incremental floorplanner after they are automatically floorplanned. The speculative floorplanner generates several feasible floorplans that meet resource requirements but may not meet timing. These floorplans are generated through the white space occupation and neighbor displacement algorithms of the incremental floorplanner. Since there is no required resource constraint for speculative changes, each module is allotted up to 100% more resources to generate modified layouts.

Speculative mode performance depends heavily on two factors: (i) the number of modules, and (ii) chip area. Speculative algorithms are executed on up to two modules at a time. Thus, designs with larger number of modules will generate more floorplans after their speculative run, hence increasing the tool run-time. On the other hand, designs that occupy larger chip area will have very few possible layout changes, since free resources will be limited. Moreover, most modules will be able to free up few resource rows or columns, leading to smaller run-times for the speculative flow.

Table 5.1 records the PATIS tool run-time during speculation. Run-time increases from `CFFT 3` up to `Viterbi 7`, where the run-time increasing effect of the number of modules dominates the negative run-time effect of chip area. Run-time then decreases for the two FloPoCo designs, due to their larger chip utilization, tight constraints, and very few unoccupied resources.

5.2.2 Incremental Flow

Layouts of modified designs are updated by the incremental floorplanner through a two-pronged strategy. The incremental floorplanner first picks floorplans from the database meeting the new resource requirements. These floorplans are verified for timing and if successful, implemented by the tools. If no suitable floorplan is obtained, the incremental

Table 5.1: PATIS speculative floorplanning time in CPU seconds

Design	**Number of floorplans generated**	**Run-time (CPU seconds)**
CFFT 3	27	7.05
MB 5	75	11.91
CFFT 6	108	18.64
Viterbi 7	147	35.98
FloPoCo 8	192	18.08
FloPoCo 10	300	21.46

floorplanner applies its algorithms to update the design floorplan.

The performance of PATIS during incremental floorplanning is shown in Table 5.2. The column under **Incremental I** is indicative of a selective search through the database for a floorplan that roughly meets the new requirements. The time taken for the search depends on the number of matches that the database returns. More matches lead to a greater run-time to validate all resource counts.

When the database returns no suitable match, PATIS attempts to re-floorplan the design. The time taken for this run, shown under **Incremental II** of Table 5.2, depends on: (i) the number of modules to be modified, and (ii) difficulty of allocating resources to a module. PATIS takes longer to re-floorplan two changed modules in the CFFT designs, as compared to a single-module change in the Viterbi decoder. The `FloPoCo 10` design takes more time for a single-module change due to the tight constraints of the module, making it hard to modify the floorplan.

5.3 Design Performance

In order to evaluate PATIS run-time improvement, the benchmarks were implemented using three flow variants: a manually floorplanned implementation using the Xilinx modular flow [10], a PATIS implementation, and an incremental PATIS implementation. The place-

Table 5.2: PATIS incremental floorplanning time in CPU seconds

Design	**Incremental I (CPU seconds)**	**Incremental II (CPU seconds)**
CFFT 3	0.27	0.45
MB 5	0.34	0.30
CFFT 6	0.36	0.41
Viterbi 7	0.76	0.26
FloPoCo 8	0.98	0.23
FloPoCo 10	1.45	0.63

Legend: Incremental I – time to search the database for a suitable floorplan; Incremental II – time to refloorplan a design if the search fails.

and-route times for the three implementations were obtained using the same clock frequency constraints, and the speedups determined.

Table 5.3 lists the place-and-route times for each benchmark design. The frequency listed is the minimum at which the design is able to meet timing with manual floorplanning. Implementation acceleration comes from the parallel reduction of a large global optimization problem to a set of smaller independent problems. 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 5.4. This table shows a substantial frequency improvement in two designs, while the remaining designs produced comparable minimum frequencies of operation for both the PATIS and the manual flows. In the latter benchmark designs, unregistered bus macros do not degrade the system clock frequency since critical nets are contained within modules.

Incremental floorplanning re-implements only the changed modules, leading to the shorter place-and-route times shown under **PATIS incremental** in Table 5.3, for the number of modified modules given in parentheses. However, in cases where static logic is changed, for instance, bus macro placement is changed between consecutive floorplans, complete re-implementation of the design will be required because of routing modifications. The place-

Table 5.3: 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 5.4: 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

and-route times for this is shown in Table 5.3 under **PATIS initial**. Even with the added overhead of complete design re-implementation, the times taken for PAR are far less than in the modular flooplan methodology. The PATIS speedups for both the complete and partial design re-implementations are shown in Figure 5.7. The substantial speedups observed for both implementations show that even drastic changes to the design resulting in a complete re-implementation will not lead to an excessive tool run-time.

