Facilitating Floorplan-Based Design for Heterogeneous FPGAs

by

Sarah Khalid

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Facilitating Floorplan-Based Design for Heterogeneous FPGAs

Sarah Khalid
Master of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto
2022

Abstract

Field Programmable Gate Arrays (FPGAs) are used for many applications due to their flexibility and reconfigurable nature. However, traditional FPGA implementation flows have been unable to keep pace with the increasing size and complexity of FPGA designs, adversely impacting designer productivity. As a result, some techniques have been used to make FPGA implementation flows more modular. These techniques often require the use of floorplan constraints.

The goal of this thesis was to add the ability to implement FPGA designs with floorplan constraints to VPR, an open-source FPGA CAD tool. We present the algorithms that were implemented to allow VPR to run with floorplan constraints. We evaluate VPR flows with a variety of floorplan constraint scenarios across 3 FPGA architectures and 32 benchmark circuits. We found that VTR was able to adhere to a variety of floorplan constraints while maintaining comparable quality of results.

Acknowledgements

I would like to express my deep gratitude to my supervisor, Vaughn Betz, for his guidance, help, and patience over the course of my degree. His advice has helped me grow immensely as a researcher and as an engineer.

I would also like to thank my committee members Jason Anderson and Paul Chow for their valuable suggestions, which helped to improve this thesis. I am also grateful to my lab members for their friendliness and willingness to help during these past two years.

My sisters have always been supportive and encouraging, and for that I thank them. Their children, my adorable niece and nephews, have never failed to make me smile during the past two years. I would like to thank my husband, Bashar, for his humour and support that always make things easier for me. Lastly, I would like to thank my parents, whose endless love, support, and encouragement have always been the foundation of the milestones I have reached in my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Field-Programmable Gate Arrays (FPGAs) are computational devices that are used to implement digital hardware circuits across many application domains. Over time, FPGA architectures have moved from homogeneous designs to increasingly heterogeneous, System-on-Chip-like entities, on which complex circuits can be implemented [8]. This increasing capability has developed as FPGAs have been adapted for use across different fields such as wireless communications [29] and machine learning [15]. FPGAs offer key benefits over other computational platforms - designs can be changed by simply reprogramming the FPGA. Due to this re-configurable nature, they have shorter design times and lower non-recurring engineering costs compared to ASIC and full-custom designs. These advantages are partially offset by the increased area and power consumption of FPGAs compared to ASICs [9]. Nonetheless, there are still numerous applications where the benefits of FPGAs outweigh the costs. By providing different hard blocks that are used for common logical operations, FPGAs have managed to remain flexible while also providing area savings to mitigate their disadvantages compared to ASICs [9]. As a result, FPGAs remain a widely-used computational platform, as demonstrated by their increasing use to manage data center operations [57].

The heterogeneous resources of FPGAs, as well as their re-configurable nature results in significant advantages over ASICs with respect to time-to-market and flexibility. However, the current long compilation times of FPGA CAD flows may dilute this advantage as they hinder designer productivity. Designers have to contend with long compilation times, and often have to run compilation multiple times[52]. To take advantage of the resources of FPGAs, while maintaining the flexibility and relative ease of design that made them so widely used, FPGA CAD tools must be able to produce implementations that effectively utilize the heterogeneous architectures within reasonable compile time limits. Thus, there is a pressing need for design flows that maintain productivity and reasonable development times as FPGAs become ever larger and more complex.

FPGA CAD flows have typically used flat compilations, meaning the entirety of the design is optimized at once. Flat compilation has some advantages - there is no set up required in terms of dividing the design into sub-parts, and optimization can be done across boundaries. However, the flat compilations have also led to long compile times and little control over the finer details of circuit implementations. To address long compile times, FPGA CAD tools can instead use a modular approach, where different modules of a design can be compiled separately before being placed together on the chip. ASIC design has long since used this modular approach. FPGA CAD tools have been slower to adopt this approach because of the complications that arise in creating and constraining the placement of modules on a chip with heterogeneous resources.

To facilitate the modular design approach, FPGA designers must be able to specify placement constraints for different parts of their design. This means that they will constrain groups of primitive blocks to certain regions on the chip. Modular design allows for team-based design where teams can work on different modules of a design independently before placing them together on a chip. A designer may want to polish certain portions of a design without changing other parts - a process known as partial reconfiguration. Partial reconfiguration is a popular design technique for many reasons - it enables hardware sharing and increased resource utilization [30]. Modular design has been effective for dealing with design and team scalability issues for ASICs. It can also be a powerful technique for FPGA compilations - the caveat is that FPGA CAD flows must be able to adhere to placement constraints for this technique to be used.

To facilitate these modular, divide-and-conquer design flows, FPGA designers must be able to floorplan. Floorplanning is the process by which designers group netlist primitives into partitions, and constrain these partitions to designated regions on the chip. Studies have been done to identify how to create effective FPGA floorplans [45], [39]. However, to be able to test and implement floorplans, CAD tools must be able to adhere to placement constraints during their physical implementation stages.

Verilog-to-Routing (VTR) is an open source CAD tool which is used for FPGA architecture and CAD flow research [51]. It is widely used in academic FPGA research due to its ability to model a variety of FPGA architectures and efficiently implement circuit designs. Enhancing VTR to adhere to placement constraints would be a useful improvement to a popular open-source FPGA CAD flow. Commercial FPGA tools can respect placement constraints, but their algorithms have not been published. Additionally, theses closed-source tools only target a vendor's specific architectures. We seek to enable the use of placement constraints in an open-source framework that can be used with many FPGA architectures.

1.2 Research Goals

The aim of this thesis is to develop specifications and algorithms for arbitrary placement constraints to be respected by the physical CAD flow for a variety of FPGA architectures. More specifically, we add these enhancements to the open-source FPGA CAD tool VTR. In order to accomplish this, we:

1. Facilitate the description of placement constraints by users of the CAD tool
2. Enhance the algorithms of the packing and placement stages, to ensure that the placement constraints are respected.

1.3 Thesis Overview

This thesis is organized as follows. The background and motivation for floorplanning and placement constraints in FPGA design will be explored in Chapter 2. Chapter 3 presents an expressive floorplan constraints language that was added to VPR. Chapter 4 explores the algorithmic changes that were made to the packing and placement stages of VPR to ensure that floorplan constraints are respected. Chapter 5 presents the results that show that VTR can adhere to floorplan constraints across different benchmark circuits and architectures. Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Background

In this chapter, we discuss the background relevant to understanding FPGAs, as well as their CAD flows and design methodologies. We also explore techniques that have been used to improve the FPGA design process. First, we discuss the architectural features and CAD design flows of modern FPGAs. We reflect on how the evolution of FPGAs has impacted traditional design methodologies, and the trends that are emerging to balance between flexibility and efficiency in FPGA design. We introduce floorplanning, its prevalent use in ASIC design, and how the technique can provide benefits for FPGA design. Finally, we examine the current commercial tools that are available for FPGA floorplanning and give an overview of VTR - the open-source CAD tool for which we will enable floorplanning.

2.1 FPGAs

Field-Programmable Gate Arrays (FPGAs) have been adopted for use across many application domains over the last few decades. Modern FPGAs are important digital hardware platforms in the fields of wireless communications [29], machine learning [15], data center operations [57], ASIC prototyping [34], and more. As FPGA architectures develop to become larger and more heterogeneous, it is likely that FPGAs will continue to be relevant across many fields. Accordingly, their design methodologies must evolve to facilitate the increasing complexities of the designs that they implement.

2.1.1 FPGA Architecture

FPGA architectures contain two main elements - computational logic blocks, and programmable routing to connect the blocks. The programmable routing is made up of wires, connection blocks, and switch blocks that form the connection between logic blocks [7], [58].

The blocks in an FPGA can be further categorized as part of its soft logic, or part of its hard logic. Much of the soft logic in an FPGA is made up of primitives such as look-up tables (LUTs) and flip-flops (FFs). The LUTs and FFs are most often paired together to form basic logic elements (BLEs) [7]. These BLEs are then grouped together into blocks called logic blocks to improve area and performance [4]. Figure 2.1 shows an example of a typical BLE.

Typically, modern FPGA architectures use fracturable LUTs. Fracturable LUTs offer the benefit of flexibility - the ability to use two smaller LUTs reduces the occurrence of unused LUT inputs, leading to greater area efficiency. The ability to use a larger LUT where necessary leads to better performance. A fracturable LUT can either be used as a K-input LUT or two LUTs with size up to K-1 [37]. With a sufficient amount of basic logic elements, homogeneous FPGAs can implement nearly any digital circuit.

As FPGAs developed for use across different fields, specialized hard blocks have been included to make the most common operations more efficient. For example, block RAMs became an important part of FPGA architectures when the need for on-chip memory became apparent. Furthermore, digital signal processing blocks (DSP) blocks became a common hard block in FPGA architectures due to their use in communications and signal processing - two domains which rely heavily on FPGAs and in which multiplication operations are common. Thus logic blocks, BRAMs, and DSPs are common logic blocks that are seen in many heterogeneous FPGA architectures. Figure 2.2 shows an example of a typical heterogeneous FPGA layout.

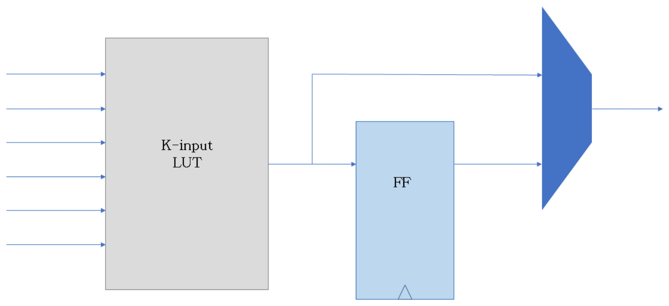


Figure 2.1: A typical Basic Logic Element

Including hard blocks on FPGAs always involves a tradeoff between flexibility and efficiency. However, the efficiency gained by including useful hard blocks is such that FPGA architectures are trending towards including more specialized hard blocks. For this reason, FPGAs are becoming increasingly heterogeneous, system-on-chip-like devices. As FPGAs are adopted across more application domains, it is likely that FPGA architectures will continue to evolve and become more complex to address the wider range of needs [8], [70]. The trends in FPGA architecture will continue to have an impact on the traditional design flows of FPGAs.

The heterogeneous nature of FPGA architectures will have a significant impact on floorplanning for FPGAs, as we discuss further in later sections.

2.1.2 FPGA CAD Flow

The FPGA CAD flow takes a designer's high-level circuit description and transforms it over many stages to produce a bitstream that can be used to program the FPGA to implement the desired circuit. Although there may be variations in the exact number of stages in any given CAD flow, and the cutoff points of each stage, the stages in FPGA CAD flows can largely be grouped into one

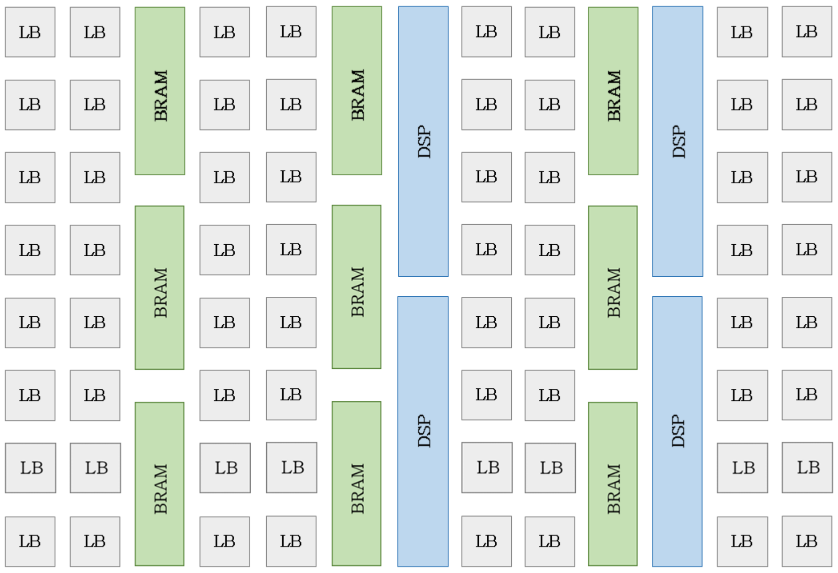


Figure 2.2: An example of a heterogeneous FPGA architecture

of three main stages: high-level synthesis, logic synthesis, and physical synthesis. We present an example FPGA CAD flow in Figure 2.3 and discuss its stages below [12].

High-Level Synthesis

The high-level synthesis (HLS) stage was introduced as a way to improve designer productivity, by providing a higher-level of abstraction in circuit descriptions. Traditionally, designers would provide a register transfer level (RTL) description of their circuit using a hardware description language (HDL) such as Verilog or VHDL. HLS is the process of using a conventional programming language, commonly C or OpenCL, to provide an algorithmic description of the circuit [53]. The conventional programming languages are typically augmented with additional pragmas to express hardware-friendly parallelism. HLS will then produce an RTL-level design based on the algorithmic description.

Logic synthesis

Next, the CAD flow performs logic synthesis. Common steps that are part of logic synthesis include elaboration, logic optimization, and technology mapping [17]. Elaboration is the step that transforms the digital circuit description provided by the designer into a set of boolean logic operations and signals [28]. Logic optimization can be performed at an RTL-level or a gate-level. One of its goals is to reduce the amount of redundant and/or duplicated logic. Finally, technology mapping creates the primitive netlist by implementing the optimized hardware with primitives found in the target FPGA architecture, such as LUTs, FFs, and multipliers [10].

Packing, Placement, and Routing

The remaining tasks necessary for generating the bitstream are packing, placement, and routing [51]. During packing/clustering, netlist primitives are grouped together onto the available blocks on the device - for example, these can be logic blocks, RAM blocks, or DSP blocks [40, 41, 63]. This step

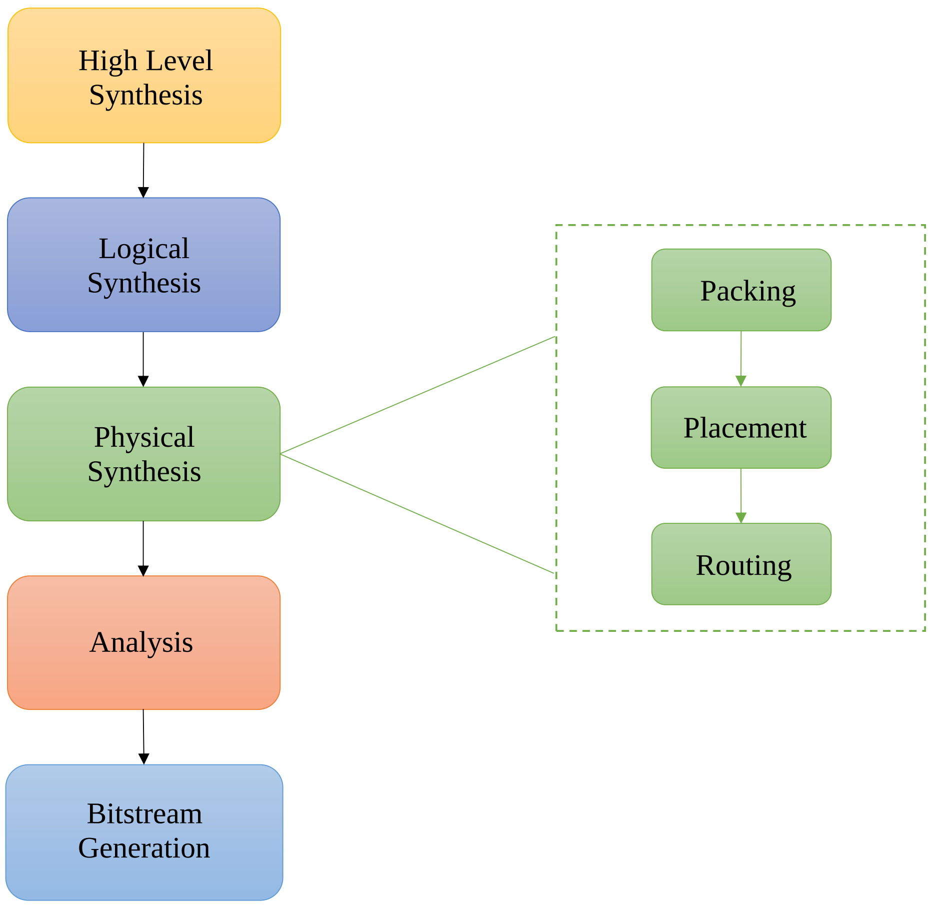


Figure 2.3: An FPGA CAD Flow

is generally done to reduce the number of blocks that must be placed during the next stage, placement, while respecting the often complex legality constraints on which primitives can be clustered into a given block. During placement, the CAD tool decides the locations of the clustered blocks on the target device [1, 43]. An important objective during this stage is to minimize wirelength, as this parameter will affect result quality in terms of critical path delay and power consumption. It is worth noting that there is not always a clear distinction between the packing and placement stages; the lines between them often blur together. For example, in the VTR CAD flow [51], packing is completed and then followed by placement, but in Intel's Quartus Prime CAD tool [20], the two stages are iterated. In some tools such as elfPlace [43], packing is generated after a flat placement is performed. Finally, routing determines what connections should be formed between the placed blocks, using the programmable interconnect on the target device [42, 47].

After these stages, a bitstream can be generated that will specify how to set all the switches on the target FPGA to achieve the desired design. Prior to bitstream generation, however, there is the additional step of analysis and verification. Timing analysis is done to ensure that timing constraints are met [46]. At this stage, if the design has not met timing constraints, the designer may have to change the design and run it through the CAD flow again, which can be a time-consuming process. It is becoming increasingly difficult as FPGA capacity now allows for ever-larger circuits to be implemented.

An FPGA CAD flow is presented in Figure 2.3. Speeding up this compilation stage is a key motivation for floorplanning for FPGAs [56, 76]. To this end, CAD tools must have two abilities to make floorplanning a viable part of the FPGA CAD flow.

1. CAD tools should be able to generate valid and optimized floorplans for FPGA designs.
2. The physical synthesis stage of the CAD flow, highlighted in Figure 2.3, must adhere to the given floorplan constraints.

The contributions of this thesis focus on adding the latter ability to the open-source CAD tool, VTR [51].

2.2 Techniques for Improving FPGA CAD Flows

2.2.1 The Case for Upgrading the Traditional FPGA Design Cycle

As predicted by Moore's law, FPGA logic capacity doubled every year during the 1990s, leading to larger FPGAs than ever before [70]. This rapid expansion led to the development of automated CAD design flows for FPGAs as manual design flows were no longer feasible [70]. Just as the rapid expansion of FPGA sizes led to the development of automated design flows during the 1990s, the current scale and heterogeneity of FPGAs will require CAD flows to adapt to improve designer productivity.

Many of today's FPGAs resemble System-on-Chip like entities, with FPGA designs resembling large systems that are made up of relatively independent sub-parts. Moreover, there is a push for hard Networks-on-Chips (NoCs) to be included as part of FPGA architectures due to the benefits they provide as interconnect solutions [8, 24, 68, 69, 79]. IP cores have also become an important part of FPGA architectures, as they can be re-used in the design to implement complex functions [31].

While the above trends have been developing, however, most FPGA CAD flows have continued to use traditional flat compilation techniques. Flat compilation is when an entire design is synthesized, placed, and routed across the whole chip. Although flat compilation has benefits (separating large designs into separately-compiled modules may result in missed cross-boundary optimizations), it has drawbacks for today's FPGA designs for a few reasons. Flat compilation can take hours or days for large circuits [27], [48]. FPGA compile times negatively impact designer productivity, as well as the time to market of FPGA designs. This problem is further exacerbated by growing device capacities, which put an upward pressure on the compile time and the number of design iterations that may be required to achieve timing closure.

Timing verification is the penultimate step in a typical FPGA CAD flow. Timing constraints must be met before a bitstream implementation can be generated for an FPGA. These constraints ensure that the circuits function correctly and at the required speed - the bounds on the critical path delay must be satisfied. To satisfy setup constraints, signals must arrive at registers before the clock edge. To satisfy hold constraints, register signals must remain stable for a given amount of time after the clock edge.

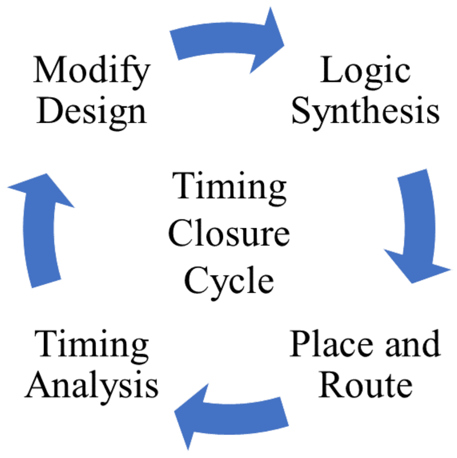


Figure 2.4: The FPGA timing cycle. This cycle repeats until timing closure is achieved.

The timing verification cycle is illustrated in Figure 2.4. The HDL design is put through the automated CAD flow to synthesize, place, and route the design on the FPGA. Then, timing analysis is performed to see whether timing constraints were met. If the constraints were not met, the designer must manually modify the design, and start the process again. This flow severely decreases designer productivity. The design iterations that must be done to achieve timing closure limit the scope of design exploration that can be done by even expert hardware designers. Another contributing factor that makes timing closure difficult is the non-deterministic nature of FPGA CAD tools. Placement and routing algorithms are often guided by heuristics, and so their results can be variable between runs. This means that when a designer sees that their design fails, fixes the failing portion of their design, and runs the CAD flow again, they may still encounter timing issues due to the changed solutions now produced by the CAD tool. Therefore, to truly address the difficulties in the timing closure cycle, the designer must be given more control over placement, in addition to having shorter run times.

A number of strategies have been studied to offset the issues outlined above. In particular, incremental placement and partial reconfiguration flows have shown promise in reducing compile times. In particular, the use of partial reconfiguration allows for modular CAD flows which break down FPGA designs into smaller sub-parts to be implemented and also have the added benefit of facilitating team-based design. As FPGA designs grow more complex, it is advantageous to have different members be able to perfect parts of a design, and them combine them without degrading quality. To be able to implement these flows, however, CAD tools must be able to respect placement constraints and floorplans at different levels of abstraction, which many non-commercial CAD tools, including VTR, are currently unable to do.

We now discuss different methods that have been used to address issues with long compile times and difficulty achieving timing closure.

2.2.2 Incremental Placement

Incremental synthesis is a technique where only the parts of an implementation that need to be changed for a given optimization are re-synthesized, while the remaining parts of the design remain the same. This means that an initial compilation would be done, and timing information would be preserved from that compilation. Then, the parts of the place-and-route implementation that are known to meet timing in that first compilation are preserved in subsequent compilations, while the parts that do not meet timing are re-synthesized [13, 64, 74]. Timing information is preserved after each compilation. Thus, in an iterative way, timing closure can be achieved with fewer cycles due to the incremental, minute way in which optimization changes are made [13]. Another advantage of incremental synthesis is that optimizations can be performed while avoiding full recompile iterations, which can reduce compile times.