5.4 Conclusions and Observations

The results described in this chapter demonstrate a clear advantage to using the DMD flow. The parallel implementation of modules in the six benchmark designs showed an average speedup of 10× for a complete design re-implementation and 35× for a partial re-implementation, compared to the manual non-PR modular flow. The floorplans for these

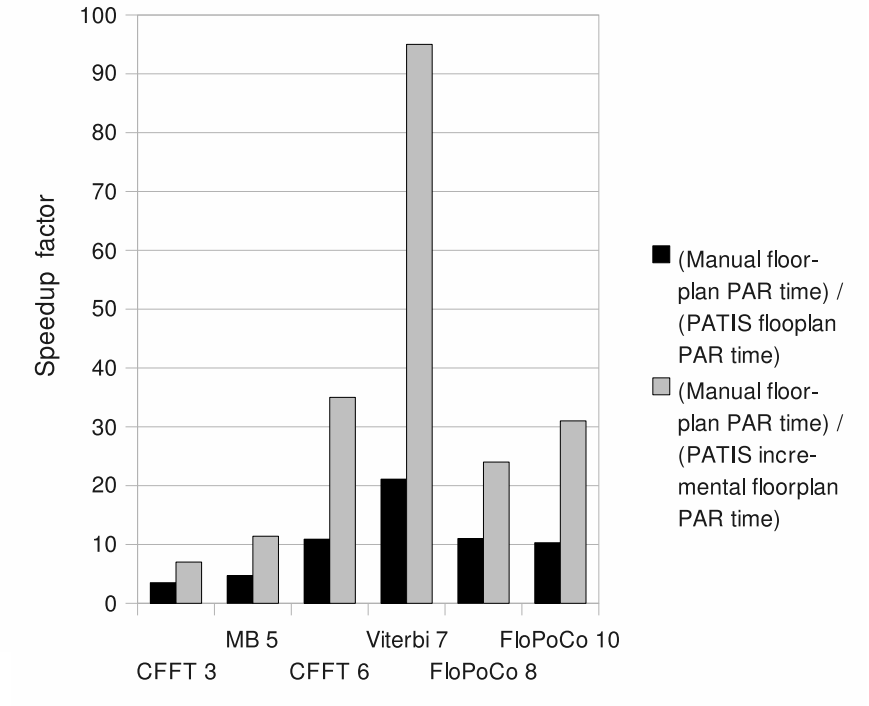

Figure 5.7: PATIS run-time improvements

implementations were generated either speculatively after automatic floorplanning, or incrementally after a design change trigger. Equally important, engineers are not tasked with manual re-floorplanning after a significant design change.

Chapter 6

Conclusions and Future Work

This thesis has defined a comprehensive approach to FPGA development acceleration to meet the demands of increasingly complex applications. Large-scale designs usually result in long run-times for implementation tools without any assurance of meeting the design's timing. This leads to several iterations producing noticeable delays. After the design has achieved timing closure, changes to the design functionality can be hard to incorporate. Updating a module can cause some parts of the design to fail timing even if the design met timing earlier, leading to repeated runs.

We propose the Dynamic Modular Design methodology to rapidly implement and incrementally update large designs without introducing drastic changes to the layout. DMD focuses on the early and middle stages of FPGA development and extends Xilinx's modular and PR flows to speed up implementation tool run-times. A design is partitioned into several self-contained units that can be implemented in parallel. These units or modules are located within partially reconfigurable sandboxes, and communicate with other modules and static logic through automatically placed bus macros. Module modifications are usually local and do not affect the whole design, unless the static logic changes. DMD also considers ripple effects by verifying the resource allocation of every module. A partial module-producing, automatic, timing-aware, incremental and speculative floorplanner was developed to assign regions to modules prior to implementation.