In [64], Singh and Brown present an algorithm for incremental placement that allows optimized placement locations to be selected based on information about the delays that will exist after placement and routing. By tightly coupling retiming - a technique for optimizing delays in a synchronous circuit - with incremental placement, they were able to improve result quality. Specifically, the authors found that their incremental placement algorithm achieved a 22% increase in operating frequency, a significant increase compared to performing retiming with a completely new placement, which achieved an increase of only 6.9%. A point to note is that the incremental placement engine requires the list of new logic elements with preferred locations for each of the new elements with each incremental recompile.

In [13], Chen and Singh present an incremental re-synthesis tool to aid timing closure and reduce CAD compile times. They note that designers often only have to change a small portion of their design to correct a bug before compiling again. Therefore, compilation time can be reduced by keeping the synthesized results of all unchanged parts of the design. They enable this preservation by making every part of the CAD flow incremental, including logic synthesis. Preserving parts of the logic synthesis solution helps preserve more of the place and route results from the previous compile. This is because changes in technology mapping would affect which primitives are available to be packed and placed later on in the flow, meaning that if logic synthesis was not preserved, some of the placement results could not be used in an incremental recompile, even if they were preserved. Their results show that in most cases, placement and routing preservation exceeds 99% between compiles. With the method of incremental re-synthesis for the whole CAD flow, they were able to preserve good timing solutions 82% of the time, and reduce synthesis runtime by 6.5x for common HDL changes.

Lin et al. [38] present a three-stage timing aware incremental placement algorithm. The results show that this approach produces placements that use 16.7% less time to route than placements produced by Xilinx Vivado. The timing-driven nature of the algorithm is particularly useful in helping to achieve timing closure. This paper demonstrates the importance of the placement stage in ensuring good quality of results in CAD flow implementations.

Xiao and Liang have shown that incremental synthesis can be used for machine learning applications [74]. Their proposed incremental synthesis framework, Acoda, speeds up the design flow for DNN accelerators on FPGAs by 9.31x to 34.17x while maintaining good quality of results.

2.2.3 Re-use of Pre-Synthesized Modules

Another method of reducing CAD run times is to create designs by combining smaller, pre-synthesized modules, which are also called hard macros. By using a hard macro, a designer can reuse the exact placement and routing of a portion of the design. This method is more suited to designs which have a high reuse of modules, such as the machine learning designs in [35].

In [36], the authors present HMFlow, a CAD flow that utilizes pre-synthesized hard macros to speed up compilation time for rapid prototyping FPGA designs. Although this technique is found to result in quality degradation in circuit speed, this trade off was acceptable for the purposes of the rapid prototyping implementations it was used for. HMFlow achieved speedups of 10-50x over Xilinx CAD tools. Although this flow can no longer be used due to its reliance on Xilinx Design Language (XDL), a design language that is no longer supported, it still demonstrates the idea that using pre-synthesized parts of a design in a CAD flow can significantly improve CAD run times.

In [35] the authors present a CAD flow called RapidWright - an open-source framework which builds off of Xilinx tools to create highly-tuned, custom implementations for FPGA designs that take full advantage of device resources. They use pre-synthesized Vivado implementations of small modules, since the Vivado tool can produce excellent implementations for small problems. They then replicate and relocate these modules for larger implementations. This is particularly useful for machine learning designs, which tend to include many repeated modules. Across five designs, the speedup of the compile time ranged from 12 - 97 times. Thus, RapidWright provides faster compilation by preserving the high quality placement of pre-implemented modules and stitching the blocks together.

The RapidWright framework was utilized by Guo et al. in [25], to help build RapidStream - a parallel implementation framework for compiling FPGA HLS designs. RapidStream takes in HLS programs in C/C++ and performs a full compilation flow which produces a fully placed and routed design. It partitions designs at the HLS level, and performs placement and routing for the partitions in parallel. It then uses RapidWright to stitch the modules together. The authors found that compilation time reduces by 5-7x with RapidStream, when compared to a commercial off-the-shelf framework.

2.2.4 Partial Reconfiguration

Partial reconfiguration is a design strategy where design implementations are produced in partial bitstreams that implement partial modules of the circuit [72]. Using this strategy, parts of the design can be reconfigured while other parts keep running. This design technique provides several benefits. Partial reconfiguration allows multiple modules to time-share the hardware resources of an FPGA, meaning that multiple applications can be implemented on a single FPGA [6]. It also allows portions of the design to keep running while other portions are being reconfigured, which leads to increased system performance [30]. Lastly, reconfiguration times are shortened since recompiling a portion of design takes less time than recompiling a whole design [30].

Park and Xiao demonstrate how partial reconfiguration can be used to reduce compile times in [56] and [76] respectively. Park breaks down large designs into small independent *leaves*. The leaves are then connected via a Packet-Switched Network-on-Chip (PSNoC). The leaves can be compiled in parallel, and only a partial bitstream needs to be generated if any unit needs to be reconfigured. Results demonstrate that this method reduced compile times on small benchmark circuits from 30 minutes to 7 minutes. Park notes, however, that modern FPGA flows are not tuned for this usage as flat compilation is still the norm.

In [76], Xiao et al. build off of the work of Park and introduce PRFlow. They demonstrate the results of their partial reconfiguration flow by implementing the Rosetta benchmarks [82] on a Xilinx FPGA and reduce compile times from hours to under 10 minutes. However, their results also demonstrate a quality degradation in the circuit speed, due to the limited bandwidth of the PSNoC. Xiao et al.. use the Vivado Out-of-Context (OoC) design flow to perform separate compilation of the leaves. They improve on their performance in [75] by using point-to-point wires rather than the PSNoC from the previous paper. The performance gain is 1.5-10x their previous work while gaining interface area overhead savings of 47-86% over their previous work.

As mentioned previously, partial reconfiguration has benefits beyond reducing compile times. Both [32] and [33] demonstrate the benefits of partial reconfiguration flows in image processing applications. In [32], Koch et al. present a smart camera system. Making such a system partially reconfigurable is advantageous because one system can have multiple uses. They implement different image processing algorithms as partially reconfigurable modules, thus allowing them to be dynamically used at run-time. In this way, their system is useful for a wide range of smart camera applications, including particle filtering, motion detection, skin colour detection, and more. In [33], the authors present a partial reconfiguration framework. They evaluate it by generating an implementation of a colour space conversion (CSC) IP core, and find that they are able generate an implementation that is optimized with respect to the area/speed ratio of the design.

2.2.5 Comments on New FPGA CAD Techniques

Overall, FPGA designs are large and complex systems that can often be compiled more efficiently when broken up into sub modules. Techniques such as incremental synthesis, the use of hard macros, and partial reconfiguration have been used to help speed up the CAD flow and achieve timing closure. The techniques mentioned above often overlap, with incremental synthesis and hard macros being used as tools to implement partial reconfiguration flows [6], [32].

A crucial point to note is that for CAD flows to be able accommodate these techniques, placement constraints and/or floorplanning are needed at various levels of abstraction during the CAD flow. To this end, the goal of this thesis is set up a framework to have placement constraints and floorplanning in VTR, an open-source FPGA CAD tool that is heavily used in academic research.

2.3 Floorplanning

Floorplanning is the process of creating logical partitions and constraining them to physical regions on a given target substrate. Floorplans can either be generated by CAD tools or manually provided by designers. In floorplanning, a logical partition refers to a set of netlist primitives, while a physical region is an enclosed area on the target device. Generally, each logical partition is assigned to one physical region on the chip. To create a floorplan for a design, logical partitions are typically assigned to physical regions in such a way that some objective function is optimized.

The purpose of floorplanning is to separate large designs into independent sub-modules early on in the design process. Floorplanning has traditionally been a technique common in ASIC CAD flows. ASIC designs are typically larger than FPGA designs, and have added complexities which require designers to have a greater level of control over them. FPGA CAD flows have generally tended towards flat compilation over whole designs due to their relatively smaller scale. Flat compilation has also been beneficial for FPGA CAD flows if the compile time is reasonable and the team working on a given design is small, since it requires less planning than modular flows. Therefore, floorplanning has not been as deeply studied with FPGAs as it has been for ASICs. As FPGA design productivity hits a road-block due to high compile times and difficulty achieving timing closure, however, floorplanning can be a key technique to overcome the productivity hurdles facing designers that use FPGAs [13].

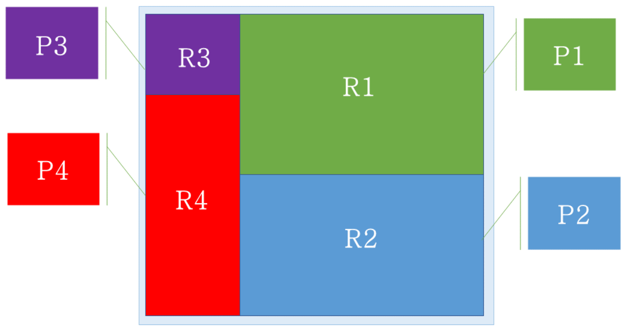


Figure 2.5: A floorplan contains physical regions on a target device, depicted in this figure as R1, R2, R3, and R4. It also contains logical partitions - groups of netlist primitives that are assigned to physical regions - depicted in this figure as P1, P2, P3, and P4

.

2.3.1 Problem Definition

A floorplan is a set of physical regions, created such that each region contains a set of resources greater than or equal to the resources required by its assigned netlist primitives [52].

Many floorplanning tools may have additional conditions to creating a valid floorplan. Some additional conditions may include:

1. *The cost of the floorplan is minimized according to some objective function.*
2. *There is no intersection between any two regions.*
3. *Each region is contained within a fixed outline.*
4. *Each region is rectangular and its aspect ratio falls within a set range.*

For the purposes of this thesis, we only require the basic condition to be fulfilled for a valid floorplan - that is - as long as each region in a floorplan contains a sufficient amount of resources for its assigned netlist primitives, the floorplan is considered valid. We do this to maximize flexibility; by minimizing the conditions that a floorplan must meet, we allow VPR to support a fairly general set of constraints.

The criteria for the cost of a floorplan and the resources it must provide may differ between different floorplan problems. ASICs only contain one resource to implement their netlist primitives - silicon area. Therefore, ASIC floorplans only have to have to optimize for parameters such as silicon area and wirelength. In contrast, modern FPGAs are heterogeneous devices, as discussed previously, and offer multiple types of resources. Therefore, any region that is created must include enough of each resource type required by its netlist primitives. Due to this key difference, some techniques that have been explored for ASIC floorplan generation cannot be applied directly for FPGAs.

2.3.2 Floorplan Generation Techniques

The methods used for generating ASIC floorplans have largely been either analytical or based on iterative improvement of solutions [52]. An example of an analytical approach can be seen in [81], where the authors optimize the packing of soft modules (i.e. modules which do not have fixed aspect ratios) into a fixed die area, such that wire length is minimized. Their technique is to floorplan in two stages. The first stage uses an objective function for minimizing wirelength to create a rough floorplan. The second stage then minimizes the overlap between the modules from the first stage. Although the analytical approach was found to be more run time-efficient than established simulated annealing methods, this approach would not translate well to FPGAs because of the shifting of the modules that occurs during the overlap reduction stage. In FPGA floorplans, shifting the regions in a legal solution may result in an illegal solution, since the new location of the region may contain different resources.

The most common iterative improvement technique for floorplanning is simulated annealing. Simulated annealing is a widely used optimization technique that perturbs an initial solution to generate neighbouring solutions (called *moves*) and calculates the cost of the moves relative to the previous solution. Moves with a lower cost are always accepted as the new solution, while moves with a higher cost are accepted with probability $e^{-\delta_c}/T$, where δ_c is the change in cost caused by the move and T is the current temperature of the anneal. The temperature of the anneal influences how many higher cost moves are accepted or rejected, and thus by extension, how widely the solution space is being explored. The acceptance of higher cost moves is to avoid falling into local minima while searching for optimal solutions. The temperature is initially set to a high value, so that most moves will be accepted. The temperature is gradually cooled (lowered) so that higher cost moves are accepted less frequently, and eventually only lower cost moves are accepted. The implementation of simulated annealing can vary widely based on a number of factors, including but not limited to:

- How the initial solution is generated
- How the solution space is represented
- How moves are created (i.e. how previous solutions are perturbed to create neighbouring solutions)
- How the cost of each move is evaluated
- How quickly or slowly the temperature is cooled

The way these characteristics are chosen can greatly affect the effectiveness of simulated annealing.

Simulated annealing has been extensively studied for ASIC floorplanning [2, 3, 14]. Much work has been done on the creation of different floorplan solution representations such as the sequence pair method [44], slicing tree [55], B*-tree [11], and O-tree [26] in a bid to improve the effectiveness of simulated annealing for floorplanning [81]. Simulated annealing had also been studied as a technique for FPGA floorplanning. However, many of the above-mentioned simulated annealing techniques and representations that were used for ASIC floorplanning must either be heavily adapted for FPGA floorplanning, or cannot be directly applied at all. From this point, we will examine the floorplan generation techniques most relevant to FPGA floorplanning. For each method we examine, we will especially pay attention to:

- Which algorithms and solution representations were used
- Which FPGA architectures were used to evaluate the floorplanner
- How was the quality of the generated floorplans measured
- What were the characteristics of the generated floorplans with respect to region shape

In [16], Cheng and Wong present the first floorplanning algorithm created for heterogeneous FPGAs. The authors use simulated annealing and slicing tree floorplan representations.

A slicing floorplan is created by recursively cutting a rectangle into smaller rectangles. The relations between the rectangles in the floorplan are depicted using a bi-partitioning tree (i.e. a slicing tree). In the slicing tree, leaf nodes represent the partitions, while each internal node is labelled H or V, to specify whether the partitions that are part of their leaf nodes are separated by horizontal or vertical cut lines respectively. An example of a slicing floorplan solution and its slicing tree is shown in Figure 2.6.

Cheng and Wong extended the use of the slicing representation, by creating a new form of this representation specifically for FPGAs. They call this representation Irreducible Realization Lists (IRLs). An IRL is defined as a set of regions that each individually satisfy the resource requirements of a partition, with the bottom left corner of each region being set at a particular point on the FPGA. The regions are called irreducible because the list is made up of the smallest possible regions for each aspect ratio. As an example, consider a partition that requires 12 CLBs, 1 RAM, and 1 multiplier. Figure 2.7, taken from the paper, shows examples of two IRLs for this example partition,

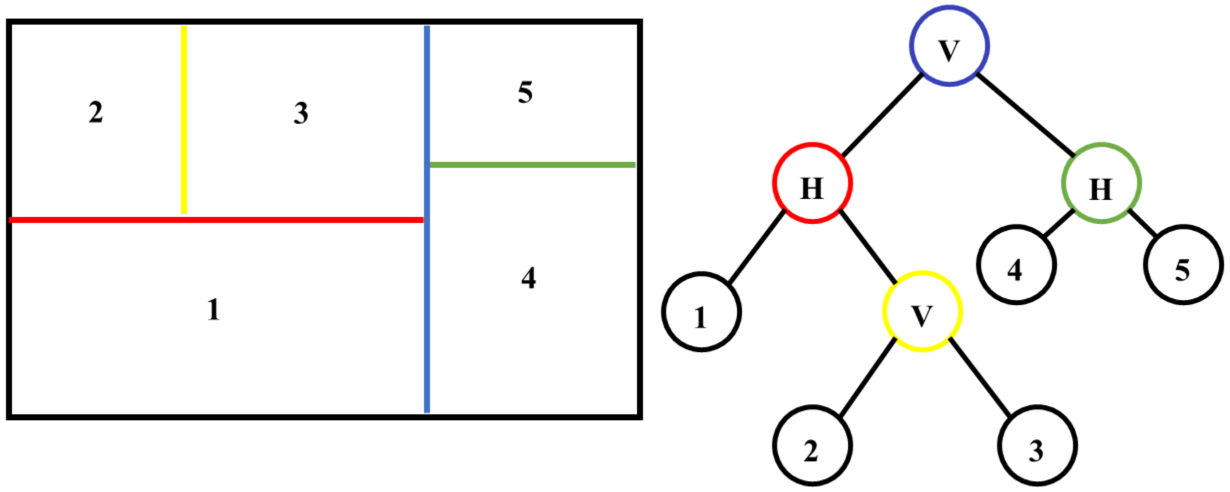


Figure 2.6: A slicing floorplan and its corresponding slicing tree representations. The nodes corresponding to cutlines on the floorplan are each given the same colours as their respective cutlines.

.

one based at (4, 1), and another at (10, 0). The IRL based at (4, 1) contains two rectangular regions. Both regions contain the required amount of each resource, and both regions cannot be made any smaller without changing the aspect ratio. The same can be said of the four regions which make up the IRL based at (10, 0). In the figure, the white squares represent CLBs, the white rectangles represent multipliers, and the black rectangles represent RAMs.

An initial solution is generated for simulated annealing by a traditional area driven floorplanning tool. The simulated annealing algorithm then uses a cost function which is a linear summation of the area of the regions, the aspect ratio, and the wire length of the solution to create new solutions. There is a step called compaction which allows for the creation of rectilinear (rather than just rectangular) regions. An example of a rectilinear region would be an L-shaped or a T-shaped region. This step is done because it allows more modules to be able to fit on the chip. Finally, a post-processing step is performed to make the region shapes more square, while also distributing unused space more evenly on the chip.

Cheng and Wong evaluate the run time of their floorplanning algorithm on the Xilinx XC3S5000 FPGA.

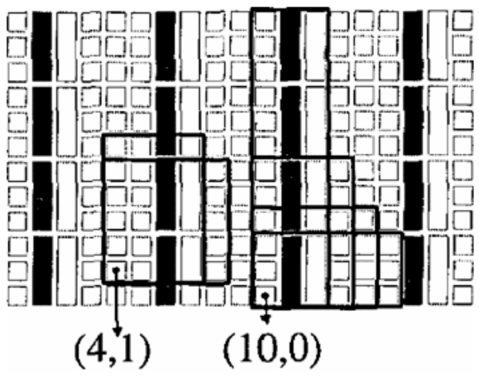


Figure 2.7: Two Irreducible Realization Lists (IRLs), set at (4, 1) and (10, 0) respectively, for a partition which requires 12 CLBs, 1 RAM, and 1 multiplier. The white squares represent CLBs, the white rectangles represent multipliers, and the black rectangles represent RAMs [16]

.

Murray and Betz [45] develop HETRIS, a floorplan generator which is based off of the work by Cheng and Wong. They add optimizations to the algorithm, such as using a dynamic programming approach for IRL calculations, which reduces the runtime. Their cost function during simulated annealing is based off of the area and wirelength of the floorplan. They evaluate the runtime of the improved algorithm by generating floorplans for titan benchmark circuits on a Stratix IV-like architecture. The floorplans generated are also composed of rectilinear regions, as in [16].

Banerjee et al. developed a floorplan generator that uses a three-phase greedy recursive bipartitioning algorithm [5]. Like the previous methods, they also use slicing trees to represent their floorplan solutions. To evaluate their floorplanning algorithm, they use nine ASIC benchmark circuits that were adapted for FPGAs and the same Xilinx Spartan-3 architecture that was used by Cheng and Wong. They evaluate the effectiveness of their floorplanner based on its runtime, and also by the wirelength of the circuits when compiled with the generated floorplans. Like [16], they also simplify their solution space by looking for basic tiles (repeated portions of resources) in the Xilinx architecture. The floorplans produced are also rectilinear in shape.

Yuan [80] presents a floorplanner that creates partitions while prioritizing less ubiquitous resources. The floorplanner uses a *Less Flexibility First* approach - resources that are less available are taken up first. They evaluate their floorplanner using randomly generated circuits on a Xilinx XC3S5000 FPGA. The floorplans generated are made up of rectangular regions.

Mehta and Feng [19] make the floorplanning problem a fixed-outline problem (i.e. their floorplan will always take the same amount of area on the FPGA) rather than an area-minimization problem like Cheng and Wong. Their cost function includes a resource term which incurs penalties based on how different available resources are from needed resources. They have a 2-stage algorithm - the first stage is simulated annealing, and the second stage is a min-cost max-flow network formulation. Their floorplans are generated for modified ASIC benchmarks on the Xilinx XC3S5000 FPGA, and evaluated based on how long it takes to generate valid floorplans. Like the other papers we have seen, their floorplans are also made up of rectilinear regions due to a post-processing compaction step.

Singhal and Bozorgzadeh [66, 65] use a sequence pair representation, which defines relative region positions via horizontal and vertical constraint graphs [44] and fixed-outline simulated annealing. They generate floorplans for MCNC benchmark circuits on a Xilinx Virtex 4 architecture. Their floorplans are composed of rectangular regions, and evaluated by the area of the floorplan, as well as the runtime needed to generate it.

In [60], Sadeghi et al. present a two-stage approach. They present a genetic algorithm that is used to perform simultaneous floorplanning and placement. They test their algorithm with modified MCNC benchmarks and evaluate the time, area, and delay of the synthesized designs. However, their algorithm is only created for FPGAs with homogeneous resource distributions, which are not commercially popular. Their algorithm involves reshaping and relocating regions, which is not possible for heterogeneous FPGAs.

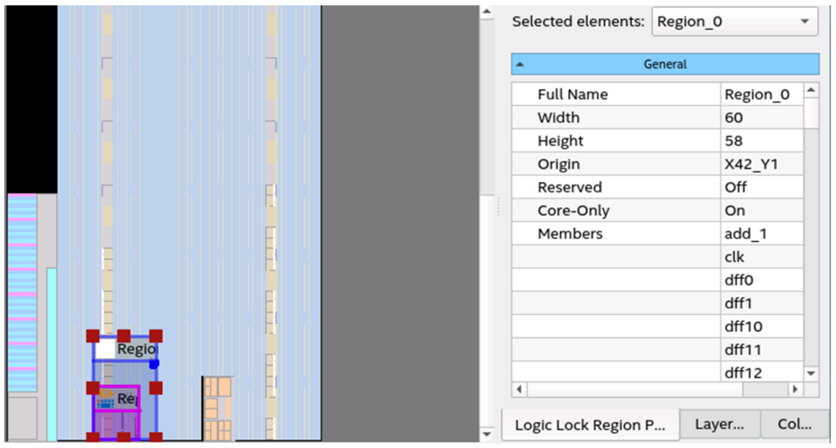
Nguyen and Kumar use recursive bipartitioning heuristics to generate floorplans for Xilinx FPGAs [54]. They created floorplans for partitions that contained a mix of resources - each partition contains CLBs, and at least one DSP and BRAM each. They evaluate the quality of the results by synthesizing the circuits with the generated floorplans, using the commerical Xilinx CAD tools, and reporting the operating frequencies of the synthesized designs. The generated floorplans are made up of rectangular regions.

2.3.3 Comments on FPGA Floorplanners

Of all the floorplan generators discussed above, many use randomly generated benchmarks, or small benchmarks that no longer reflect the size of modern circuits. Furthermore, many are limited to testing with Xilinx FPGAs due to the lack of non-commercial tools that can implement packing, placement, and routing while adhering to placement constraints. The resulting floorplans are often evaluated based on their legality, and the runtime taken to produce them. Ideally, full flow evaluations would be performed after producing the floorplans, to study impacts on circuit quality in terms of speed and wirelength. The two papers which performed full-flow evaluations - [54] and [60] - were not using realistic benchmarks, and in the case of [60], were not using heterogeneous architectures. To further study the area of FPGA floorplan generation, full-flow evaluations must be performed on more realistic benchmarks circuits across a greater variety of architectures. To this end, being able to perform placement while adhering to floorplan constraints would be an important addition to VPR - an open-source tool that can support many different FPGA architectures.

2.4 Current state of Placement Constraints in CAD Tools

Intel Quartus Prime and Xilinx Vivado are two widely-used commercial FPGA CAD tools. They both have the ability to compile FPGA designs with placement constraints. Quartus Prime has a default flow which performs flat compilation on FPGA designs. However, other flows and features such as block-based compilation [21], incremental compilation [22], and partial reconfiguration [23] are available. Block-based design is another name for the modular, team-based design flow that was mentioned earlier in this chapter. These flows are recommended by the user guide for large designs. When using these flows, users are highly recommended, and in some cases required, to use floorplan constraints. For example, when using the incremental compilation flow to achieve timing closure, the user guide highly recommends using floorplan constraints as it helps achieve timing closure with fewer iterations [22]. Also, floorplan constraints are required for the block-based compilation flow to avoid resource conflicts [21]. Users can use LogicLock regions to specify the size and location requirements of the floorplan constraints needed for their designs. Quartus' Chip Planner is available to visualize the floorplan regions that are created. An image of the Chip Planner showing the floorplan regions created with LogicLock is shown in Figure 2.8. The Vivado design suite also supports hierarchical design flows and partial reconfiguration [77], [78]. Users can set floorplan constraints (called AREA_GROUP constraints in Vivado) for these flows using the design analysis tool Xilinx PlanAhead.



Selected elements: Region_0

Full Name	Region_0
Width	60
Height	58
Origin	X42_Y1
Reserved	Off
Core-Only	On
Members	add_1
	clk
	dff0
	dff1
	dff10
	dff11
	dff12

Figure 2.8: Chip Planning with Quartus LogicLock

.

Currently, adhering to placement constraints and floorplans is not possible in VTR [51]. CAD flows that use incremental synthesis or partial reconfiguration must often rely on commercial tools such as Xilinx's Vivado or Intel's Quartus Prime. Open-source tools exist, but they often only target specific FPGA architectures. For example, OpenPR is an open-source partial reconfiguration toolkit that can be used with Xilinx FPGAs [67]. Nextpnr is an open-source Verilog-to-bitstream CAD flow that can adhere to placement constraints, however, the flow supports only the Lattice iCE40 and Lattice ECP5 device families [61]. Adding placement constraints to VTR would be a stepping stone to create more efficient CAD flows for modern FPGA designs across a variety of architectures. VTR-based CAD flows encompass several stages from logic synthesis all the way to timing analysis. The contributions of this thesis will be to Versatile Place and Route (VPR) - the portion of VTR that performs packing, placement, and routing.

2.5 Summary

In this chapter we have highlighted the diversity of FPGA architectures and presented FPGA CAD flows. We examined the issues facing FPGA designs - chiefly long compile times and difficulty achieving timing closure - and also explored some of the techniques used in CAD flows to resolve these issues. We have also given an overview of floorplanning and discussed the current state of automatic FPGA floorplanning tools. The key point to note from this chapter is that all of the techniques presented need some level of control that allows packing and placement to occur while adhering to placement/floorplan constraints. The work of this thesis is to make VTR floorplan-aware, in order to create more efficient CAD flows that fully utilize the increasingly diverse resources available on modern FPGAs. In the next chapter we will present the way in which placement constraints can now be expressed in VTR.

Chapter 3

VPR Floorplanning Infrastructure

3.1 Design Goals

Versatile Place and Route (VPR) is the portion of VTR that performs the packing, placement and routing of FPGA designs. It is characterized by its flexibility - unlike commercial tools that target FPGA architectures from a specific vendor, VPR algorithms implement designs on a variety of FPGA architectures. This flexibility allows it to evaluate and compare the benefits of different FPGA architectures.

In order to remain relevant and useful as an FPGA CAD tool, VPR must keep its compile times competitive for large FPGA designs. As discussed in Chapter 2, many techniques used to address long compile times and excessive design iterations involve breaking designs down into sub-designs and compiling them separately. To effectively use these techniques, however, CAD tools must be able to adhere to placement constraints during the physical implementation process.

VPR 8 does not have this capability. Prior to the work done in this thesis, VPR only had the ability to lock down post-packed blocks to specific (x, y, z) locations on the grid and the ability to place blocks in a carry chain in their required orientations. The overarching goal of this thesis is to add user-defined placement constraints as a feature of VPR. We must do this in an architecture-independent way so that the constraints are usable with a wide variety of FPGA architectures. Also, the constraints specification itself should be flexible so that it can be used by both researchers and end-users of novel FPGAs to constrain the tools in different ways for different purposes. In order to do this, we must

1. Facilitate the creation of partitions, which can be assigned primitives and assigned to different physical regions on the FPGA chip.

2. Ensure that during packing, no primitives with incompatible placement constraints are packed together, and - in cases with rigid floorplan constraints - packing is done densely enough to satisfy the constraints.

3. Ensure the placement constraints are adhered to during the placement flow.

4. Add the above changes in a way that upholds the quality of results (QoR) and does not excessively increase compilation times.

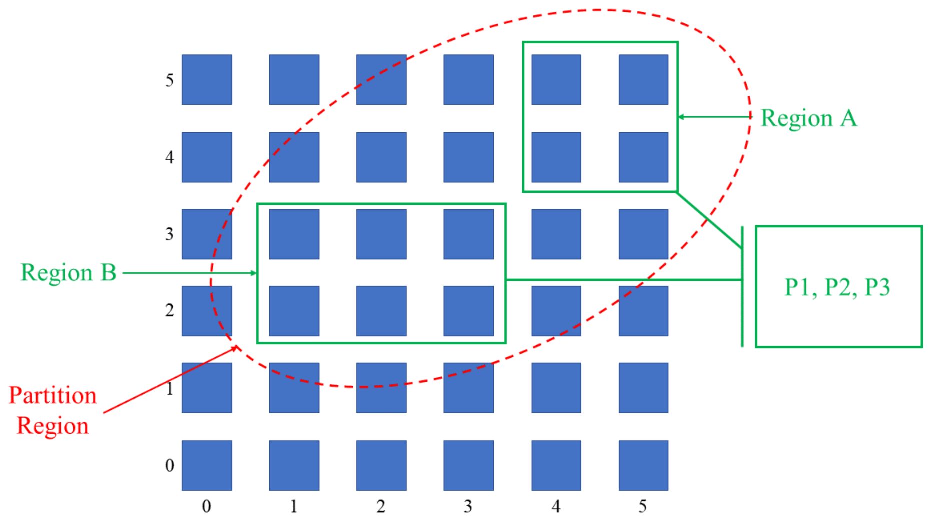


Figure 3.1: A VPR Partition - composed of Regions A and B, and primitives P1, P2, and P3

.

3.2 VPR Constraints Definitions

In this section, we will define some terms related to the implementation of placement/floorplan constraints which will be used throughout the rest of this thesis. Please note that the terms *placement constraints* and *floorplan constraints* will be used interchangeably for the remainder of this thesis.

Region A *Region* is a physical, enclosed, rectangular area on the FPGA chip. It is described by two sets of coordinates (xmin, ymin) and (xmax, ymax). These bounds define the inclusive boundaries of the rectangular area. It can also optionally include a sub-tile value. In the VPR grid description, a physical block located at any given (x, y) coordinate may contain several possible locations - these locations are referred to as sub-tiles. The option to include a sub-tile is most useful if the user wishes to lock a particular group of primitives down to one physical location on the grid. To do this, they can make xmin = xmax, ymin = ymax, and specify a sub-tile. In Figure 3.1, there are two regions. Region A would be described by the two points (4, 4) to (5, 5) and Region B would be described by (1, 2) to (3, 3).

Partition Region A *Partition Region* (also referred to as a *PR* in this thesis) is the union of all of the physical regions in a partition. The primitives that are assigned to a partition must be placed within the bounds of its partition region. In Figure 3.1, Region A and Region B together make up the *Partition Region* of the *Partition*.

Partition A *Partition* is an entity that is assigned logical primitives and physical regions, as they are defined above. There are no limitations on the number of primitives it can contain, nor on the number of regions. In Figure 3.1, the two Regions A and B and the group of primitives P1, P2, and P3 together encompass a VPR Partition.

3.3 VPR Infrastructure

A VPR constraints file format was created to allow users to input their desired placement constraints in a flexible and intuitive manner. An XML format was chosen for the file due to the expressiveness, popularity, and ease-of-use of the language. Furthermore, the XML format is already in use for some VPR files, such as the architectural description file and the routing resource graph file. Due to this prior use, VPR has schema systems in place to aid in the reading of XML files, making XML an ideal candidate for an additional input file.

3.3.1 XML

Extensible Markup Language (XML) is a text formatting language that is commonly used for the exchange of data [73]. A set of standards define the format, and it has been designed to be both human and machine-readable. XML documents are made up of tags, elements, and attributes. Elements are the building blocks of XML - tags enclose the elements and attributes provide information about them. Elements often contain other elements, leading to an inherently hierarchical way of organizing information in XML.

A tag is a component of an XML document that begins with < and ends with >. There are three types of tags in XML: start-tags, end-tags, and empty-element tags. Start-tags and end-tags are used in pairs. In the XML example snippet below, <ticket> is an example of start tag for the element called *ticket*. It has a corresponding end tag </ticket>. The *ticket* element contains two other elements - *destination* and *price*. Similar to *ticket*, the *destination* element also has start and end-tags. It also contains the content Toronto. The *price* element is enclosed by an empty-element tag. It contains the attribute *amount*, which is set to 30.

```
<ticket>
  <destination> Toronto </destination>
  <price amount="30"/>
</ticket>
```

In addition to these basic elements, XML files can also contain comment lines. A comment is enclosed as follows:

```
<!-- This is a comment in XML -->
```

Lastly, an XML file can optionally contain an XML declaration line. This line contains basic information about the XML document, such as the XML version or encoding being used. An example declaration is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
```

These are all the basic components that are used to create a well-formed XML document.

3.3.2 VPR Constraints File

We use an XML Schema Definition (XSD) to specify the format of the VPR constraints file. XSD files are used to describe the legal format of a specific type of XML document. They define which elements and attributes can appear in a document, as well as their respective data types, and default values. They can also specify the order and number of child elements (i.e. elements contained within other elements). Creating an XSD file is beneficial because it eases the the process of validating the data.

A VPR constraints file must contain the following elements:

- *vpr_constraints* - the top-level element which contains all the information about the vpr constraints
- *partition_list* - the element which contains 1 or more partitions
- *partition* - the element which contains the information for an individual partition (i.e. the primitives that are in it and the regions they are constrained to on the chip)
- *add_atom* - the element that is used to assign a primitive to a partition
- *add_region* - the element that is used to assign a region to a partition

In a VPR constraints file, there is only one vpr_constraints element and one partition_list element allowed. It must include at least one partition element, and each partition element must include at least one add_atom element and one add_region element. There are no limits on the number of regions or atoms a partition may contain, and there are no limits on the number of partitions allowed in a constraints file.

Some of the above-mentioned elements have attributes. These include the following:

- Partition elements must contain a string *name* attribute to specify the name of the partition.
- Add_atom elements must contain a string *name_pattern* attribute to specify the name or regular expression name pattern of primitive(s).
- Add_region elements must contain the integer attributes *x_low, y_low, x_high* and *y_high.* They can also optionally contain an integer attribute *subtile*, if the user wishes to constrain a primitive to a particular sub-tile location.

The combination of these elements and attributes covers the needs of the user for specifying VPR constraints. As discussed in chapter 2, floorplans are used to lock down timing critical parts of a design. They are also used to lock down all or part of a design to reduce subsequent compile times or preserve timing on already compiled modules. Floorplanning partitions are assigned to rectangular areas, or irregular rectilinear regions. Rectilinear areas can be easily described with this XML format by adding multiple regions to a partition.

Regular expressions can also be used to specify primitives with the add_atom element. If a regular expression is used, any primitive that has any part of its name matching the regular expression will be added to the partition.

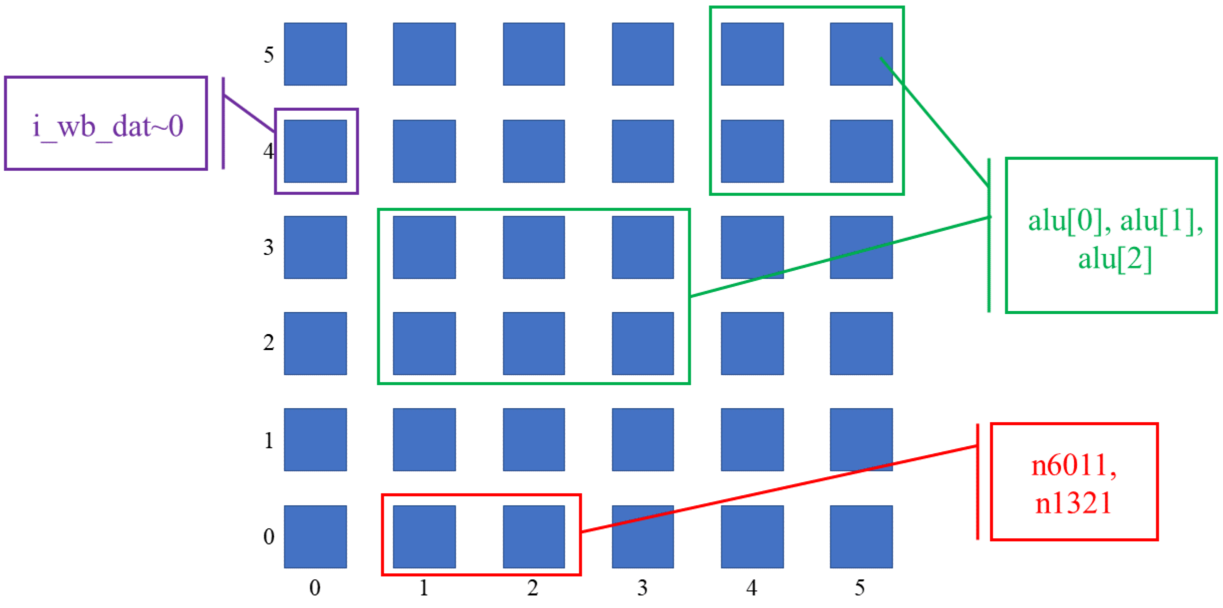


Figure 3.2: Three simple partitions

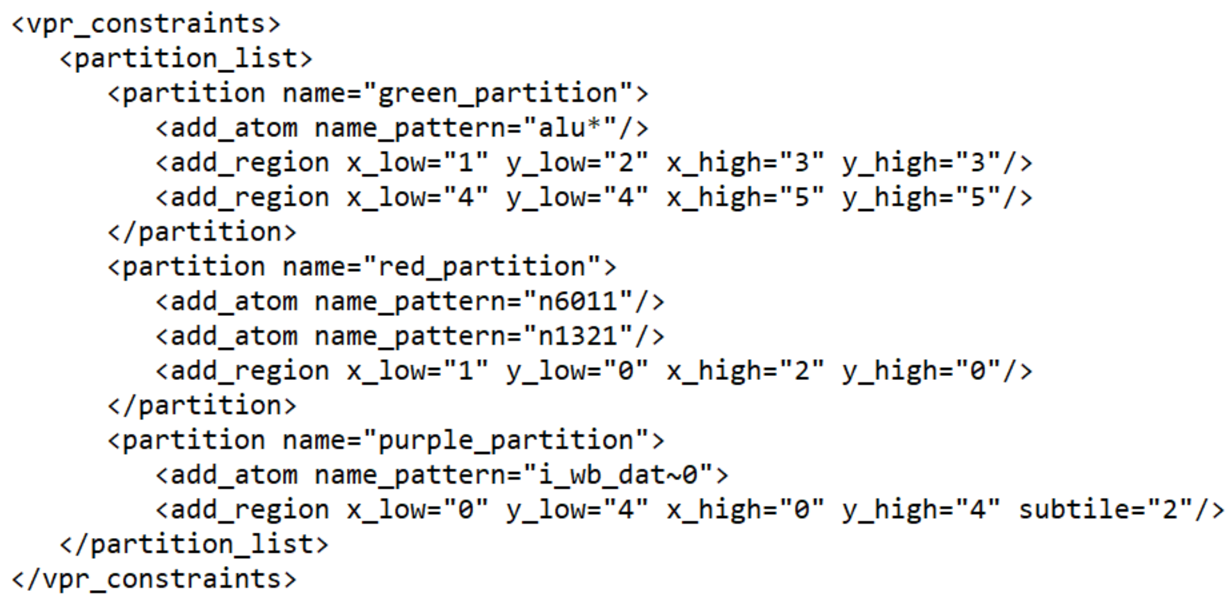
.

3.3.3 Example Constraints File

For the purposes of demonstrating how to write a VPR constraints file, Figure 3.2 shows three simple partition examples. Figure 3.3 shows a constraints file that could be written to describe these constraint partitions. The green partition has two regions from (1, 2) to (3, 3) and (4, 4) to (5, 5). It contains primitives alu[0], alu[1], and alu[2]. The XML constraints file specifies the two regions and uses the regular expression *alu** to describe the name pattern of the three atoms that must be added. The red partition has one region from (1, 0) to (2, 0). It contains primitives n6011, and n1321. These primitive are added to the red partition in the XML file with their exact name patterns. The purple partition has a primitive i_wb_dat∼0 constrained to grid location (0, 4). In the XML file, the region attributes specify the grid location, and there is also a subtile value specified, as an example of how this attribute would be used. In this case, the primitive i_wb_dat∼0 would be constrained to subtile location 2 at the grid location (0, 4).

3.3.4 Naming in VPR Flows

The names of primitives passed into VPR can have an impact on the manual creation of placement constraints. If the placement constraints are generated by a tool, the names will not matter to the end user. However, when a user is manually writing constraints, it is convenient to have hierarchically-named primitives. For example, hierarchical naming is useful for cases when users wish to add all the primitives in a certain module to the same partition. If module names are preserved, they can use a regular expression to add all primitives in the module to the same partition.



```
<vpr_constraints>
   <partition_list>
      <partition name="green_partition">
         <add_atom name_pattern="alu*"/>
         <add_region x_low="1" y_low="2" x_high="3" y_high="3"/>
         <add_region x_low="4" y_low="4" x_high="5" y_high="5"/>
      </partition>
      <partition name="red_partition">
         <add_atom name_pattern="n6011"/>
         <add_atom name_pattern="n1321"/>
         <add_region x_low="1" y_low="0" x_high="2" y_high="0"/>
      </partition>
      <partition name="purple_partition">
         <add_atom name_pattern="i_wb_dat~0">
         <add_region x_low="0" y_low="4" x_high="0" y_high="4" subtile="2"/>
   </partition_list>
</vpr_constraints>
```

Figure 3.3: A constraints file that describes the partitions in Figure 3.2

.

VPR's input primitive netlists are written in Berkeley Logic Interchange Format (BLIF). The most commonly used directives in VPR BLIF files are ".model", ".names", ".latch" and ".subckt". The directive ".model" is used to describe a flattened hierarchical circuit, and one BLIF file can have have multiple models specified - there can be a model for each module in the hierarchy of the circuit. The ".names" and ".latch" directives specify primitives such as logic gates and latches respectively. Lastly, the ".subckt" directive is used in VPR to specify black-box architectural primitives such as RAMs and multipliers. We examine the primitive names from BLIF files that are produced by two different logic synthesis flows used by VTR and discuss whether they can be improved.

The VTR Titan benchmarks are run through Quartus for the elaboration stage, and so the names in the Titan BLIF files provide insight into the primitive names coming out of Quartus. We examine some of the primitive names that were generated by Quartus for the stereo_vision benchmark circuit. These names are shown in Figure 3.4.

From 3.4, we see that module names such as sv_chip:sv_chip0_inst are preserved in the primitive names. This indicates that these names would be convenient for end users who would like to use regular expressions to add primitives to partitions.

Another flow is the default VTR flow, which runs ODIN II (used for logic synthesis and elaboration) and ABC (used for logic optimization and technology mapping). The primitive names coming out of ABC for the arm_core VTR benchmark circuit are shown in Figures 3.5 and 3.6.

From Figures 3.5 and 3.6, it can be seen that some hierarchical module names are preserved in the primitive names, such as arm_core.u_fetch.u_cache.u_tag0. However, there is an issue that arises from the randomly generated latch names. ABC changes all internal output names to a random alpha-numerical value, such as _n23109 or _n23114. This name change would make it inconvenient for the user to manually add multiple primitives, as they would have to individually add each randomly

```
.names gnd tm3_sram_addr2[17]
1 1

.names gnd tm3_sram_addr2[18]
1 1


# Subckt 0: tm3_sram_data0[0]~output_I
.subckt stratixiv_io_obuf \
    oe=sv_chip0:sv_chip0_inst|tm3_sram_data[0]~en \
    i=sv_chip0:sv_chip0_inst|tm3_sram_data[0]~reg0 \
    o=__out__tm3_sram_data0[0]

# Subckt 1: tm3_sram_data0[1]~output_I
.subckt stratixiv_io_obuf \
    oe=sv_chip0:sv_chip0_inst|tm3_sram_data[1]~en \
    i=sv_chip0:sv_chip0_inst|tm3_sram_data[1]~reg0 \
    o=__out__tm3_sram_data0[1]
```

Figure 3.4: Primitives names from the Quartus logic synthesis flow for the Titan benchmark stereo_vision

.

```
.subckt adder a[0]=gnd b[0]=vcc cin[0]=arm_core.u_execute^MIN~7152-0[0] cout[0]=arm_core.u_execute^MIN~7152-1[0] \
 sumout[0]=arm_core.u_execute^MIN~7152-1[1]


.subckt adder a[0]=gnd b[0]=vcc cin[0]=arm_core.u_execute^MIN~7152-1[0] cout[0]=arm_core.u_execute^MIN~7152-2[0] \
 sumout[0]=arm_core.u_execute^MIN~7152-2[1]


.names arm_core.u_fetch.u_cache.u_tag0^out~20_FF arm_core.u_fetch.u_cache.u_tag0.RAM^out1~20 \
 arm_core.u_fetch.u_cache.u_tag0^nMUX~1^MUX_2~13792 _n3224
1-0 1
-11 1


.names arm_core.u_fetch.u_cache^select_way~0_FF n13206 arm_core.u_fetch.u_cache^source_sel~0_FF \
 arm_core.u_fetch.u_cache.u_tag0^nMUX~1^MUX_2~13792
--1 1
11- 1
```

Figure 3.5: Primitives names from ABC for the VTR benchmark arm_core

.