Chapter 2 introduced several FPGA design flows and tools developed to incrementally modify designs. Floorplanning is typically a manual flow but PATIS automates design floorplanning by capturing the designer's intent and treating modules differently according to the attributes described in Section 3.2. Chapter 4 described the main contribution of this thesis — the PATIS incremental floorplanner methodology and algorithms. In the basic and enhanced FPGA flows, designs are implemented sequentially, sometimes taking hours or days before complete implementation. In contrast, PATIS compiles all modules in parallel, thus exploiting multi-core computing by assigning implementation jobs to several processor cores. The PATIS debug module passively monitors module interfaces as opposed to the active probing done in the standard flows. A comparison of the PATIS flow with the Xilinx basic and enhanced flows is summarized in Table 6.1.

Table 6.1: Comparison of basic/enhanced and PATIS flows

Basic/Enhanced Flow	**PATIS Flow**
Manual floorplanning	*Timing-aware automatic* floorplanning that discriminates modules based on fickleness and viscosity
Incremental implementation	*Incremental* implementation with *speculative* floorplanning that explores the design space and achieves faster system adaptation to module modifications
Sequential compilations and bitstream generation of modules	*Concurrent* compilations and bitstream generation of modules by implementing them as partial bitstreams
Compilations mostly use a *single core* only	Compilations exploit *multi-cores*
Debug uses a specialized core that *actively* introduces changes to the design floorplan each time monitored modules change	Debug uses a processor that *passively* monitors module interfaces through configuration readback

This thesis focused on a subset of the DMD flow — the incremental floorplanner. New designs go through a speculative flow that makes placement updates to the initial floorplan, thus generating several alternatives stored in a database. This ready availability of floor-

plans speeds up incremental floorplanning of large-scale designs. When modules change, the speculatively-generated floorplans are verified for new resource requirements before further incremental floorplanning. The selected floorplan can then be implemented saving significant time. Chapter 5 shows a 35× speedup over the static flow when only modified modules are implemented. If static changes result in a full implementation, the speedup is 10×.

DMD works best on multi-core and multiprocessor systems since module implementation consumes significant time and memory resources on single-core systems. The DMD flow can be applied to both static and dynamic designs. However, implementing a static design with PR requires careful decomposition of the design to ensure that the modules meet timing individually.

6.1 Future Work

Although the results in this thesis show that the PATIS incremental floorplanner is able to significantly improve implementation time, there exist multiple avenues for expanding the breadth and depth of this work, including:

- **Portability**: Currently, PATIS supports Xilinx Virtex-4 devices. The algorithms require some knowledge of these FPGA devices, especially the configuration frame physical boundaries. However, the DMD flow and the PATIS algorithms are applicable to any FPGA device and static design that can be implemented with PR. By extending the algorithms to other device families, the application area can be broadened since some applications are best suited to certain families of FPGAs.

- **Background speculation**: Speculation is currently a sequential step in the DMD flow. Once a floorplan is automatically generated, the incremental floorplanner anticipates changes producing several feasible variants. After speculation has exhausted all reasonable options for a floorplan, design implementation follows. Running design speculation as a background job when an initial floorplan is available will allow the

design to be implemented in parallel.

- **Removal of duplicates**: Applying different combinations of speculation algorithms on the design modules generates several floorplans which are indexed in a database. Some of these floorplans may have identical placements for all modules. During incremental floorplanning, matching layouts are selected from the database for further analysis. Ensuring that no duplicates exist among these variants speeds up and checks for resource satisfaction and timing verification.

Bibliography

[1] J. Zhu, I. Sander, and A. Jantsch, "Pareto efficient design for reconfigurable streaming applications on CPU/FPGAs," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 8-12 2010, pp. 1035 –1040.

[2] C. Farabet, C. Poulet, and Y. LeCun, "An FPGA-based stream processor for embedded real-time vision with convolutional networks," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, sept. 2009, pp. 878 –885.

[3] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAs," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 910–921, 2009.

[4] V. K. Prasanna and A. Dandalis, "FPGA-based cryptography for internet security."

[5] T. Wollinger, J. Guajardo, and C. Paar, "Cryptography on FPGAs: State of the art implementations and attacks," 1999.

[6] V. Subramanian, J. G. Tront, C. W. Bostian, and S. F. Midkiff, "A configurable architecture for high-speed communication systems."

[7] C. J. Comis, "A high-speed inter-process communication architecture for FPGA-based hardware acceleration of molecular dynamics," Tech. Rep., 2005.

[8] GNU Make. [Online]. Available: http://www.gnu.org/software/make/

[9] T. Frangieh, A. Chandrasekharan, S. Rajagopalan, Y. Iskander, S. Craven, and C. Patterson, "PATIS: using partial configuration to improve static FPGA design productivity," in *17th Reconfigurable Architectures Workshop (RAW 2010), Atlanta, GA*, 2010.

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

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

[12] Guided PAR. [Online]. Available: http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0088_14.html

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

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

[15] Quartus II incremental compilation for hierarchical and team-based design. [Online]. Available: http://www.altera.com/literature/hb/qts/qts_qii51015.pdf

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

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

[18] J. Crenshaw, M. Sarrafzadeh, P. Banerjee, and P. Prabhakaran, "An incremental floorplanner," in *GLS '99: Proceedings of the Ninth Great Lakes Symposium on VLSI*. Washington, DC, USA: IEEE Computer Society, 1999, p. 248.

[19] S. Liao, M. A. Lopez, and D. Mehta, "Constrained polygon transformations for incremental floorplanning," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 3, pp. 322–342, 2001.

[20] Y. Liu, M. Yuchun, H. Xianlong, D. Sheqin, and Z. Qiang, "An incremental algorithm for non-slicing floorplan based on corner block list representation," in *Chinese Journal of Semiconductors*, 2005, pp. 2335–2343.

[21] H. Menager, M. Basel, and R. Kadiyala, "Dynamic floorplanning: A practical method using relative dependencies for incremental floorplanning," *International Workshop on IP-Based SoC Design*, pp. 124–130, 2003.

[22] Y. Liu, H. Yang, R. Luo, and H. Wang, "A hierarchical approach for incremental floorplan based on genetic algorithms," in *ICNC 2005, LNCS 3612*, 2005, pp. 219–224.

[23] 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.

[24] Xilinx floorplanner. [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/design_entry/floorplanner.htm

[25] PlanAhead user guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf

[26] Partial reconfiguration design with PlanAhead. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/PR_User_Guide.pdf

[27] About BNF notation. [Online]. Available: http://www.cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html

[28] 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.

[29] Placement constraints. [Online]. Available: http://www.xilinx.com/itp/xilinx4/data/docs/cgd/types3.html

[30] J. Becker, R. Woods, P. Athanas, and F. Morgan, *Reconfigurable Computing: Architectures, Tools, and Applications*. Springer, 2009.

[31] Early access partial reconfiguration user guide. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/ug208_92.pdf

[32] SQLite. [Online]. Available: http://www.sqlite.org/

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

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

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

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

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

Nomenclature

ASIC Application-Specific Integrated Circuit, page 2

BIT Bitstream file, page 10

BNF Backus-Naur Form, page 40

BRAM Block Random Access Memory, page 7

CBL Corner Block List, page 15

CFFT Complex Fast Fourier Transform, page 67

CLB Configurable Logic Block, page 7

DCM Digital Clock Management, page 8

DFT Discrete Fourier Transform, page 67

DMD Dynamic Modular Design, page 3

DSP Digital Signal Processor, page 7

EDA Electronic Design Automation, page 10

EDIF Electronic Data Interchange Format, page 9

EDK Embedded Development Kit, page 67

FFT Fast Fourier Transform, page 33

FPGA Field-Programmable Gate Array, page 1

FSL Fast Simplex Link, page 33

GUI Graphical User Interface, page 30

HDL Hardware Description Language, page 4

HLV High-Level Validation, page 28

ICAP Internal Configuration Access Port, page 29

IDE Integrated Development Environment, page 2

IP Intellectual Property, page 1

IRL Irreducible Realization List, page 26

ISE Integrated Software Environment, page 8

JTAG Joint Test Action Group, page 29

LLD Low-Level Debugging, page 28

NCD Native Circuit Description, page 9

NGC Native Generic Circuit, page 9

NGD Native Generic Database, page 9

PAR Place-And-Route, page 2

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

PR Partial Reconfiguration, page 3

RPM Relationally Placed Macro, page 33

RTL Register Transfer Logic, page 8

RTR Run-Time Reconfiguration, page 1

TCL Tool Command Language, page 42

UCF User Constraints File, page 30

XDL Xilinx Design Language, page 43

XML Extensible Markup Language, page 43

XST Xilinx Synthesis Technology, page 24