```
.latch _n23109 arm_core.u_decode^irq_FF re arm_core^i_clk 0


.latch _n23114 arm_core.u_execute^o_iaddress~4_FF re arm_core^i_clk 0


.latch _n23119 arm_core.u_wishbone.u_wishbone_buf_p2^wbuf_addr_r0~4_FF re arm_core^i_clk 0
```

Figure 3.6: Primitives names from ABC for the VTR benchmark arm_core

.

generated primitive name. Improving the primitive names coming out of ABC is part of the future work of this thesis.

3.4 Summary

This chapter detailed the work done to address the first design goal in section 3.1, which was to facilitate the creation of partitions. The creation of partitions was set up in a flexible way, such that the user has multiple levels of control. A VPR user can lock down a small number of primitives to specific grid locations, or large groups of primitives to large regions of the chip. Addressing the second, third, and fourth design goals listed in Section 3.1 requires algorithmic changes to the packing and placement stages of VPR. These algorithmic changes are discussed in Chapter 4.

Chapter 4

Algorithmic Improvements in Packing and Placement

4.1 Overview

In this chapter, we present the improvements that were made to the packing and placement algorithms of VPR to accommodate CAD flows that use placement constraints. We start by presenting the general packing algorithm of VPR 8, the latest VPR release. We then detail what changes were needed in the packing algorithm and how the necessary changes were made. We then follow the same flow for the placement algorithm.

VPR takes the general approach of performing packing, then placement and keeping solutions legal at all times during the process. We followed this general framework of VPR when enhancing the algorithm. One other approach could be to allow illegal solutions, and assign a gradually-increasing cost to illegal solutions so that the algorithm tends towards choosing legal solutions. Another approach could be to perform a flat placement of primitives that directly adheres to floorplan constraints, and then perform packing, as in [71]. These two approaches are more radical changes from the existing framework of VPR, and it is not clear that they would work better than our approach. Therefore, we chose to minimize changes to VPR by maintaining this general approach of packing followed by placement with solution legality maintained throughout while enhancing both the packing and placement algorithms to ensure placement constraints are respected.

4.2 Packing Algorithm

Packing takes in the primitive netlist produced by the technology-mapping stage and groups the primitives into clusters. The clusters are created such that they can be placed in complex blocks on the FPGA device, such as logic blocks, DSP blocks, or RAM blocks. This stage of physical synthesis is generally only present in FPGA CAD flows. It is done to simplify the problem of placing logic primitives in the heterogeneous resources of an FPGA.

The VPR packing algorithm is a greedy, seed-based clustering algorithm [40]. In a seed-based algorithm, a primitive is chosen to start a cluster, and more primitives are added to the cluster until it is considered full, according to given criteria. Legality checking is performed for each primitive that is added to the cluster. This general method can produce very dense clustering, which can be beneficial in minimizing the area of a circuit, as the number of clustered blocks in a circuit is minimized. However, packing too densely can negatively impact routability and wirelength. The VPR 8 algorithm was therefore adapted to produce more naturally connected clusters [51]. Earlier versions of VTR also followed connectivity-based packing, however, they sometimes used unrelated logic to fill clusters when there was nothing that was connected that could legally fit. The VPR 8 algorithm focuses more on connectivity than it does on filling each cluster.

VPR has a pre-packing step that groups together primitives that must be packed together. In VPR, these groups of primitives are called *molecules*. A carry chain is a common example of when multiple primitives must be put into one molecule - each primitive in the chain must be packed together to use a logic block's carry logic [51]. If a primitive does not have any specific primitives it must be packed with, it will simply be put into its own molecule. VPR's packing algorithm packs molecules into clusters, rather than primitives.

Algorithm 1 The VPR 8 Packing Algorithm

```
Require: The primitive netlist $prim\_netlist$, the target FPGA architecture $arch$
 1: $clusters \leftarrow \varnothing$
 2: $unpacked\_molecules \leftarrow$ PRE_PACKING($prim\_netlist, arch$)
 3: function DO_CLUSTERING($unpacked\_molecules$, $arch$)
 4:     while $unpacked\_molecules \neq \varnothing$ do
 5:         $new\_cluster \leftarrow$ CREATE_CLUSTER($arch, unpacked\_molecules$)    ▷ Picks a seed molecule
 6:         $cluster\_mol\_candidates \leftarrow \varnothing$
 7:         $next\_mol \leftarrow \varnothing$
 8:         while CLUSTER_NOT_FULL($new\_cluster$) do
 9:             $cluster\_mol\_candidates \leftarrow$ FIND_GOOD_MOLS($new\_cluster, unpacked\_mols$)
10:             $next\_mol \leftarrow$ PICK_NEXT_MOL($cluster\_mol\_candidates, new\_cluster$)
11:             if $next\_mol = \varnothing$ then
12:                 break
13:             end if
14:             $molecule\_legal \leftarrow$ TRY_PACK_MOLECULE_INTO_CLUSTER($new\_cluster, next\_mol$)
15:             if $molecule\_legal = true$ then
16:                 $unpacked\_mols \leftarrow unpacked\_mols \setminus next\_mol$    ▷ Molecule is now packed
17:                 $cluster\_mol\_candidates \leftarrow cluster\_mol\_candidates \setminus next\_mol$
18:             end if
19:         end while
20:         $clusters \leftarrow clusters \bigcup new\_cluster$
21:     end while
22:     return $clusters$
23: end function
```

Algorithm 1 describes the general process of clustering in VPR. To start a new cluster, a seed molecule is selected from the unpacked molecules (line 5). Candidate molecules are found that are best suited to add to the cluster (line 9). The molecules are selected based on their connectivity to the primitives already in the cluster. The molecule with the highest gain is picked as the next molecule to add to the cluster (line 10). If a beneficial molecule was found, a series of legality checks are performed to see whether it can be packed into the cluster (line 14). If the molecule passes the checks, it is packed into the cluster, and removed from the unpacked molecule and candidate molecule lists (lines 16 and 17). If no beneficial molecule is found, the cluster is deemed full, and a new cluster is started (line 12). This method of declaring the cluster full was added to VTR 8 to alleviate the issue with overly dense packing that affected earlier versions of VTR. This process continues until there are no more unpacked molecules, at which point the clustered netlist is returned.

4.2.1 Legality Checking during Clustering

Recall from section 3.1 that one of our design goals is to ensure that no primitives with incompatible placement constraints are packed together in a cluster. This condition can be met by adding a placement constraints legality check to the clustering algorithm, which takes place at line 14 in Algorithm 1. The placement constraints legality is checked as follows. Each molecule has a Partition Region *PR* associated with it, based on the primitives it contains. Recall from Chapter 3 that a Partition Region *PR* represents the physical region(s) on the chip to which the primitives in a partition are constrained. If it contains primitives that are constrained, the *PR* of the molecule will match the *PR* of the Partition that the primitives belong to. If it contains no constrained primitives, its *PR* will be empty (i.e. it has no placement constraints so it can be placed anywhere on the chip). Similarly, a cluster has a *PR* associated with it based on the molecules it contains. A molecule is legal to be packed into a cluster (from a constraints perspective) if the *PR* of the molecule is compatible with the *PR* of the cluster. There are four cases that may occur when checking compatibility, and they are discussed below. Note that when we say the molecule is legal to be packed into the cluster, this is from a placement constraints perspective. The other legality checks for packing a molecule into a cluster also still need to be performed.

1. **Cluster is not constrained, Molecule is not constrained**
   (a) The molecule is legal to be packed into the cluster.
2. **Cluster is not constrained, Molecule is constrained**
   (a) The molecule is legal to be packed into the cluster.
   (b) The cluster's *PR* must be updated to match the constraints of the constrained molecule it now contains.
   (c) For any subsequent molecules, the cluster will be treated as a constrained cluster.
3. **Cluster is constrained, Molecule is not constrained**
   (a) The molecule is legal to be packed into the cluster.
4. **Cluster is constrained, Molecule is constrained**
   (a) The *PR* of the cluster is intersected with the *PR* of the molecule.
   (b) If the intersection is not empty (i.e. there is some overlap between the *PR*s), the molecule is legal to be packed into the cluster.
   (c) The cluster's *PR* must be updated to match the intersection of the two *PR*s. Figure 4.1 shows an example of this case.
   (d) If the intersection is empty, the molecule cannot legally be packed into the cluster.

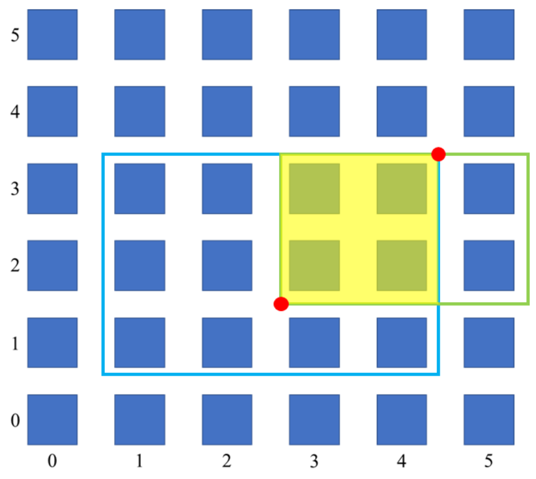


Figure 4.1: The intersection (shaded yellow) of a molecule's PR (outlined in green) and a cluster's PR (outlined in blue) The intersection is encoded by the red points, which bound the opposite corners of the intersection.

In Figure 4.1, we see an example of how a cluster's PR and a molecule's PR would intersect. The cluster's PR is outlined in light blue and covers the area from (1, 1) to (4, 3). The molecule's PR is outlined in green and covers the area from (3, 2) to (5, 3). Their intersection is therefore the area bounded by (3, 2) to (4, 3), which is shaded yellow in the figure. If the molecule were to be packed into the cluster, the cluster's PR would be updated to be the area from (3, 2) to (4, 3).

4.2.2 Attraction Groups

From section 3.1, another design goal of this thesis was to ensure that clustering is done densely enough to accommodate all placement constraints. When clustering with placement constraints, the clustering tends to become less dense. This is because cases may occur where a molecule is a potential candidate to be packed in a cluster, but it cannot be packed into it due to incompatible placement constraints. Therefore, we may end up with cases where some regions on the chip are assigned more clusters than they are able to place. An illustration of this case is shown in Figure 4.2. A group of netlist primitives are constrained to two grid tiles, but packed into three clusters. Therefore, the clustering is not dense enough to adhere to placement constraints.

As another example, let us say that in one flow, twenty clusters of type CLB are placed in a particular region A, which has room for 23 clusters of type CLB. When we run the same flow with placement constraints, 24 clusters of type CLB are created, and they all contain primitives that are constrained to region A. Region A now contains more blocks of type CLB than it has tiles of that type. Therefore, clustering does not meet the requirements of the placement constraints.

We created *attraction groups* to solve situations like the one presented above. An attraction group is a group of primitives that belong to the same constraints partition, and therefore are constrained to the same physical regions on the chip. During clustering, a gain is assigned to each molecule candidate for a cluster based on how beneficial it would be to pack the molecule into the

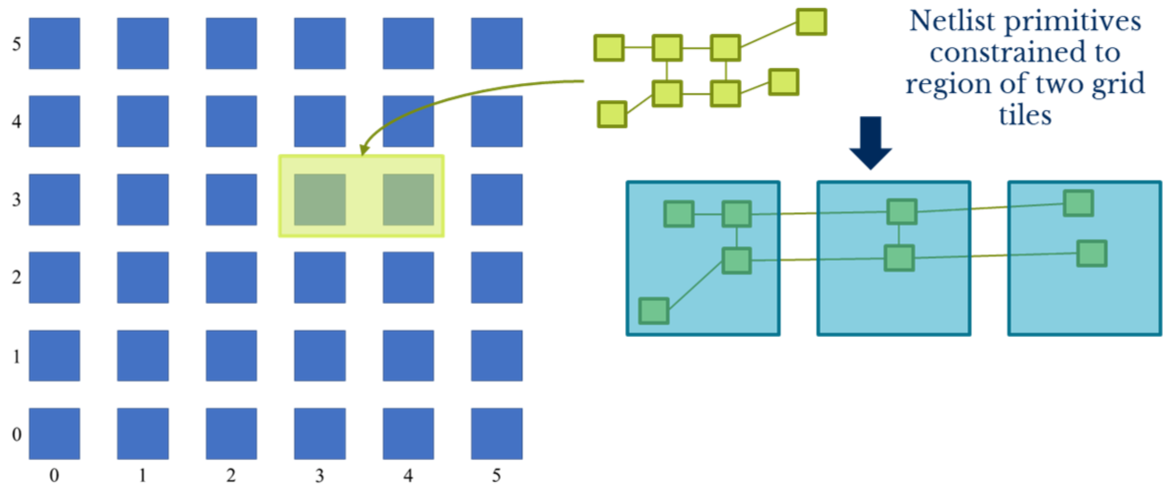


Figure 4.2: A group of netlist primitives are constrained to two grid tiles, but packed into three clusters.

cluster. The benefit of creating attraction groups is that during clustering, the gain of molecules can be increased if they belong to the same attraction group as the prospective cluster. This way, molecules that are part of the same attraction group (i.e. same partition) are more likely to be packed together. Furthermore, when packing a cluster, good molecule candidates are searched for according to their connectivity to the current cluster (line 9 in Algorithm 1). However, if no good molecule candidates are found, the cluster is considered full and a new cluster is created. If a cluster has an associated attraction group, then candidates can also be selected from among the attraction group molecules. This means that the cluster can be packed with more molecules, and therefore more densely. For example, if the primitives from Figure 4.2 were put into an attraction group, they could be clustered more densely, so as to fit into two clustered blocks rather than three, as shown in Figure 4.3.

There is a balance between when to use and not use attraction groups. We found that the attraction groups were very effective in packing clusters more densely (this is discussed in more detail in Chapter 5). However, as mentioned previously, packing too densely can have adverse effects on QoR. Therefore, the attraction groups must be used with a balanced approach. The VPR 8 packing algorithm is capable of performing two iterations. If the first iteration did not succeed in packing all primitives onto the chip, a second iteration was done where unrelated clustering was allowed. This, in turn, led to more dense packing since more molecules could be considered as candidates for each cluster.

We implement a similar iterative approach with our attraction groups. In the first iteration, no attraction groups are created. If the clustering algorithm is able to pack densely enough to satisfy

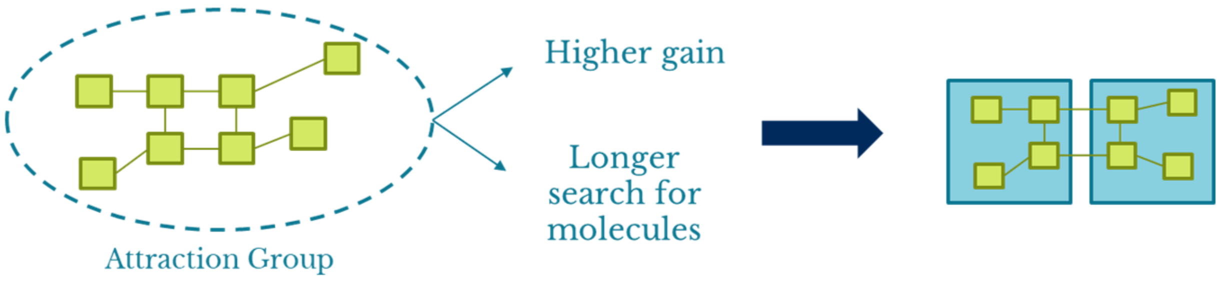


Figure 4.3: A group of netlist primitives are added to an attraction group, helping them to be clustered more densely.

placement constraints, no more iterations need to be done. A check is done after each clustering iteration to calculate how many cluster blocks are constrained to a particular region, and how many tiles of the block's type are available in the region. If the blocks constrained outnumber the tiles available, the region is overfull. In the second packing iteration, attraction groups are created only for the primitives that belong to partitions with overfull regions. This way, those regions can be packed more densely, while other regions which were not overfull do not become any more dense.

For example, let us say that there are placement constraints that are set up as follows: partition A is constrained to region A, partition B is constrained to region B, partition C is constrained to region C, and partition D is constrained to region D. Let us assume that after one packing iteration, regions A and B are found to be overfull. In the second iteration, the primitives from partition A will be put into one attraction group, and the primitives from partition B will be put into a different attraction group. The primitives in partitions C and D will not be in any attraction group, because the regions that are constrained to do not need to be packed anymore densely.

Our packing algorithm can perform up to five iterations to achieve the required clustering density. Multiple iterations were included because in some cases, some regions might become overfull in the second iteration which were not overfull in the first iteration. With each iteration, more attraction groups are gradually created as more regions become overfull. By the last iteration, if the clustering is still not dense enough, an attraction group is created for each partition. If, after five iterations, the packing is still not dense enough for the given placement constraints, the program errors out with a message specifying to the user which constraint regions are overfull, and how many extra blocks are in the overfull regions. The algorithm checks for overfull regions by counting the number

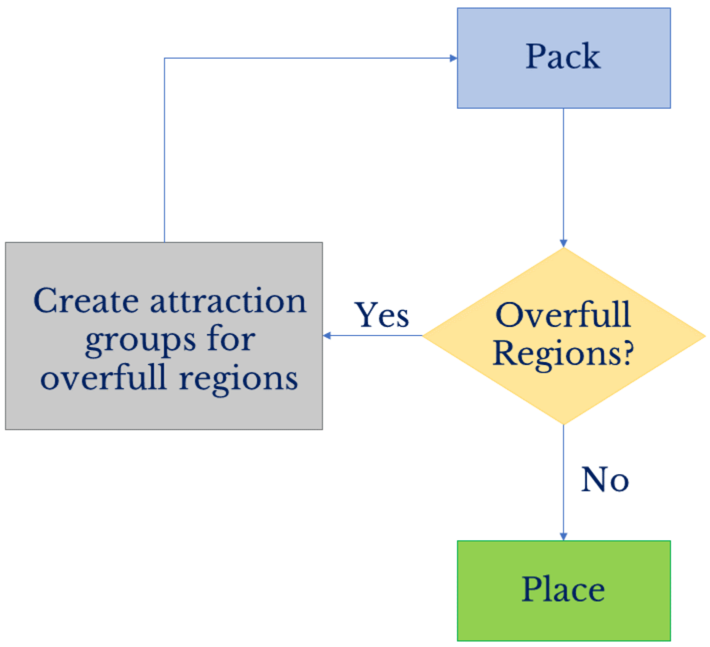


Figure 4.4: The iterative flow of packing with attraction groups.

of blocks of each type that are assigned to each region, and comparing that number to the number of available tiles of each type in the region. If for any type, the number of blocks assigned exceeds the number of tiles, the region is overfull. This flow of this process is shown in Figure 4.4.

4.3 Placement Algorithm

VPR uses a simulated annealing based approach for placement. As described in Section 2.3.2 of this thesis, simulated annealing begins with an initial solution for an optimization problem, and creates neighbouring solutions called moves by perturbing the initial solution. The solutions are accepted or rejected based on the cost and temperature - as the temperature cools, less moves are accepted. VPR has an initial placement stage that randomly places the blocks at valid locations on the grid. It then creates moves by swapping the locations of cluster blocks and macros and calculating the cost of the swap.

4.3.1 Initial Placement

To generate a initial placement solution, VPR 8 placed blocks at the first valid location that was randomly picked from the grid. In Algorithm 2, we present an updated algorithm for initial placement that takes placement constraints into consideration when placing the blocks.

Our algorithm has two pre-processing steps when performing initial placement. The first step is to calculate the tightest possible placement constraints for the blocks that are part of a macro, a step we refer to as constraints propagation (line 2).

Algorithm 2 Constraint-Aware VPR Initial Placement Algorithm

```
Require: The clustered netlist $clustered\_netlist$, the FPGA device grid $grid$
1: function INITIAL_PLACEMENT($clustered\_netlist$, $grid$)
2:     $placement\_macros \leftarrow$ CALCULATE_TIGHEST_MACRO_CONSTRAINTS($placement\_macros$)
3:     $unplaced\_blocks \leftarrow$ SORT_BLOCKS($clustered\_netlist, grid$)
4:     while $unplaced\_blocks \neq \varnothing$ do
5:         while $block\_placed \neq true$ do
6:             $blocked\_placed \leftarrow$ TRY_RANDOM_PLACEMENT($block$)
7:             if $block\_placed = true$ then            ▷ Random placement succeeded
8:                 break
9:             end if
10:            $block\_placed \leftarrow$ DO_EXHAUSTIVE_PLACEMENT($block$)
11:        end while
12:    end while
13:    return $initial\_placement$
14: end function
```

In VPR, placement macros are groups of blocks that must be placed at a specific orientation relative to each other. Placement macros can be created for carry chains or DSP block cascade chains. For example, in a macro with two blocks, the first block will be the head block, while the second block will be placed at a specified offset relative to the head block. If the head block is placed at (2, 10) and the offset of the second block is (0, -1), the second block must be placed at (2, 9). Therefore, if one block in a macro is constrained, that implies a constraint on all of the other blocks in a macro. Constraint propagation for macro constraints makes the generation of legal moves simpler and faster, and the speed of move generation and evaluation is key in simulated annealing.

Our algorithm handles constraints propagation as follows.

1. Check if the macro is constrained by checking whether it has any constrained members (i.e. does any macro member have a non-empty *PR*?). If no macro members have floorplan constraints, no further calculations are performed for the macro.

2. Calculate the *head* PR (i.e. the *PR* of the head member) of the macro by intersecting the *PR*s of all constrained macro members. The offset on each *PR* is reversed before intersecting, since we are performing the calculation for the head macro block.

3. A constraint is applied to each block in the macro (whether it was formerly constrained or not). The constraint applied is that of the head *PR*, with the appropriate offset applied for each member block.

Figures 4.5 and 4.6 after show the example of macro block constraints before and after constraints propagation.

The second pre-processing step is to sort the blocks (Algorithm 2, line 3). This step is necessary to ensure that blocks with tight placement constraints do not get crowded out of their small regions by other non-constrained blocks. The difficulty of placing a block is based on two criteria

1. Is the block part of a macro?

2. Is the block constrained, and if so, how many valid locations are there for the block in its *PR*?

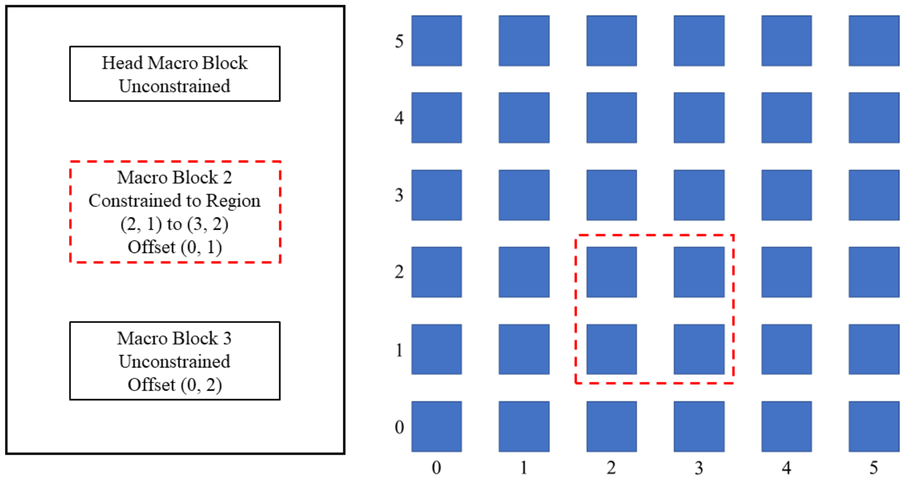


Figure 4.5: The constraints on the blocks in a macro before constraints propagation. Originally, only one block is constrained

.

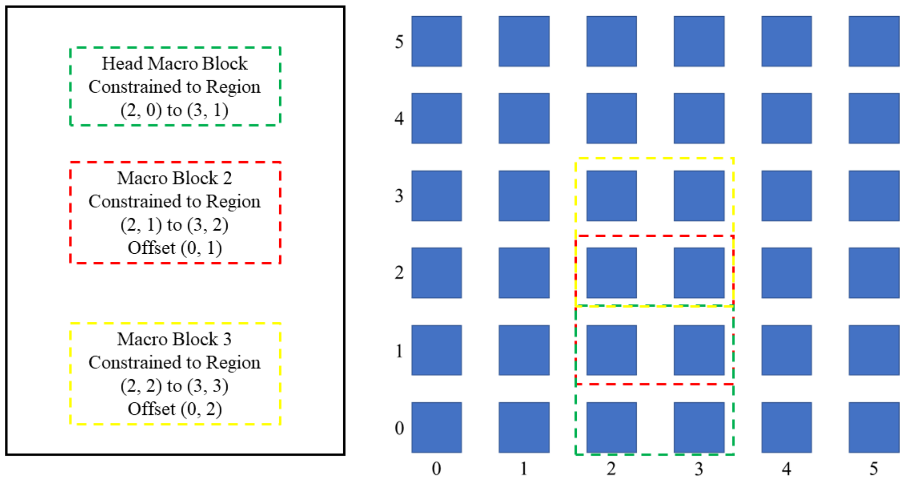


Figure 4.6: After constraints propagation. Due to the constraint on the second block, constraints were also applied to the head block and the third block

.

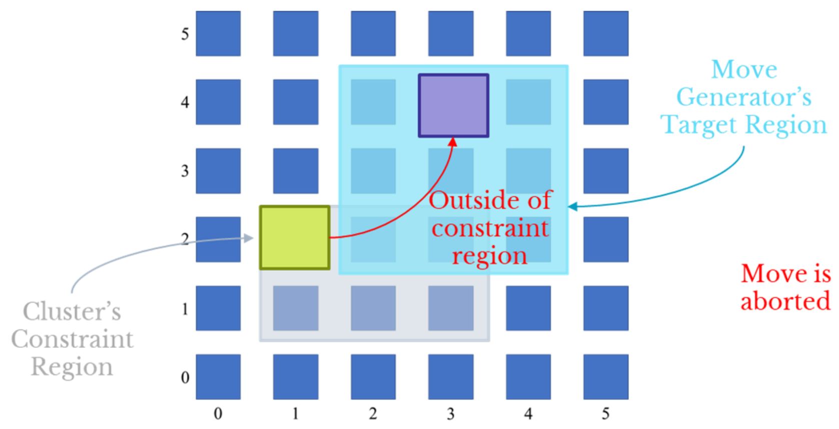

5
4
3
2
1
0
0
1
2
3
4
5
Move Generator's Target Region
Outside of constraint region
Move is aborted
Cluster's Constraint Region

Figure 4.7: If a placement move will move the block out of its constraint region, the move is aborted. In this figure, swapping the green and purple blocks would move the green block outside of its constraint region, and the move is therefore aborted.

.

Based on these criteria, the blocks are sorted in order from most difficult to place to least difficult to place.

After these steps, the blocks are ready for initial placement. For each block, random placement is attempted a limited number of times (Algorithm 2, line 6). The random locations are chosen from within a block's *PR* if the block is constrained. If the block is not constrained, the location is chosen randomly from anywhere on the chip. If the random placement does not succeed (due to the selected locations already being full), exhaustive placement is performed (Algorithm 2 line 10). During exhaustive placement, every valid and available location is tried as a placement location for the block, until a location succeeds. Similar to random placement, exhaustive placement only looks through the *PR* locations for constrained blocks.

4.3.2 Placement Moves

VPR has several move generators for its simulated annealing process [18]. During a placement move, clustered blocks or placement macros may have their locations swapped, or a block or a placement macro may be moved to an empty location. During these moves, a constrained block may be moved out of its *PR*. To prevent this from occurring, a check is performed to see whether the location that each block will be moved to is still within the *PR* of the block. If any block is being moved out of its *PR*, the move is aborted. This process is shown in Figure 4.7.

Running a flow with floorplan constraints was found to produce many aborted moves, especially when blocks were constrained to small regions. This led to an increase in place time. The details of this time increase, as well as the cases when it occurred will be discussed in Chapter 5. To alleviate the time increase, an optimization was performed to lessen the number of aborted moves.

Each time a move is generated for a block, a target region is specified for the move, and the block can only be moved to a location within that target region. To lessen the number of aborted moves, the algorithm was modified to intersect the target with the *PR* of a block if it is constrained, as shown in Figure 4.8. The move generator would not create swaps with blocks that are outside of

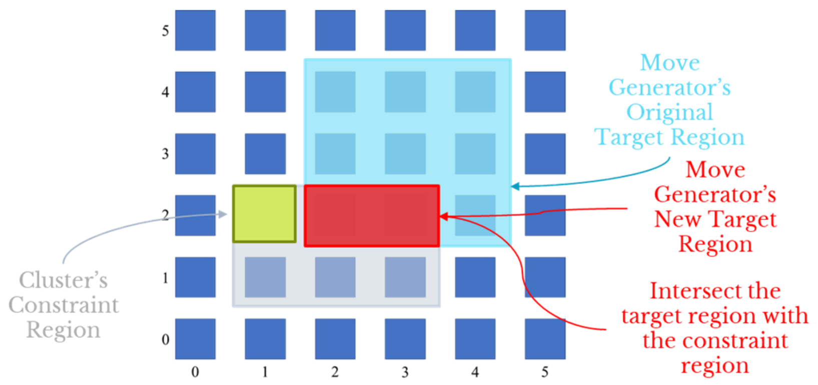

Move Generator's Original Target Region
Move Generator's New Target Region
Intersect the target region with the constraint region
Cluster's Constraint Region
0
1
2
3
4
5

Figure 4.8: The constraint region of a cluster is intersected with the target region of a move generator. The intersection, shown in red, becomes the new target region of the move generator

.

the red region in Figure 4.8, once the intersection was performed. This way, fewer illegal moves are created and placement time decreases in a number of cases. The details of this place time decrease will be discussed in more detail in Chapter 5.

4.3.3 Error Checking

Error checking was added to the following placement steps, to ensure that constraints were being respected.

1. During constraints propagation, if the members of a macro have incompatible placement constraints, an error is reported for that macro. Incompatible constraints in a macro would constitute a user error, since the constraints passed in cannot be implemented. Therefore, a message is printed informing the user of the incompatible macro constraints.

2. After initial placement, if any blocks are still unplaced, an error is reported. The error message tells users which regions are overfull, and by how many blocks.

3. After both initial placement and final placement, all blocks are checked to see whether they are within their respective placement constraints. This double-check allows the program to check that placement constraints are adhered to during the placement stage, before moving onto routing. If this check fails, that would indicate a bug in the placement code.

4.4 Summary

In this chapter, we detailed the enhancements that were added to VPR's packing and placement algorithms. The packer now checks legality to avoid clustering primitives in a way that makes placement impossible, and also activates attraction groups as needed to force a denser packing of portions of the design that are tightly floorplanned. The placer performs constraints propagation prior to initial placement and completes initial placement in a constraints-aware manner. The placement move generators now have checks to ensure that blocks are not swapped out of their placement constraint bounds. In the next chapter, we present experimental evaluation results to determine how effectively the VPR CAD flow can now adhere to placement constraints, while minimizing adverse impacts on QoR and run times.

Chapter 5

Experimental Results

In this chapter we begin by presenting the benchmark circuits and architectures that we used to evaluate our changes to VPR. We also discuss the various kinds of placement constraints we used to test VPR's improved packing and placement algorithms, and the reasoning behind testing with each variation. We present the results of running VPR with these different placement constraints cases and analyze any changes in QoR arising from them. Lastly, we summarize the general effectiveness of the packing and placement algorithms in adhering to placement constraints.

5.1 Experimental Methodology

In order to effectively evaluate VPR's adherence to placement constraints, we present constrained CAD flow results from three different architectures, using two sets of benchmarks circuits. It is important to include results from a variety of architectures because the ability to evaluate results across different architectures is a key feature of the VPR CAD flow. We also consider a variety of constraint patterns, from coarse to detailed, to examine how the algorithms handle constraints as they get more rigorous.

The first set of benchmarks we use are the VTR benchmarks [59]. The VTR benchmarks are a set of 19 benchmarks which come from various applications such as Monte Carlo simulations of photons and and LU Decomposition methods [59]. We also gather results using the Titan23 benchmark suite [49, 50]. The Titan23 suite contains circuit designs which are more comparable to modern FPGA circuit sizes than the VTR benchmarks, and also make more use of the heterogeneous resources available on FPGAs [52].

There are two FPGA architectures that we use with the VTR benchmarks. The first is *k6_frac_N10 frac_chain_mem32K_40nm.xml* - one of the flagship architectures that was released with VTR 7.0 [59]. It is a 40 nm architecture that contains clusters of 10 fracturable LUTs, which can operate as either one 6-input LUT or two 5-input LUTs. This architecture will hereafter be referred to as the K6 architecture. The second - *k4_N8_topology-0.85sL2-0.15gL4-on-cb-off-sb_22nm_22nm.xml* - is a 22 nm architecture that contains clusters of 8 non-fracturable LUTs. This architecture will hereafter be referred to as the K4 architecture. Lastly, the architecture used with the Titan23 benchmark circuits is *stratixiv_arch.timing.xml* - an approximate capture of the Stratix IV FPGA architecture.

In order to effectively evaluate adherence to placement constraints, we test with a variety of constraint cases ranging from easy to difficult. To automatically generate constraints files across the variety of circuits we are testing with, a VPR constraints writer was implemented which generated constraints files based on a previous VPR placement. The automatically generated constraints files would contain only one rectangular region per partition, due to the way the writer was implemented. This is acceptable because the most common case when running with placement constraints is for a group of primitives to be constrained to one rectangular region. We also created cases where constraints files contained multiple regions per partition, which together formed L-shaped and T-shaped PRs; however, these constraints files were created manually.

In order to generate a constraints file for a given circuit, a run of VPR was first performed with no constraints. Based on the placement generated by this initial run, the VPR constraints writer would produce a file with the desired number of partitions (specified by the user). The user could control how many partitions were created by specifying the desired split factors in the x and y dimensions of the grid.

For example, a user could specify that the grid was to be split once in the x dimension, and zero times in the y dimension. The generator would create two partitions, splitting the grid into halves across the x-dimension, as shown in Figure 5.1. All of the primitives that were placed in the left partition would be constrained to the left partition while all of the primitives that were placed in the right partition would be constrained to the right partition.

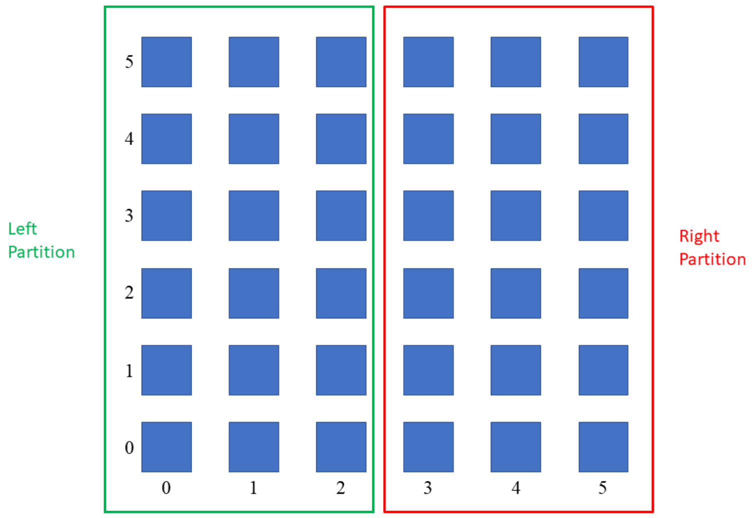


Figure 5.1: If a user specified that the grid should be split into halves across the x-dimension and have no split in the y-dimension, the VPR constraints writer would create two partitions. Each partition would be assigned a region that covers one half of the grid respectively.

.

The process of automatically generating constraints files is shown in Figure 5.2. It is important to note that this constraints generator was created with the purpose of generating *test* constraints. It generates physically reasonable floorplans, but it does not attempt to create floorplans that assign different design hierarchies or modules to different regions of the chip. Therefore, it is useful to test VPR's ability to respect constraints, but is not sufficient to generate floorplans for modular design or partial reconfiguration of a part of a design.

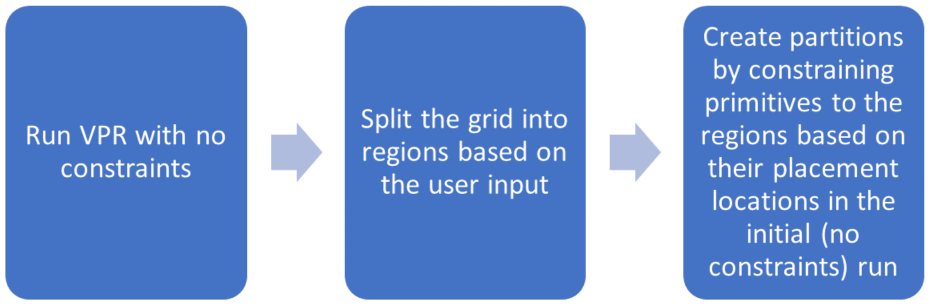

Run VPR with no constraints
Split the grid into regions based on the user input
Create partitions by constraining primitives to the regions based on their placement locations in the initial (no constraints) run

Figure 5.2: The process of generating a VPR constraints file for testing. Primitives are assigned to partitions based on their placement in an initial run with no constraints.

.

Six categories of constraints files were created for testing purposes. They are as follows:

One Big Partition

This test case places all primitives in the circuit into one big partition, which has a region that is the full size of the grid. This test case is essentially no different from running with a no constraints case, as all primitives are still allowed to be placed anywhere on the grid. However, it was created to check that there was no quality degradation occurring by simply running with placement constraints.

Half Partitions

This test case creates two partitions, each with a rectangular region that takes up one half of the grid. The primitives that were originally placed in the left side are constrained to the left side, and the primitives that were originally placed in the right side are constrained to the right side. There were two kinds of constraints files generated in this category - one where *all* primitives are constrained to one of the two partitions, and one where *half* of all primitives (randomly selected) are constrained to one of the partitions, while the other half are free to be placed anywhere. This is a relatively easy constraints case, as primitives are constrained to relatively large areas of the chip.

Quadrant Partitions

The idea of this test cases is to make the placement constraints more difficult to adhere to than the *Half Partitions* cases. The grid is split once in the x dimension and once in the y dimension in order to create four partitions. Each partition is assigned a region that takes up a quarter of the grid. The primitives that were originally placed in a given quadrant will be constrained to that quadrant. This category of constraints also has two kinds of constraints files - one where all primitives are constrained, and one where half of all primitives are constrained.

Sixteenth Partitions

This test case follows in the same vein as the *Half Partitions* and *Quadrant Partitions* cases. In this case, the grid is split into sixteen partitions, with each partition assigned a rectangular region that takes up a sixteenth of the grid. This test case is meant to be more strenuous for the tool to pack and place, due to the fact that the primitives are constrained to tight areas. It again has two variations: one constraints file with all primitives constrained, and one with half of the primitives constrained.

One Location Partitions

This test case is a very difficult test case for the CAD tool, wherein all primitives are constrained to exactly one location on the grid. It is difficult because packing must be exactly reproduced, and there are many ways to pack primitives into a cluster. The previous placement is used to identify the specific x, y, and subtile location of each primitive so that they can all be constrained to an exact location. This test case is meant to see whether a placement could be re-created from previous flows by simply constraining the primitives to their former locations.

L-shape and T-Shape Partitions

Since the automatically generated constraints files assigned one region per partition, some constraints files were created manually to test with partitions that contain more than one rectangular region. Two partitions are created - one with two rectangular regions that form an L-shape, and one with two rectangular regions that form a T-shape. These shapes often appear in floorplans on which compaction was performed as a post-processing step, as discussed in Chapter 2. They can also be used to match heterogeneous resource distribution - for example, getting near enough to peripheral I/Os, or getting enough RAM without covering too much logic.

Overall, running VPR with the above-described scenarios allows us measure how well VPR can obey placement constraints during packing and placement while minimizing impacts on QoR and run-time. We compare each of the above constraints cases with VPR runs that were performed with no constraints, in order to measure the difference in the following parameters between the two runs: run-time, memory usage, number of blocks, total wirelength, and critical path delay.

For the purpose of analyzing the results, we will separate the six constraints categories above into three result groups as follows. The first broad groups will be the general rectangular constraint cases, and these constraints cases include *One Big Partition*, *Half Partitions*, *Quadrant Partitions*, and *Sixteenth Partitions*. The second group of results include the one location constraint results. The third group of results includes the L and T shaped constraint results. The results have been grouped in this way because the discussions of the first groups naturally have much overlap, while the second and third groups have different characteristics that are best discussed separately.

5.2 General Rectangular Constraints

To evaluate the rectangular constraint cases, we used 10 of the largest VTR benchmarks, as well as 22 out of the 23 Titan benchmarks. We selected the 10 VTR benchmarks based on which

Benchmark	Netlist Primitives	IOs	CLBs	DSPs	RAMs
mcml	165809	36	7196	27	159
LU32PEEng	101542	114	6919	32	167
arm_core	18437	133	1034	0	40
or1200	4530	385	255	1	2
LU8PEEng	31396	114	2062	8	44
stereovision1	19549	115	702	40	0
bgm	24865	257	2136	11	0
sha	2744	38	154	0	0
spree	1229	45	65	1	3
blob_merge	11415	36	623	0	0

Table 5.1: The 10 VTR benchmarks used for evaluation.

benchmarks had the most similar minimum route channel width, due to the fact that we were going to use the same fixed route channel width for all ten benchmarks. We did not use the largest Titan benchmark, gaussianblur, as it has a run-time 4.5x that of the next largest Titan benchmark and therefore increased the CPU time too much in our experiments.

We perform all of the runs with a fixed channel width. The fixed channel width for the Titan benchmarks was 300, as this is the channel width sufficient to route all benchmarks in the latest VPR version [49]. For the VTR benchmarks, the fixed channel width was calculated as 1.2x the largest minimum channel width needed among the 10 circuits, for each architecture respectively. For the VTR benchmarks with the K6 architecture, it was 156. For the VTR benchmarks with the K4 architecture, it was 134. As Table 5.1 shows, the 10 VTR benchmarks used are mcml, LU32PEEng, arm_core, or1200, LU8PEEng, stereovision1, bgm, sha, spree, and blob_merge.

Table 5.2 shows the abbreviations that will be used for the different constraints cases for the remainder of this section.

In this section, we use a run with default VPR options as a baseline (i.e. no placement constraints are specified). We run on VPR trunk code from early 2022, which is a more recent version of VPR than VPR 8.0. We will also have another case to compare against - the case where no placement constraints are used, and unrelated clustering is allowed during packing. Unrelated clustering is when molecules with no connectivity are allowed to be packed together [51]. This setting is off by default in VPR; it is only turned on when the tool expressly needs to pack more densely. In a certain sense, packing with attraction groups, as described in Section 4.2.2 is a form of allowing unrelated clustering. Unrelated molecules can be packed together, with the caveat that they must belong to the same attraction group. However it is not exactly the same, because attraction group molecules are only selected when no more connected molecules can be found. Comparing to both of these base cases shows us how we compare to unconstrained packing in typical density cases and higher density cases. The abbreviation for this case will be AUC (allow unrelated clustering).

5.2.1 QoR

It can be seen from the three tables, 5.3, 5.4, and 5.5 that the unrelated clustering base case produces less clusters - it packs more densely than any of the constrained runs. Table 5.3 shows that across all constraints cases, the variation in the number of blocks produced by the packing stage is relatively low for the Titan benchmarks. However, with the constraints cases of 2_A, 4_A, and

Run Type	Abbreviation	Number of Partitions	Fraction of Primitives Constrained
No Constraints	NC	0	0
No Constraints, Allow Unrelated Clustering	NC_AUC	0	0
One Big Partition	OBP	1	1
Half Partitions - All Primitives	2_A	2	1
Half Partitions - Half Primitives	2_H	2	0.5
Quadrant Paritions - All Primitives	4_A	4	1
Quadrant Paritions - Half Primitives	4_H	4	0.5
Sixteenth Partitions - All Primitives	16_A	16	1
Sixteenth Partitions - Half Primitives	16_H	16	0.5

Table 5.2: The abbreviations used for the various constraint cases.

16_A, we see a 1% decrease in the number of clustering blocks created, relative to the no constraints case. The variation in the number of post-packed blocks is also low for the vtr benchmarks, in the case of either architecture they were mapped to, as shown by tables 5.4 and 5.5. This is an interesting result, because it shows that the number of blocks does not decrease significantly even when packing is done in multiple iterations and attraction groups are used to pack the clusters more densely, as explained in section 4.2.2. Running with placement constraints will naturally produce more clusters, since more molecules (i.e. primitives) will be rejected for each cluster on the basis of incompatible floorplan constraints - therefore requiring more clusters to be made for the rejected molecules. However, in cases where this occurs, the necessity of packing more densely to satisfy constraints triggers the attraction group iterations, which then bring the number of blocks back down, close to the number of blocks in the original run.

Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
NC_AUC	0.94	1.29	1.06
OBP	1.00	1.01	1.02
2_A	0.99	1.05	1
4_A	0.99	1.06	1.03
16_A	0.99	1.09	1.02
2_H	1.00	1.01	1.06
4_H	1.00	1.00	1.03
16_H	1.00	1.02	1.07

Table 5.3: QoR results for the Titan benchmarks across different the rectangular constraints cases. All results are geometric means and normalized to the no constraints case.

The effects of placement constraints on wirelength appear to be slightly different across the three testing sets. When the VTR benchmarks are run on the K4 architecture, there is no increase in wirelength for any constraints case, as shown in Table 5.5. On the other hand, when the same benchmarks are run on the K6 architecture, there is a slight increase in wirelength in the most stringent constraints case - when all primitives are locked to tight regions in 16_A, there is a modest wirelength increase of 4%, as seen in Table 5.4. The other constraints cases remain relatively stable, with wirelength increasing by 2% at most.

The Titan results set shows a slightly higher increase in wirelength in some constraints cases, relative to the other two sets. However, it is still a relatively modest increase - the most stringent

Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
NC_AUC	0.96	1.02	1.02
OBP	1.00	1.00	1.01
2_A	1.00	1.01	1.00
4_A	0.99	1.02	1.01
16_A	0.96	1.04	1.01
2_H	1.00	0.99	1.00
4_H	1.00	1.01	1.01
16_H	1.00	1.02	1.01

Table 5.4: QoR results for the VTR benchmarks, mapped to the K6 architecture, across different the rectangular constraints cases. All results are geometric means and normalized to the no constraints case.

Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
NC_AUC	0.97	1.03	1.00
OBP	1.00	1.00	1.00
2_A	1.00	1.00	0.99
4_A	1.00	0.99	1.00
16_A	1.00	0.99	0.99
2_H	1.00	1.00	0.99
4_H	1.00	1.00	1.00
16_H	1.00	0.99	0.98

Table 5.5: QoR results for the VTR benchmarks, mapped to the K4 architecture, across different the rectangular constraints cases. All results are geometric means and normalized to the no constraints case.

case, 16_A, sees an increase of 9%. To understand why this is occurring, we will examine the data presented in Table 5.6, which presents the normalized wirelength result of each constraints run for each Titan benchmark. For each run, it also shows the number of packing iterations that were needed to cluster densely enough to fit the required amount of primitives inside each floorplan region. Any blank cell in the table indicates a run that resulted in a routing failure.

By examining 5.6, we observe that there is a clear correlation between >1 packing iterations and increased wirelength. As described in section 4.2.2, attraction groups are used from the second packing iteration onwards to pack the clusters more densely. Multiple packing iterations are performed on a needs-only basis - they are used when the clustering did not succeed in fitting all the primitives that must be in a certain region into the region. Attraction groups increase the logic utilization per cluster, as they select a higher number of molecule candidates per cluster by searching through the cluster's attraction groups.

The three constraints cases from Table 5.3 which experience the highest wirelength increases are 2_A (5%), 4_A (6%), and 16_A (9%). From Table 5.6, we can see that these constraints cases are also the most likely to require additional packing iterations. If we observe column 2_A in Table 5.6, we see that all of the runs which only required one packing iteration experience no increase in wirelength, or slight increases in wirelength. This trend is consistent across all of the constraints cases. In contrast, benchmark circuits such as stap_qrd and LU_Network, which both require 2 packing iterations in the 2_A constraints case show significant increases in wirelength of 44% and 38% respectively. Furthermore, a few constraints cases which required multiple packing iterations resulted in routing failures. These cases of increased wirelength and routability issues due to denser packing are consistent with the findings of [62] that denser packing negatively impacts wirelength and routability. No routing failures were observed when running the VTR benchmarks with placement constraints. All of the constraints cases ran to completion for all of the VTR benchmarks that were tested.

It is interesting to note that the number of packing iterations affects some constraints cases more severely than others. For example, for the stap_qrd benchmark, the 2_A case requires two packing iterations while the 16_A case requires five. However, the 2_A case shows a 44% increase in wirelength while the 16_A case shows an 18% increase. This general trend of the wirelength being more negatively impacted in the constraints cases with larger regions can be explained by the relatively large size of the attraction groups while packing. When running with attraction groups in a 2_A case, all of the primitives from the left half of the grid, for example, would be in the same attraction group during clustering. This increases the chances of a completely unrelated molecule being packed in a cluster. In contrast, in a 16_A case, there would be less molecules in each attraction group, as each group would only include the molecules that are constrained to the same sixteenth of the grid.

We can see from Table 5.6 that less rigid constraints cases (OBP, 2_H, 4_H, and 16_H) typically do not require multiple packing iterations. Correspondingly, the wirelength generally does not increase in these cases.

The critical path delay is found to remain generally the same for the VTR benchmarks in the case of both architectures, as seen in Tables 5.4 and 5.5. The Titan results set shows a modest increase in critical path delay in some of the constraints cases - most notably in Table 5.3, the 2_H and 16_H show an increase in critical path delay of 6% and 7% respectively. This increase may be a result of poor clustering decisions due to the mix of constrained and unconstrained blocks. This mix may have more significant implications for some hard blocks like RAMs that are more commonly used in the Titan benchmark set. This may be why we see a slight degradation in critical path delay for the Titan benchmark set, whereas there is no degradation showcased by the VTR benchmarks. Additionally, the Titan benchmarks are run with a Stratix-IV-like architecture, which is a more complex architecture than the K6 and K4 architectures that the VTR benchmarks were run with. This complexity may also explain why the Titan set sees a slight degradation in critical path delay, when the VTR benchmarks do not.

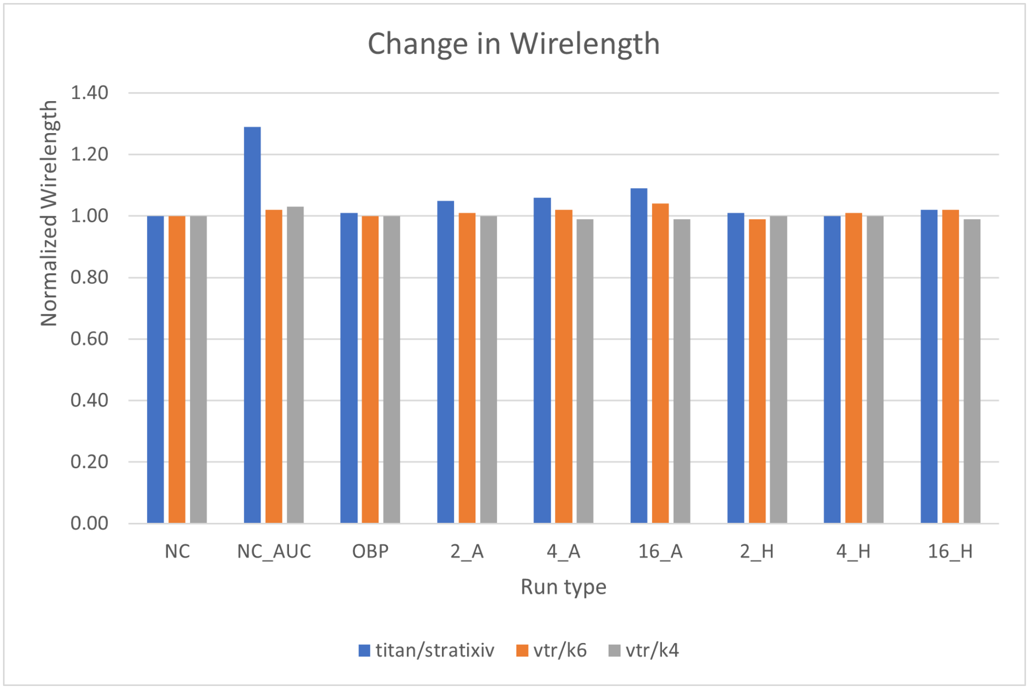


Figure 5.3: Normalized values of wirelength across different constraints cases for three sets of results.

.

Figures 5.3 and 5.4 show the normalized values of wirelength and critical path delay across all the different constraints cases for each results set. Overall, from this section we have observed that the QoR of the results remained relatively close to the baseline values across all of the constraints cases.

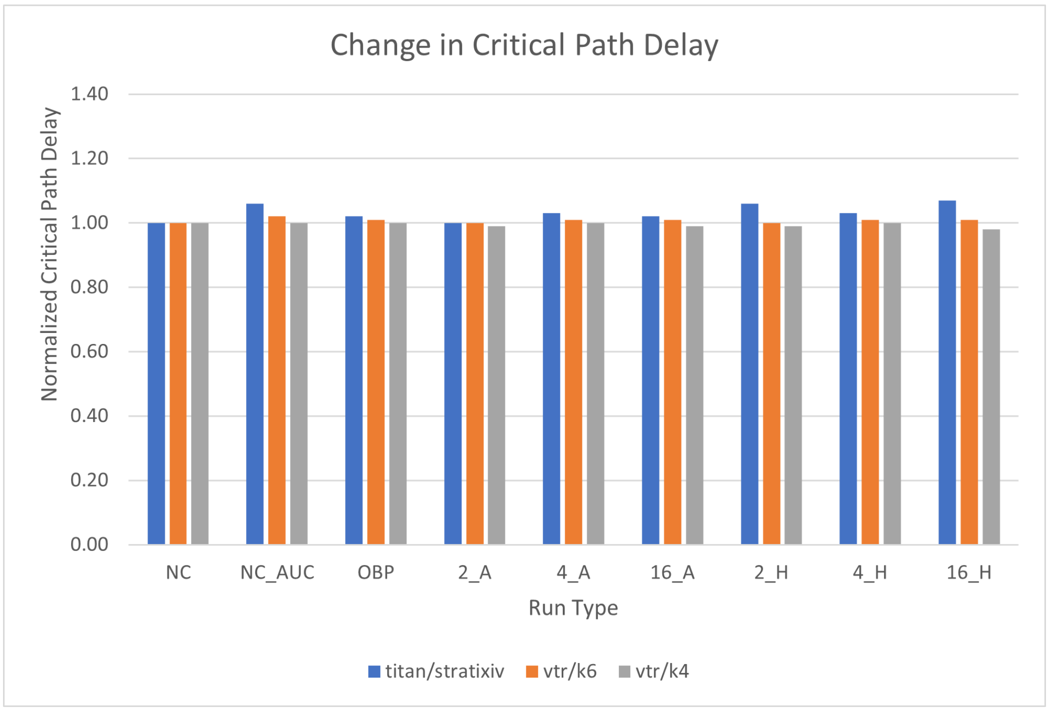


Figure 5.4: Normalized values of critical path delay across different constraints cases for three sets of results.

.

Benchmark	NC	NC_AUC	OBP		2_A		4_A		16_A		2_H		4_H		16_H	
	WL	WL	WL	Pack Its.	WL	Pack Its.	WL	Pack Its.	WL	Pack Its.	WL	Pack Its.	WL	Pack Its.	WL	Pack Its.
segmentation	1.00	1.36	1.03	1	1.04	1	1.04	1	1.04	2	1.04	1	1.03	1	1.05	1
LU230	1.00		1.00	1	0.95	1	0.97	1	0.97	1	0.99	1	1.01	1	1.02	1
des90	1.00	1.41	0.97	1	0.98	1	0.98	1	0.96	1	1.01	1	0.97	1	0.98	1
neuron	1.00	1.16	1.02	1	1.06	1	1.00	1	1.03	2	1.06	1	0.99	1	1.03	1
direct_rf	1.00		1.00	1	0.98	1	0.96	1	1.10	3	0.96	1	0.95	1	0.94	1
bitcoin_miner	1.00		1.00	1	1.04	1	1.09	1	1.01	1	1.08	1	1.06	1	1.04	1
dart	1.00	1.15	1.00	1	0.96	1	0.97	1	1.06	2	0.99	1	0.95	1	0.98	1
minres	1.00	1.11	1.00	1	0.97	1	1.00	1	0.98	1	0.97	1	0.99	1	0.99	1
openCV	1.00	1.33	1.05	1	0.99	1	0.97	1	0.99	1	1.00	1	0.98	1	1.00	1
sparcT1_chip2	1.00		1.01	1	1.00	1	0.98	1	1.12	2	1.01	1	1.01	1	1.02	1
LU_Network	1.00	1.32	1.00	1	1.38	2	1.47	2	1.18	2	1.00	1	1.01	1	1.01	1
sparcT1_core	1.00	1.10	1.01	1		2		5	1.29	5	1.02	1	1.05	1	1.30	1
stereovision	1.00	1.25	1.00	1	1.01	1	1.00	1	1.01	1	0.99	1	1.03	1	1.01	1
cholesky_mc	1.00	1.11	1.00	1	1.01	1	0.99	1	1.14	5	1.00	1	1.01	1	1.02	1
gsm	1.00	1.48	1.00	1	1.02	1	0.99	1	1.03	2	1.01	1	1.00	1	0.99	1
mes_noc	1.00	1.56	1.00	1		2	1.60	2	1.31	3	0.98	1	1.01	1	1.03	1
denoise	1.00	1.47	1.02	1	1.01	1	0.99	1	1.18	4	1.01	1	0.99	1	1.02	1
sparcT2_core	1.00		1.00	1		2		2	1.20	2	1.04	1		4		4
cholesky_bdti	1.00	1.12	1.00	1	0.99	1	0.98	1	1.17	5	1.01	1	1.05	1	1.01	1
bitonic_mesh	1.00	1.49	1.00	1	0.96	1	0.99	1	0.99	1	0.95	1	0.98	1	1.01	1
SLAM_spheric	1.00	1.29	1.00	1	1.26	2	1.22	2	1.12	4	1.07	1	1.05	1	1.05	1
stap_qrd	1.00	1.30	1.00	1	1.44	2	1.28	2	1.18	5	0.98	1	0.96	1	0.99	1
Geometric Mean	1.00	1.29	1.01	1.00	1.05	1.21	1.06	1.26	1.09	2.09	1.01	1.00	1.00	1.07	1.02	1.07

Table 5.6: Wirelength results for each of the rectangular constraints cases, for each titan benchmark, normalized to the no constraints case. The number of packing iterations needed for each constraints run is included. Cells that are blank were routing failures.

5.2.2 Run-times and Memory Usage

The results show that, as would be expected, pack time increases for constraints runs that require multiple iterations. As discussed in Chapter 4, multiple packing iterations occur when there are too many clusters in a floorplan regions, leading to retries with attraction groups. This can be observed from Table 5.7, which shows the normalized run-time and memory values for the Titan results set across the different constraints. The three constraints cases which most typically require multiple pack iterations - 2_A, 4_A, and 16_A - experience pack time increases. The same phenomenon can be seen in the K6 architecture and VTR benchmark set, shown in table 5.8. For the K4 architecture, multiple packing iterations were not needed, which is why we do not see any pack time increase in Table 5.9.

Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
NC_AUC	1.31	0.97	2.37	1.23	0.99
OBP	0.94	0.98	0.99	0.98	1.02
2_A	1.12	1.11	1.16	1.07	1.02
4_A	1.16	1.32	1.14	1.18	1.02
16_A	1.87	1.39	1.16	1.43	1.02
2_H	0.93	1.36	1.04	1.14	1.01
4_H	0.99	1.46	1.03	1.2	1.01
16_H	0.98	1.59	1.14	1.3	1.01

Table 5.7: Run-time and memory results for the Titan benchmarks with the Stratix-IV-like architecture across different the rectangular constraints cases. All results are geometric means and normalized to the no constraints case.

Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
NC_AUC	1.02	0.95	1.04	0.99	0.99
OBP	1.00	1.04	1.02	1.01	1.06
2_A	0.96	1.15	1.04	1.03	1.06
4_A	1.11	1.32	1.04	1.16	1.06
16_A	2.50	1.33	1.06	1.86	1.07
2_H	0.91	1.23	1.03	1.03	1.04
4_H	0.99	1.33	1.03	1.11	1.04
16_H	1.11	1.39	1.06	1.21	1.04

Table 5.8: Run-time and memory results for the VTR benchmarks, mapped onto the K6 architecture, across different the rectangular constraints cases. All results are geometric means and normalized to the no constraints case.

All three results sets experience increases in place time across all constraint cases, with the exception of the OBP case. The reason for the increase is the same for the different types of constraints files, and is as follows.

During placement, new placement solutions are generated by swapping the locations of different blocks and evaluating the change in cost. During this process, any proposed swap that moves a cluster block outside of its constraints region must be aborted. Many swaps attempt to move blocks outside of their placement constraint region, so many wasted moves are created and aborted, leading

Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
NC_AUC	1.17	1.00	1.10	1.01	0.99
OBP	0.93	1.04	1.00	1.01	1.05
2_A	0.92	1.11	1.02	1.04	1.05
4_A	0.92	1.37	1.04	1.16	1.05
16_A	0.91	1.35	1.06	1.15	1.05
2_H	0.93	1.33	1.03	1.14	1.05
4_H	0.92	1.45	1.06	1.20	1.04
16_H	0.92	1.57	1.07	1.26	1.04

Table 5.9: Run-time and memory results for the VTR benchmarks, mapped onto the K4 architecture, across different the rectangular constraints cases. All results are geometric means and normalized to the no constraints case.

to an increase in placement time. This is also the reason why constraints cases with relatively larger constraint regions (ex. 2_A) experience a smaller place time increase than constraints cases with relatively tighter regions (ex. 16_A), as observed in all three results sets in Tables 5.7, 5.9, and 5.8. The larger the region, the less likely it is that block will be swapped out of it, thus leading to less aborted moves.

In order to mitigate this place time increase, we implemented an optimization to reduce the number of illegal (with respect to placement constraints) moves created during placement. Each time a swap is created, a range limit is set to specify the area where the "from" block can be moved. To improve place time, whenever a potential swap is being created for a constrained block, the range limit is intersected with the block's constraint region. There are many move generators in VPR, but all begin by choosing a "from" block and choosing a target region to which it should be moved [18]. We intersect this target region with the region constraint (if any) of the block. Implementing this optimization significantly mitigated the increase in place time. Tables 5.10 and 5.11 shows the place times in seconds of 5 Titan benchmarks across 5 different constraint cases, before and after the constraint-aware placement moves were implemented. Figure 5.5 shows the difference in place time across the different circuits for the 16_A case. The decrease in place time from the optimization is significant. From tables 5.10 and 5.11, we can see that the place time increase went from 42% to 5% for the 2_A case, from 72% to 27% for the 4_A case, and 70% to 27% for the 16_A case.

Even with the constraint-aware placement moves implemented, however, some increase in place time still occurs. When a swap occurs during place moves, two (or more, in the case of macros) blocks are involved. We intersect the target region of the first block with its constraint region, to remove the possibility of it being moved out of its constraint region. However, the move may result in the block it is being *swapped with* moving out of *its* constraint region, resulting in an aborted move. We are not able to see the proposed location of the second block until the swap is nearly completed, so we cannot fix the move and must abort it, wasting CPU time.

Route time shows slight increases across the different results sets. Some of these increases can be attributed to the modest increases in wirelength discussed earlier in Section 5.2.1. In Table 5.7, the route time increases most for constraints cases 2_A, 4_A, and 16_A, which experienced wirelength increases of 5%, 6%, and 9% respectively.

The total run-time increased across every results set for nearly every constraints case, with the

Before Constraint-Aware Placement Moves					
Benchmark	NC	OBP	2_A	4_A	16_A
neuron	74.9	77.5	112.78	144.98	139.33
des90	336.1	258.62	521.1	506.29	520.39
minres	489.47	499.04	718.25	946.43	937.79
dart	456.91	361.75	604.16	768.68	736.73
openCV	534.62	432.2	689.35	857.21	853.16
Geometric mean	313.1197	274.6872	445.6605	539.6669	532.3087
normalized	1.00	0.88	1.42	1.72	1.70

Table 5.10: The place time results of five Titan benchmarks across five different constraints cases with conventional placement moves. The results shown are in seconds.

After Constraint-Aware Placement Moves					
Benchmark	NC	OBP	2_A	4_A	16_A
neuron	74.9	79.74	82.91	107.27	107.98
des90	336.1	247.21	390.94	354.5	391.34
minres	489.47	467.29	508.57	724.58	680.09
dart	456.91	370.73	403.6	549.64	555.84
openCV	534.62	484.8	575.1	669.56	626.63
	313.1197	277.8368	328.5129	399.2177	398.1851
normalized	1.00	0.89	1.05	1.27	1.27

Table 5.11: The place time results of five Titan benchmarks across five different constraints cases, with the constraint-aware placement move generator. The results shown are in seconds.

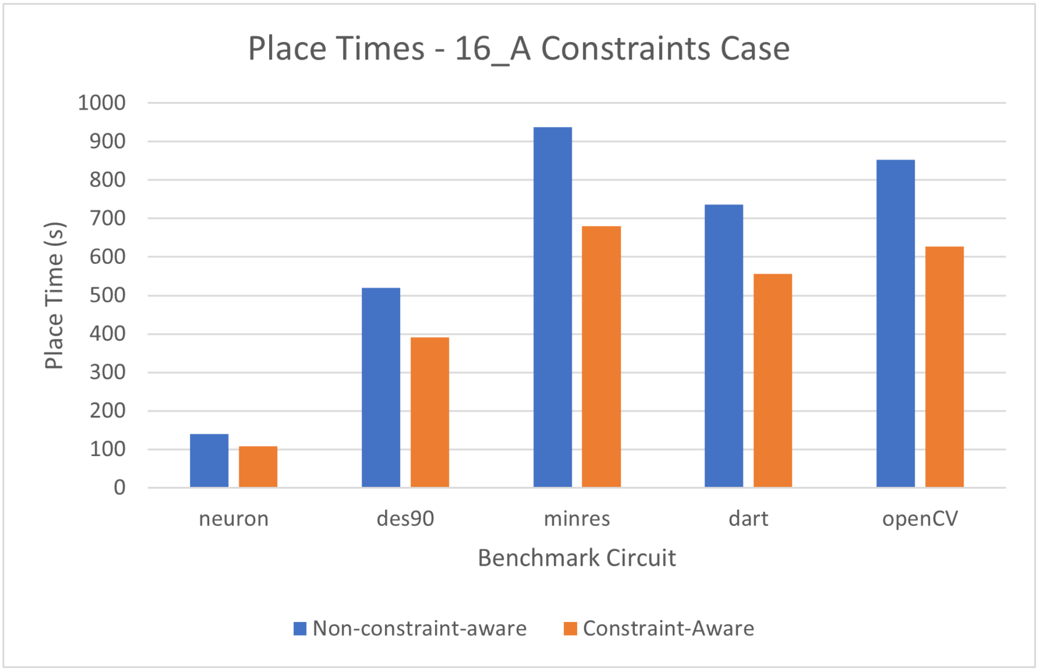


Figure 5.5: Place times before and after the constraint-aware placement moves optimization for the 16_A constraints case across five Titan benchmarks.

.

exception of OBP. The largest increase of 86% occurred for the 16_A case in the VTR/K6 results set. This case required multiple packing iterations for many circuits (pack time is 2.5x for this constraints case), and was also a constraints case with many tight regions. For these reasons, this increase was exceptionally high - the next biggest increase was 43%. Therefore, for all constraints cases across the three results sets, the total run-time increase was always significantly less than 1.5x the no constraints case, with one exception.

The total run-time increased primarily as a result of the place time increases discussed earlier in this section. The route time increases were relatively insignificant and since we were running the tests with a fixed channel width, the route time was generally the shortest portion of the flow. The pack times could also contribute to the increased overall run-times, in cases where multiple packing iterations were needed.

The changes in memory were insignificant for the Titan benchmarks, as can be seen in Table 5.7. For the VTR benchmarks, slight increases can be observed across the constraints cases in tables 5.9 and 5.8.

Figures 5.6 and 5.7 shows the changes in place time and total run-time across all three results sets.

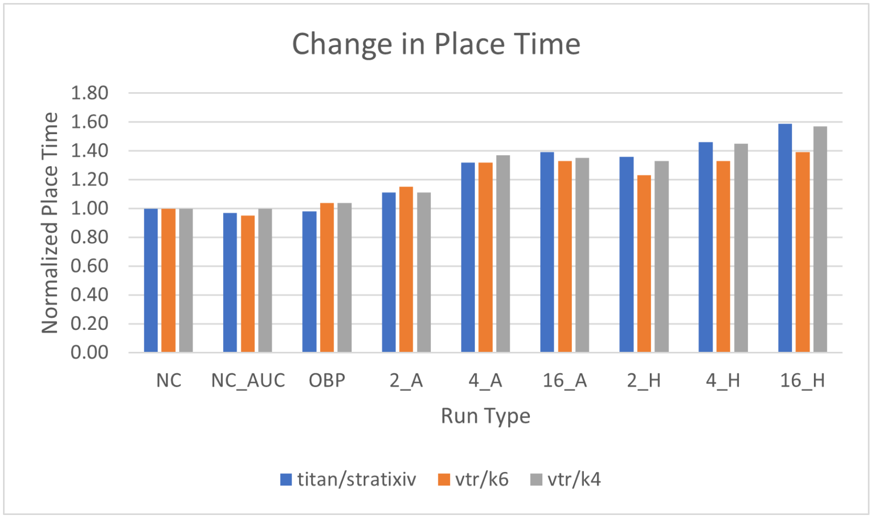


Figure 5.6: Normalized values of place times across different constraints cases for three sets of results.

.

5.3 One Location Constraints

For the one location constraints, we constrained every primitive to an exact grid location. We found that VPR was generally only able to adhere to these constraints when mapping the VTR benchmarks to the K4 architecture. The one location constraints do not work for the VTR/K6 runs and the Titan runs because the packing algorithm is not able to exactly reproduce every cluster with the primitives that must be packed in it on architectures with fracturable LUTs. Some clusters are not able to reproduce because of the different ways primitives can be organized internally. The order in which primitives are packed in the cluster may change, and some primitives may not be able to be paired in the 5-LUT mode in the same way that they were in the original clustering. This results

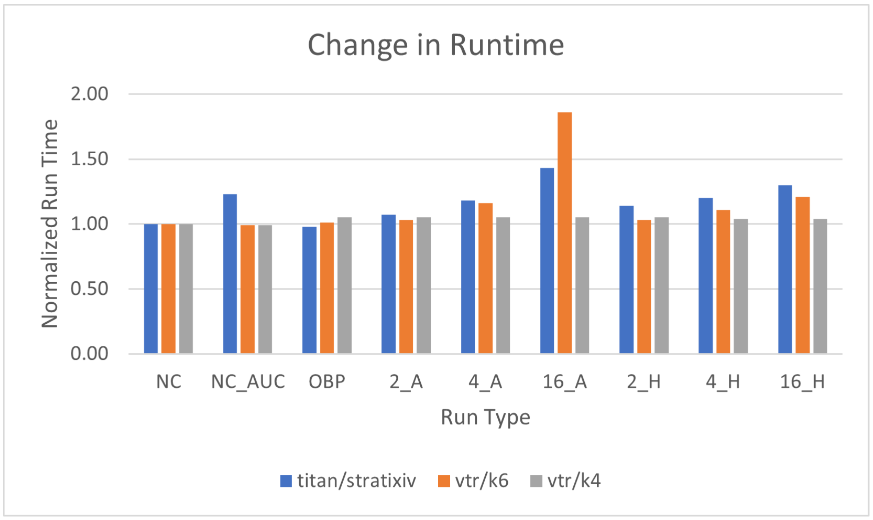


Figure 5.7: Normalized values of run-times across different constraints cases for three sets of results.

.

in primitives that are constrained to one location being packed into two different clusters. The two clusters cannot then be placed at the same location, so this results in a placement consistency error.

Table 5.13 shows the QoR results for the VTR/K4 set when run with the one location constraints case. As expected, there is no change in QoR, since the primitives are locked to the exact locations that they were placed in during the no constraints run.

As shown in Table 5.14, the overall run decreases by 41% because of a drastic reduction in place time. As described in section 4.2, blocks that are constrained to a region of size one (fits one location on the grid) are automatically locked to that spot - they do not need to be placed during initial placement, nor are they moved during place moves. For this reason, the place time becomes negligible with respect to the rest of the CAD flow, as can be seen by the place times shown in Table 5.12.

Place Time (s)		
Benchmark	NC	All Blocks Locked to One Location
mcml	388.89	3.02
LU32PEEng	559.01	4.89
arm_core	24.77	0.38
or1200	4.85	0.1
LU8PEEng	64.58	0.79
stereovision1	14.31	0.23
bgm	81.13	0.92
sha	2.17	0.06
spree	0.89	0.03
blob_merge	9.59	0.2
Geometric mean	22.69604	0.357153134
normalized place time		0.02

Table 5.12: Place time results of the VTR benchmarks on the K4 architecture.

Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
One_Spot	1.00	1.00	1.00

Table 5.13: QoR results of the VTR benchmarks on the K4 architecture, for the one location constraints case.

Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
One_Spot	0.85	0.02	0.99	0.59	1.05

Table 5.14: Run-time and Memory results of the VTR benchmarks on the K4 architecture, for the one location constraints case.

5.4 L & T-shaped Constraints

The L and T constraints are meant to check that the packing and placement algorithms can handle partitions with multiple rectangular regions. The L and T-shaped constraints files were created by randomly choosing the location for and creating one L-shaped and one T-shaped partition per constraints file. Each partition would be made up of two rectangular regions. From a previous placement, we observed which primitives were placed in the designated L-shaped or T-shaped regions - these primitives were then constrained to the respective region that they were previously placed in. Any primitives outside of these regions in a previous placement remained unconstrained. An example of how the L-shape and T-shape partition regions would look conceptually is shown in Figure 5.8.

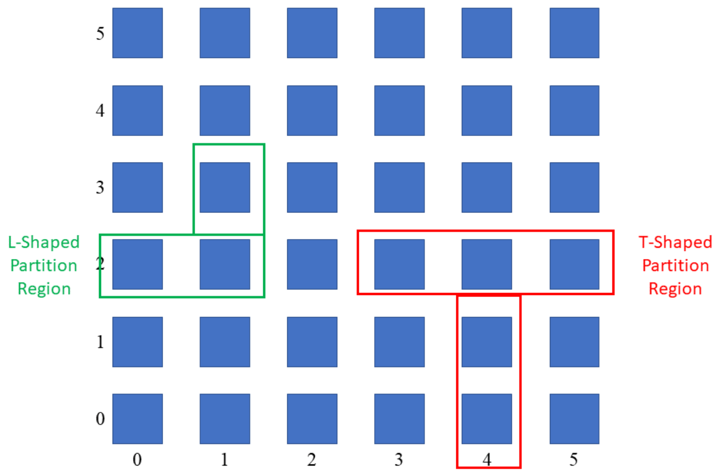


Figure 5.8: An example of L-shaped and T-shaped partition regions

.

Due to the time-consuming nature of manually creating these constraints, L and T constraints files were created for fewer benchmarks than the automatically generated rectangular constraints cases. They were created for the bgm, blob_merge, LU32PEEng, or1200, and stereovision1 VTR benchmarks. For the Titan benchmarks, they were created for dart, des90, minres, openCV, and neuron. The results shown in the tables for this section are averaged across these specified benchmarks.

As can be seen in Tables 5.16, 5.17, and 5.15, the QoR results for these cases follow trends similar to those discussed in section 5.2.1. The effects on run times and memory usage are also similar to the rectangular constraints cases, as shown in tables 5.18, 5.19, and 5.20.

Titan			
Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
L_and_T	1.00	1.04	1.04

Table 5.15: QoR results of the Titan benchmarks, for constraints with L and T-shaped regions.

K6			
Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
L_and_T	1.00	1.03	1.04

Table 5.16: QoR results of the VTR benchmarks on the K6 architecture, for constraints with L and T-shaped regions.

K4			
Run Type	Number of Post-Packed Blocks	Wirelength	Critical Path Delay
NC	1.00	1.00	1.00
L_and_T	1.01	1.02	1.00

Table 5.17: QoR results of the VTR benchmarks with the K4 architecture, for constraints with L and T-shaped regions.

5.5 Summary

In this chapter, we saw that VPR is able to handle a wide range of placement constraints cases while maintaining reasonable QoR results. There is some increase in run time in more stringent cases, due to the increase in placement time, in particular. Full result tables detailing the the number of post-packed blocks, wirelength, critical path delay, run times, and memory usage when running with the rectangular constraints cases from Section 5.2 can be found in Appendices A, B, and C of this thesis. While the constraints work well in most cases, there is still room for improvement in very detailed constraints on complex logic architectures; we will discuss some possible future enhancements in the next chapter.

Titan					
Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
L_and_T	0.91	1.60	0.97	1.11	0.99

Table 5.18: QoR results of the Titan benchmarks, for constraints with L and T-shaped regions.

K6					
Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
L_and_T	0.98	1.46	1.04	1.13	1.00

Table 5.19: QoR results of the VTR benchmarks with the K6 architecture, for constraints with L and T-shaped regions.

K4					
Run Type	Pack Time	Place Time	Route Time	Total Run-time	Memory
NC	1.00	1.00	1.00	1.00	1.00
L_and_T	0.94	1.63	1.06	1.29	1.00

Table 5.20: QoR results of the VTR benchmarks with the K4 architecture, for constraints with L and T-shaped regions.

Chapter 6

Conclusions and Future Work

6.1 Summary

As FPGAs continue to grow in capacity and capability, FPGA CAD flows will need to evolve to facilitate designer productivity. CAD flows must allow designers to exploit the full potential of the FPGA device while keeping run times from becoming prohibitively long. Several techniques have been used to improve the run times of FPGA compilation, including partial reconfiguration and incremental synthesis. Also, modular, team-based design will be an important tool for handling large designs. These flows often involve running the CAD flow with placement constraints, as the general idea is to turn the compilation of an FPGA design into several sub-problems of a smaller size. In this thesis, we presented the work that was done to enhance an open-source FPGA CAD tool, VPR, to ensure that it could follow placement constraints while maintaining good quality of results.

In this thesis, we defined a flexible format for VPR constraints, and enhanced the packing and placement algorithms to be floorplan-aware. We ran the VPR CAD flow with several kinds of placement constraints, from coarse to fine, to ensure that packing and placement could be performed while adhering to user-defined placement constraints. We demonstrated that VPR can respect the given constraints during packing and placement for all our test cases except one - the constraints where all primitives are locked to one location on the grid.

The one location constraints are adhered to very well on the K4 architecture, and drastically reduce placement time on all of the benchmark circuits when running with this architecture. These constraints cannot be adhered to when running the K6 and Stratix IV-like architectures - this is an area of future work which will be discussed in the next section.

All of the other rectangular, L-shaped, and T-shaped constraints work well with minimal impact on result quality. Three results sets were used for gathering results - the VTR benchmarks run on the 4-LUT based K4 architecture, the VTR benchmarks run on the 6-LUT based K6 architecture, and the titan benchmarks on the Stratix IV-like architecture. Across those three sets, in the most stringent constraints case - where all blocks are constrained to one of sixteen regions - the geometric mean for increase in wirelength was 9% and 4% for the titan runs and the VTR/K6 runs respectively, and decreased by 1% in the VTR/K4 runs. Furthermore, for the other constraints cases that were tested, the increase in wirelength was 5% or less for the titan runs, and 2% or less for the VTR runs on both architectures that were used. Critical path delay showed small increases for the titan benchmarks, and no increases for the VTR benchmarks. The number of post-packed blocks never increased across any of the three results sets.

As for impacts on run times, it was found that the place time was the most affected VPR stage. Across the three results sets and different rectangular constraints cases that were tested, the place time increase was in the range of 4% to 59%. The overall run time was under 1.5x for all constraints cases across all results sets, except for the 16_A constraints cases on the VTR/K6 results set. The 16_A case constraints cases on the VTR/K6 results set experienced a runtime increase of 86%, mainly due to the multiple packing iterations that were needed to achieve dense clustering for these runs.

6.2 Future Work

The following enhancements can be made to VPR to further facilitate CAD flows that utilize placement constraints.

Firstly, VPR can be made to handle constraints cases where all primitives are locked to one location on the grid. The reason that VPR could not perform packing and placement with these constraints was that it could not reproduce all of the clusters exactly, and so some groups of primitives that were constrained to one spot were packed into two clusters. There are two methods to facilitate these flows. The first method is to add another dimension to the region location in the VPR constraints file. This dimension could specify where in the cluster the primitive should be placed. The second method would be to alter the packing flow such that clusters could be taken apart and re-packed during packing. Currently, once a cluster is packed in VPR, that formation of the cluster is accepted, and no optimization is attempted for the internal organization of the cluster. This second method would address the issue of clusters not being reproduced exactly, because during packing, the clustering process could try to take apart and pack each cluster until each required primitive has fit into it. The first approach may be better than the second approach, as the second approach may lead to an increase in packing time, since clusters would need to be taken apart and re-packed multiple times.

To better serve modular flows, VPR's routing algorithm could also be enhanced to respect region constraints. This would be helpful during modular flows when signals between modules must be matched up.

Another helpful feature would be to add graphics for the placement constraints in VPR. VPR currently has graphics to display the placements of circuits and FPGA architectures. Graphics for placement constraints could take the form of outlining the boundaries of each partition, and colour-coding which primitives belong to which partition.

Lastly, logic synthesis flows that are commonly used with VPR could be updated to have better, more hierarchical naming systems for their primitives. This would be very helpful for floorplanning. For example, if a designer wanted to constrain all of the primitives belonging to a certain module to the same partition, it would helpful for all of the primitive names to include the name of the module in question. Currently, logic synthesis flows that are used in VTR like ODIN II and ABC produce arbitrary names for some of the internal output signals, which results in primitive names that are not conducive to floorplanning, as discussed in section 3.3 of this thesis.

Appendix A

VTR Benchmarks, K4 Architecture Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	13623	13333	13623	13623	13623	13623	13623	13623	13623
LU32	16283	15784	16283	16283	16283	16283	16283	16283	16283
arm_core	2528	2465	2528	2528	2528	2528	2528	2528	2528
or1200	1321	1291	1321	1321	1321	1321	1321	1321	1321
Lu8	4767	4601	4767	4767	4767	4767	4767	4767	4767
stereovision1	2029	1995	2029	2029	2029	2029	2029	2029	2029
bgm	5818	5531	5818	5818	5818	5818	5818	5818	5818
sha	489	482	489	489	489	489	489	489	489
spree	227	222	227	227	227	227	227	227	227
blob_merge	1456	1365	1456	1456	1456	1456	1456	1456	1456

Table A.1: Number of Post-Packed Blocks Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	24.88	35.54	21.42	21.61	21.73	21.89	21.78	21.84	21.63
LU32	28.06	42.05	24.25	23.82	23.48	24.72	24.82	24.61	24.55
arm_core	2.59	3.01	2.5	2.49	2.49	2.52	2.53	2.51	2.52
or1200	0.62	0.67	0.6	0.6	0.6	0.6	0.6	0.6	0.6
Lu8	6.66	7.98	6.21	6.19	5.83	6.29	6.31	6.29	6.2
stereovision1	2.18	2.3	2.08	2.05	2.02	2.08	2.08	2.05	2.07
bgm	7.04	8.7	6.21	6.15	6.02	6.39	6.25	6.3	6.28
sha	0.44	0.48	0.42	0.42	0.42	0.43	0.42	0.41	0.42
spree	0.2	0.2	0.19	0.2	0.19	0.19	0.19	0.19	0.18
blob_merge	1.14	1.22	1.02	1.02	1.02	1.03	1.02	1.02	1.02

Table A.2: Pack Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	388.89	392.8	400.9	592	579.07	373.01	582.09	515.57	659.1
LU32	559.01	558.95	596.14	608.91	772.22	583.45	819	813.98	1034.49
arm_core	24.77	23.87	27.28	33.21	30.66	26.34	32.43	33.92	38.34
or1200	4.85	4.97	6.9	9.06	7.64	5.21	6.77	8.91	8.01
Lu8	64.58	62.36	67.26	88.8	84.57	67.49	86.05	89.7	108.23
stereovision1	14.31	15.14	15.41	19.83	17.79	14.87	15.94	19.61	18.5
bgm	81.13	80.78	94.67	121.1	117.12	85.24	121.26	160.39	171.46
sha	2.17	2.26	2.3	2.65	2.99	2.23	2.75	2.63	3.09
spree	0.89	0.84	1.01	1.05	1.1	0.94	1.02	1	0.95
blob_merge	9.59	9.45	10.36	13.17	12.03	9.87	13.18	15.64	15.56

Table A.3: Place Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	1287390	1382996	1288352	1258299	1249238	1287390	1268035	1283782	1267301
LU32	2220328	2452425	2235643	2180421	2195878	2220328	2244108	2195645	2196777
arm_core	313089	319951	315575	314380	315061	313089	314551	314487	315392
or1200	65917	66173	66112	66376	66706	65917	66548	66275	66452
Lu8	521507	546173	517277	516562	527150	521507	518210	518850	524305
stereovision1	189408	178649	181650	183276	181939	189408	182838	183004	184353
bgm	634379	664266	653763	627071	629075	634379	656282	642842	633162
sha	43199	42942	42651	42285	42505	43199	42421	42527	42474
spree	17066	17368	17175	17293	17154	17066	16885	17381	16889
blob_merge	135487	138928	134799	134473	135403	135487	134246	135427	135580

Table A.4: Wirelength Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	70.05	75.35	65.23	64.19	65.58	70.57	69.12	62.17	64.29
LU32	134.96	225.41	139.55	138.8	145.26	144.43	158.63	148.17	141.35
arm_core	12.45	18.85	13.23	17.44	18.59	12.49	14.36	19.01	19.3
or1200	1.81	1.79	1.7	1.76	1.77	1.66	1.89	1.86	1.78
Lu8	33.6	35.52	34.91	34.11	37.32	33.28	31.52	32.76	36.24
stereovision1	9.18	8.45	10.01	9.21	9.81	8.99	7.39	9.26	11.16
bgm	24.38	24.09	27.56	27.06	26.11	24.56	29.32	29.09	26.64
sha	1.13	1.14	1.12	1.15	1.14	1.14	1.2	1.15	1.14
spree	0.7	0.69	0.74	0.69	0.68	0.69	0.72	0.67	0.68
blob_merge	3.41	3.55	3.33	3.36	3.34	3.38	3.27	3.42	3.31

Table A.5: Route Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	61.3788	59.4583	61.8323	61.802	61.0824	61.3788	62.1042	63.5739	61.253
LU32	75.1765	75.1683	73.6378	72.9732	73.1803	75.1765	73.1028	74.2039	73.0003
arm_core	15.6579	15.4795	15.8076	16.0482	16.1657	15.6579	15.8101	15.7045	15.9867
or1200	13.4219	13.882	13.7846	13.4527	13.3983	13.4219	13.4495	13.7494	13.5286
Lu8	71.975	73.2425	72.5327	74.0494	73.241	71.975	73.2999	73.2892	72.226
stereovision1	4.87901	4.83332	4.48015	4.83042	4.48175	4.87901	4.56624	4.57443	4.45444
bgm	15.0366	14.7621	14.4052	14.8555	14.6947	15.0366	13.8999	14.4556	14.5886
sha	11.3773	11.8681	11.2782	11.292	11.5325	11.3773	11.5005	11.5448	11.0598
spree	9.74468	9.99619	9.75641	9.85896	9.53193	9.74468	9.90323	9.66744	9.55584
blob_merge	9.95356	9.59308	9.71462	9.81806	9.6552	9.95356	9.66486	9.84145	9.61766

Table A.6: Critical Path Delay Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	739.08	769.79	740.51	946.12	929.15	716.68	933.07	851.73	1001.26
LU32	974.51	1080.66	1015.65	1027.26	1188.11	1007.94	1253.51	1238.73	1453
arm_core	65.01	70.82	68.27	78.41	77.06	66.71	74.73	80.63	85.34
or1200	11.65	11.7	13.49	15.72	14.33	11.76	13.57	15.64	14.69
Lu8	162.43	163.36	165.91	185.9	185.99	164.62	180.64	186.57	208.19
stereovision1	53.33	46.79	55.4	58.81	57.21	53.49	53.13	58.63	59.63
bgm	163.66	162.44	179.18	204.8	200.05	167.17	207.77	247.6	255.19
sha	7.1	7.15	7.16	7.56	7.86	7.12	7.69	7.52	7.96
spree	5.29	5.18	5.4	5.48	5.51	5.31	5.4	5.3	5.29
blob_merge	25.04	24.53	25.45	28.24	27.07	25.04	28.19	30.81	30.65

Table A.7: Total Run-time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	2258.4	2261.9	2334.3	2327.3	2308.8	2333.1	2339.6	2321.1	2319.7
LU32	2370.6	2368	2444.6	2444.1	2443.3	2444.8	2437.7	2435.1	2435.3
arm_core	377.1	377.1	403.3	404.2	403.8	403.4	401.8	402	402.2
or1200	104.1	102.6	110.1	110.1	109.9	109.7	108.1	108.3	108.5
Lu8	674.4	672.8	706.4	712.4	718.4	719.3	715	714.5	713.9
stereovision1	361.8	334.9	380.8	381	379.8	378.9	380.8	379.6	379.3
bgm	697.7	685.6	747	738.1	746.6	751	744.7	748.7	752.2
sha	86.2	87.3	88.3	88.1	90.7	88.2	89.6	89.1	89.3
spree	65.3	65.4	66.5	67.3	65.3	66.5	66.5	65.6	65.6
blob_merge	204.6	201.3	214.5	215.5	215.4	216	215.7	216.2	216.3

Table A.8: Memory Usage Results

Appendix B

VTR Benchmarks, K6 Architecture Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	7619	6747	7634	7160	7037	7619	7621	7622	7658
LU32	7911	7712	7638	7656	7369	7911	7909	7850	7857
arm_core	1349	1303	1350	1361	1284	1349	1343	1349	1344
or1200	1040	1014	1044	1046	1015	1040	1044	1046	1047
Lu8	2462	2391	2474	2484	2399	2462	2463	2462	2471
stereovision1	975	950	975	975	975	975	975	975	975
bgm	3041	2909	3041	3007	2977	3041	3041	3038	3042
sha	230	225	230	230	230	230	230	230	230
spree	146	144	146	147	146	146	146	147	146
blob_merge	752	685	752	737	683	752	752	753	753

Table B.1: Number of Post Packed Blocks Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	205.25	210.87	195.04	352.58	623.98	202.94	195.99	197.38	189.2
LU32	196.08	207.44	441.76	427.79	1232	198.16	192.14	784.42	751.83
arm_core	21.59	21.47	19.02	16.98	94.77	21.77	19.68	19.06	16.89
or1200	6.46	6.8	4.97	4.28	18.61	6.45	5.39	4.7	6.81
Lu8	58.44	58.69	53.87	51.11	208.93	57.92	55.44	53.69	51.01
stereovision1	7.28	7.31	6.86	6.2	5.63	7.28	6.98	6.32	5.99
bgm	53.11	54.38	50.15	152.84	144.6	52.84	50.15	51.2	46.97
sha	1.73	1.76	1.38	1.32	1.02	1.71	1.49	1.47	1.28
spree	2.09	2.06	1.59	1.28	5.08	2.06	1.67	1.28	5.18
blob_merge	13.1	13.27	12.14	19.29	48.17	13.24	12.78	12.02	11.13

Table B.2: Pack Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	339.75	281.1	371.46	360.29	431.23	332.25	363.59	455.38	398.33
LU32	246.17	234.24	261.56	307.88	307.43	237.99	306.52	321.97	365.9
arm_core	12.39	12.63	15.95	17.35	20.41	12.85	16.78	16.33	19.88
or1200	3.84	3.76	4.97	6.01	5.16	4.14	4.93	6.19	5.15
Lu8	36.28	37.21	40.77	47.82	46.73	39.09	48.83	50.93	57.18
stereovision1	6.7	6.24	7.18	8.99	10.08	7.26	8.95	8.69	9.74
bgm	42.9	43.59	49.92	63.41	60.58	45.15	49.92	63.52	77.27
sha	1.04	1.01	1.06	1.23	1.24	1.05	1.14	1.15	1.2
spree	0.51	0.48	0.61	0.74	0.59	0.52	0.6	0.58	0.56
blob_merge	4.56	4.11	5.62	5.53	6.19	5.2	5.82	6.16	6.22

Table B.3: Place Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	942357	1015121	946118	987921	970940	947792	932496	956882	958865
LU32	1330870	1450113	1601270	1517285	1487724	1346527	1349434	1469522	1486891
arm_core	185980	187532	189563	185021	191507	185980	184225	183834	185707
or1200	42869	43496	42047	42377	43743	42059	42376	43258	43438
Lu8	318215	317583	314496	315291	325597	311344	315044	315702	319147
stereovision1	117765	111880	109541	113549	114277	117765	111793	113812	115449
bgm	359490	398245	362382	373556	381121	359490	362382	362364	363859
sha	14795	14474	14831	14615	14754	14795	14437	14580	14773
spree	10343	10444	10336	10472	10783	10343	10377	10443	10761
blob_merge	61831	62555	61588	63205	66978	61831	62410	62797	62480

Table B.4: Wirelength Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	38.09	37.14	32.8	40.1	41.46	35.99	35.04	42.18	41.17
LU32	60.7	78.18	127.18	79.48	77.09	62.36	73.48	77.39	78.07
arm_core	4.02	6.52	4.5	6.66	6.68	4.02	6.55	6.44	6.74
or1200	0.79	0.77	0.69	0.74	0.77	0.87	0.74	0.79	0.82
Lu8	10.82	10.55	10.73	9.92	11.76	12.32	9.74	10.06	10.43
stereovision1	3.22	2.53	3.06	3.2	2.73	3.37	3.24	2.82	2.91
bgm	7.73	8.08	7.75	7.53	7.64	7.85	7.75	7.46	7.25
sha	0.22	0.23	0.22	0.22	0.24	0.22	0.23	0.21	0.23
spree	0.28	0.23	0.27	0.24	0.23	0.26	0.24	0.25	0.25
blob_merge	0.95	1.03	0.85	0.89	0.93	0.96	0.94	0.88	0.88

Table B.5: Route Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	42.7842	44.7767	43.15	43.7171	42.7125	42.7675	43.9186	43.1007	43.448
LU32	74.8194	74.7199	76.519	75.3974	75.0921	75.5549	75.3756	77.4801	75.9311
arm_core	18.8513	18.4524	18.8183	18.8479	19.4331	18.8513	18.7368	19.0122	19.1577
or1200	8.44718	8.90758	8.86426	9.03094	8.6432	8.95094	8.58878	8.58324	8.67988
Lu8	75.1163	75.022	76.4445	74.7363	76.6708	76.6046	75.8053	76.1555	75.7567
stereovision1	5.58626	6.19944	5.27431	5.4429	5.82403	5.58626	5.44068	5.43736	5.29962
bgm	19.8134	21.3173	20.6283	21.0699	20.4863	19.8134	20.6283	21.0793	21.2379
sha	9.87232	9.69072	9.63361	9.88833	9.56439	9.87232	9.70693	9.63877	9.85032
spree	11.5174	11.5602	11.3234	11.1618	11.1967	11.5174	10.9882	11.2839	11.2813
blob_merge	15.2605	14.8532	15.2882	15.6701	15.5335	15.2605	15.5031	15.0488	15.0771

Table B.6: Critical Path Delay Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	659.34	598.26	676.06	829.66	1174.71	647.83	669.06	771.24	704.88
LU32	579.31	594.16	906.79	890.94	1692.17	574.32	648.28	1259.38	1271.61
arm_core	47.44	49.42	48.94	50.5	131.07	48.28	52.61	51.25	53
or1200	14.4	14.57	13.96	14.45	28.04	14.96	14.49	15.18	16.1
Lu8	127.29	127.28	126.7	130.25	288.96	130.5	135.32	135.83	139.67
stereovision1	25.95	24.71	26.08	27.41	27.52	27.06	28.37	26.98	27.72
bgm	128.49	130.86	133.97	249.44	238.45	131.49	133.97	147.32	156.8
sha	4.35	4.31	3.98	4.12	3.84	4.3	4.18	4.17	4.03
spree	3.75	3.56	3.27	3.09	6.71	3.65	3.33	2.93	6.8
blob_merge	23.8	23.3	23.86	31.03	60.58	24.84	24.77	24.29	23.45

Table B.7: Total Run-time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
mcml	1875.4	1822.9	1970.3	1974.7	1997.1	2017.3	1952.9	1956.7	1926
LU32	1741.2	1734.4	1864.2	1861.3	1852	1875.4	1809	1803.4	1803.9
arm_core	298.4	294.5	318.7	318.2	318.8	318.8	310.6	309.5	311.5
or1200	105.4	104.2	111.5	109.3	116.3	111.9	109.2	108.7	109.4
Lu8	569.6	566.1	608.2	606.2	603.4	599.6	591.2	586.8	587.2
stereovision1	290.8	288.9	301.7	303.3	301.8	303.1	297	298.8	297.9
bgm	600.1	599.4	651	645.4	651.4	650	651	624.8	626.1
sha	64.5	64.4	69.3	69.9	69.2	70.1	68.9	69.1	67
spree	48.2	47.7	50.2	50.2	50	49.8	49.8	50.3	49.9
blob_merge	184.3	182.3	195.2	194.9	198.2	195.3	191.3	191.7	191.5

Table B.8: Memory Usage Results

Appendix C

Titan Benchmarks, Stratix IV Architecture Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	7874	7535	7881	7888	7874	7874	7877	7879	7889
LU230	22116	20970	22315	22813	23894	22116	22116	22116	22116
des90	5254	4833	5287	5277	5348	5254	5232	5232	5262
neuron	3425	3373	3425	3492	3558	3425	3422	3430	3424
direct_rf	64544	59279	64618	64806	61547	64544	64549	64538	64549
bitcoin_miner	34220	31566	34220	34220	34220	34220	34220	34220	34220
dart	7461	7162	7536	7682	7673	7461	7461	7461	7452
minres	9585	8950	9705	9750	9831	9585	9585	9585	9629
openCV	8391	7525	8446	8562	8668	8391	8391	8391	8397
sparcT1_chip2	36029	33611	36444	36856	35628	36029	36031	36116	36367
LU_Network	32694	30678	31930	31047	31624	32694	32780	32737	32771
sparcT1_core	4439	4149	3954	3918	4030	4439	4447	4451	4448
stereovision	3941	3667	3948	3944	3977	3941	3940	3940	3938
cholesky_mc	5546	5449	5601	5692	5397	5546	5545	5542	5538
gsm	23477	21044	23543	23821	23402	23477	23463	23466	23446
mes_noc	24573	22613	23124	22530	22187	24573	24578	24579	24592
denoise	15265	14617	15282	15298	14581	15265	15269	15268	15284
sparcT2_core	15436	14397	13982	14155	14733	15436	15493	14924	15449
cholesky_bdti	10574	10413	10742	10922	10203	10574	10582	10600	10600
bitonic_mesh	9107	8432	9109	9182	9249	9107	9102	9102	9113
SLAM_spheric	5882	5574	5488	5506	5596	5882	5900	5868	5874
stap_qrd	16677	16247	16232	16305	16207	16677	16677	16678	16673

Table C.1: Number of Post Packed Blocks Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	169.11	191.09	162.04	160.81	323.22	166.66	162.99	165.77	165.54
LU230	992.32	2221.68	740.41	705.38	677.58	743.85	730.02	727.23	709.15
des90	294.13	307.88	241.53	236.03	224.84	250.72	242.11	235.83	238.87
neuron	82.01	89.02	81.48	80.39	160.31	80.75	78.92	80.76	80.03
direct_rf	1045.31	1538.43	820.67	799.44	2724.32	847.74	848.66	806.06	844
bitcoin_miner	708.41	1454.11	655.15	661.29	665.01	697.6	667.29	697.42	678.16
dart	401.88	414.59	352.98	341.23	619.42	359.18	360.4	354.31	357.41
minres	351.68	490.09	325.98	325.05	319.59	329.18	324.09	331.39	326.63
openCV	369.35	399.83	311.74	308.78	294.49	311.88	301.55	302.27	300.46
sparcT1_chip2	1526.82	2110.95	1441.2	1411.86	2561.75	1444.76	1471.07	1476.67	1406.44
LU_Network	774.35	1611.23	1506.3	1565.62	1473.07	755.92	751.93	757.59	742.98
sparcT1_core	227.34	230.09	460.57	1176.52	1105.37	223.12	220.31	215.87	543.07
stereovision	66.36	69.27	64.21	65.21	64.21	64.41	65.19	64.81	65.13
cholesky_mc	111	134.33	110.43	109.45	557.64	110.22	109.78	110.15	110.38
gsm	682.48	1095.84	648.45	648.58	1186.26	670.82	661	643.26	606.42
mes_noc	1368.64	1610.83	2527	2566.7	3728.03	1321.44	1328.33	1313.64	1299.86
denoise	367.15	490.09	343.35	347.69	1347.36	354.27	356.91	353.61	349.01
sparcT2_core	685.83	814.62	1387.14	1324.29	1203.43	673.82	673.36	2575.15	1158.27
cholesky_bdti	264.32	338.84	260.9	264.25	1326.92	260.87	261.24	259.2	260.37
bitonic_mesh	464.93	649.95	434.07	431.65	415	442.78	451.12	445.41	438.77
SLAM_spheric	180.31	179.08	331.53	324.93	623.75	179.39	176	173.96	174.9
stap_qrd	250.06	367.4	524.31	510.04	1325.27	246.92	245.33	246.16	246.58

Table C.2: Pack Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	457.82	459	554.9	551.49	555.56	441.4	550.97	561.22	648.92
LU230	2453.39	2264.69	2538.86	3108.94	3335.31	2258.89	3249.6	3095.25	4201.59
des90	336.1	268.97	390.94	354.5	391.34	247.21	342.76	349.31	361.47
neuron	74.9	76.73	82.91	107.27	99.26	79.74	104.89	103.57	105.66
direct_rf	6652.35	6650.63	6906.65	8878.3	9274.34	6985.67	9103.73	11851.14	10948.45
bitcoin_miner	6476.47	6536.1	6751.86	9179.34	9021.31	6753.24	9655.61	12675.24	11278.74
dart	456.91	374.37	403.6	549.64	555.84	370.73	561.2	552.96	761.48
minres	489.47	460.09	508.57	724.58	680.09	467.29	702.5	731.58	866.67
openCV	534.62	408.58	575.1	669.56	626.63	484.8	592.69	583.58	668.89
sparcT1_chip2	3557.78	3732.64	5159.07	5629.28	5641.83	3361.19	5247.74	5613.52	6984.04
LU_Network	3213.84	3157.64	2915.67	3447.09	4405.49	3022.95	4259.82	5645.26	5185.59
sparcT1_core	136.47	122.06	182.68	194.32	191.4	132.64	176.23	221.11	207.81
stereovision	68.76	68.15	72.63	99.35	100.68	71.88	95.76	91.31	92.35
cholesky_mc	128.21	130.67	179.27	177.88	212.72	148.58	208.55	199.01	249.74
gsm	1254.86	1292.53	1320.3	1733.07	1746.27	1228.44	1771.1	1806.72	2279.38
mes_noc	2471.22	2434.24	2678.92	2673.38	2873.29	2483.05	3524.51	3809.87	4737.56
denoise	1542.15	1600.46	1557.46	2036.69	2099.5	1573.42	2046.06	2668.23	2371.11
sparcT2_core	1213.01	1226.01	1293.17	1387.79	1758.9	1177.23	1685.55	1826.09	2116.33
cholesky_bdti	481.68	431.91	480.99	670.64	768.92	498.86	647.7	635.16	829.01
bitonic_mesh	677.91	733.85	683.31	959.06	955.92	715.27	991.39	968.73	963.45
SLAM_spheric	233.05	233.28	257.2	316.8	317.01	237.61	310.24	387.68	376.53
stap_qrd	712.39	799.31	1062.04	1067.05	1345.49	743.13	1292.65	1232.73	1105.78

Table C.3: Place Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	1630334	2222375	1699490	1691674	1688227	1677902	1687870	1680717	1704252
LU230	17977564		17121097	17425314	17515026	17977564	17861572	18216804	18281482
des90	2230768	3141202	2180869	2188468	2148707	2165819	2264169	2174047	2193778
neuron	749667	868721	796833	753367	775509	767281	791122	739547	771770
direct_rf	12847272		12640277	12282615	14162065	12847272	12347075	12237858	12025807
bitcoin_miner	10324760		10761398	11273376	10425976	10324760	11171267	10989751	10778382
dart	2237227	2562735	2149848	2172682	2374872	2238463	2208412	2127740	2189159
minres	2913341	3230863	2838023	2900794	2855457	2913341	2831805	2886257	2877307
openCV	3370536	4470938	3335870	3274912	3323662	3536599	3357967	3292667	3363481
sparcT1_chip2	7726080		7756167	7581798	8625259	7814244	7808175	7780711	7918723
LU_Network	5827891	7694743	8029449	8586988	6873401	5827891	5844854	5865696	5873108
sparcT1_core	1293421	1418726			1670878	1309501	1313638	1361608	1683317
stereovision	581019	726505	586764	578793	585170	581019	577193	595967	587856
cholesky_mc	1142987	1272068	1152795	1132415	1308353	1142987	1143175	1158019	1167964
gsm	5385844	7966995	5478572	5340110	5541103	5372486	5451946	5369068	5334188
mes_noc	5138525	8025568		8218017	6741973	5138525	5057744	5199121	5296444
denoise	3000897	4401382	3037349	2979603	3535009	3069575	3022505	2978440	3061310
sparcT2_core	4769636				5717385	4759128	4961884		
cholesky_bdti	2663623	2969989	2646178	2621727	3104937	2663623	2678226	2807666	2693073
bitonic_mesh	4628018	6908874	4446095	4593612	4575821	4628018	4418548	4544161	4655833
SLAM_spheric	1634613	2116361	2051696	1986271	1825090	1634613	1751135	1712715	1713669
stap_qrd	2816056	3667724	4052208	3594191	3325154	2816056	2752426	2705869	2791075

Table C.4: Wirelength Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	135.86	178.65	98.51	118.81	95.91	108.39	115.21	123.77	93.6
LU230	828.39		795.57	969.04	1012.61	807.54	1232.48	1655.94	1443.27
des90	151.68	690.96	151.12	147.44	159.02	144.77	139.47	137.42	154.3
neuron	45.53	49.23	46.48	49.45	41.35	43.57	45.06	40.64	43.22
direct_rf	1133.21		837.85	527.02	1298.45	1185.29	683.56	718.09	581.48
bitcoin_miner	1246.82		1863.48	2345.69	1557.05	1247.31	2361.46	2730.85	1996.62
dart	99.5	120.66	82.73	90.45	100.94	94.52	96.57	90.18	89.59
minres	124.09	131.05	111.4	116.59	118.56	119.71	108.87	117.59	119.84
openCV	229.66	1140.21	244.27	237.31	263.23	247.6	228.86	224.14	257.24
sparcT1_chip2	458.2		378.73	293.38	351.6	373.5	435.93	312.5	308.31
LU_Network	288.3	314.58	1320.75	355.86	284.72	284.44	266.03	279.32	284.78
sparcT1_core	75.93	163.07			246.22	96.21	79.67	75.49	991.56
stereovision	28.22	33.49	28.62	30.33	27.52	28.35	26.76	26.54	30.41
cholesky_mc	81.89	88.6	100.89	80.95	94.75	80.9	94.5	87.92	87.23
gsm	167.08	1415.46	197.74	151.62	260.46	155.1	186.66	162.26	179.38
mes_noc	311.69	4284.52		2832.44	429.82	302.73	444.78	307.39	355.9
denoise	235.98	828.76	236.77	237.27	294.16	304.41	212.36	257.39	188.42
sparcT2_core	248.56				472.29	232.72	244.62		
cholesky_bdti	201.04	304.86	206.44	234.06	213.89	202.94	223.43	271.43	248.01
bitonic_mesh	238.67	2163.13	216.51	242.09	232.6	240.12	236.46	233.94	245.02
SLAM_spheric	92.47	221.86	186.79	144.3	116.7	90.45	134.74	127.07	126.32
stap_qrd	142.38	196	380.83	178.28	164.96	139.9	124.2	116.39	140.18

Table C.5: Route Time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	837.727	843.752	849.167	842.038	842.751	849.41	848.719	848.06	852.394
LU230	23.0087		22.8788	22.7331	23.0204	23.0087	23.1429	23.281	22.8725
des90	12.2596	13.2082	12.0062	11.4679	11.3616	11.5092	12.9878	11.8524	14.6884
neuron	8.05602	8.60584	8.39232	7.31279	7.91946	8.09566	8.78308	8.53124	8.53953
direct_rf	11.2544		9.34572	10.6162	10.7298	11.2544	12.0026	10.2449	10.5959
bitcoin_miner	9.5385		8.96256	11.3854	9.15701	9.5385	11.0363	10.0506	9.36002
dart	14.3573	14.1928	14.6329	14.8588	13.9174	15.395	16.2312	15.6171	15.1534
minres	8.94983	10.3068	8.37289	8.80121	8.08742	8.94983	13.2808	8.04186	8.77988
openCV	10.8759	12.3917	10.6678	10.9693	11.1057	12.7107	10.6561	10.6446	10.6261
sparcT1_chip2	16.6384		17.3413	20.4917	22.3453	20.1695	20.0583	20.6045	22.503
LU_Network	9.38323	10.1428	11.2874	9.62669	8.91605	9.38323	9.85267	9.99766	9.19932
sparcT1_core	8.28235	8.95775			10.0136	8.78549	9.08928	8.97483	11.3201
stereovision	7.24216	7.95187	7.3067	7.49181	7.39499	7.24216	7.34552	7.48752	7.85996
cholesky_mc	7.52925	6.99452	6.97884	7.55045	7.27436	7.52925	7.1392	7.39575	7.55077
gsm	8.20679	9.1301	8.46663	8.90967	8.69276	8.84814	11.1345	9.24426	9.55859
mes_noc	12.0087	15.5267		15.8368	12.802	12.0087	11.7186	13.1087	16.3991
denoise	869.14	860.371	861.101	858.314	858.836	857.427	855.787	862.011	866.985
sparcT2_core	12.2141				13.9381	11.9838	10.8453		
cholesky_bdti	9.34597	9.19163	9.96712	8.83921	9.50759	9.34597	9.09497	9.52333	9.07354
bitonic_mesh	13.3692	14.2004	13.8526	12.4956	13.8699	13.3692	13.4658	13.1608	13.3157
SLAM_spheric	79.831	79.5171	81.3192	79.7969	79.5023	79.831	78.8146	79.3111	80.631
stap_qrd	7.24425	7.67572	7.54324	7.69059	7.11201	7.24425	7.82976	8.10329	7.40853

Table C.6: Critical Path Delay Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	1022.16	1070.52	1060.91	1075.74	1191.42	959.65	1073.38	1096.54	1155.13
LU230	7020.55		6935.3	7490.21	7847.44	6530.08	8046.17	8303.77	9154.99
des90	1227.05	1650.7	1184.44	1127.84	1187.68	1031.56	1102.78	1097.44	1132.16
neuron	420.06	430.67	427.73	456.8	522.78	427.22	449.34	449.95	447.81
direct_rf	10561.5		10107.38	11750.37	14854.69	10686.81	12236.24	14921.87	13965.02
bitcoin_miner	9265.67		10172.07	13105.72	12158.88	9684.41	13576.01	17019.74	14864.93
dart	1282.22	1175.21	1120.26	1257.27	1559.16	1102.25	1298.56	1275.66	1490.97
minres	1669.66	1781.16	1640.2	1899.43	1819.64	1609.54	1818.4	1894.66	1992.98
openCV	1810.88	2486.61	1827.33	1930.35	1793.31	1670.87	1747.6	1717.59	1813.5
sparcT1_chip2	6814.72		8174.54	8591	9842.73	6364.56	8358.9	8679.72	9950.27
LU_Network	5089.6	5901.93	6557.37	6251.79	7044.73	4908.27	6070.94	7522.25	6996.96
sparcT1_core	538.66	608.58	957.71	1606.95	1632.25	551.18	575.52	611.02	1842.81
stereovision	374.33	383.03	375.07	420.34	417.85	378.88	400.84	396.16	405.72
cholesky_mc	530.06	572.58	603.72	589.63	1094.78	568.37	637.42	620.55	675.42
gsm	3032.32	4733.26	3140.15	3506.95	4052.2	3024.91	3564.12	3569.39	4017.25
mes_noc	4748.99	8872.01		8653.01	7610.45	4677.29	5880.04	6014.15	6971.02
denoise	2481.16	3253.21	2488.53	2965.92	4056.11	2584.41	2963.08	3626.01	3252.81
sparcT2_core	2503.7				3785.75	2427.86	2957.98		
cholesky_bdti	1341.44	1468.39	1357.3	1596.59	2719.69	1362.15	1537.14	1563.39	1742.92
bitonic_mesh	2163.22	4397.56	2124.67	2428.52	2436.49	2258.61	2528.76	2512.35	2510.47
SLAM_spheric	636.54	763.68	908.26	918.24	1190.63	640.45	753.61	821.76	810.8
stap_qrd	1445.61	1702.87	2317.49	2104.44	3184.51	1485.01	2009.99	1940.58	1841.27

Table C.7: Total Run-time Results

Benchmarks	NC	AUC	2_A	4_A	16_A	OBP	2_H	4_H	16_H
segmentation	3627.8	3606.6	3757.2	3738	3737	3742.8	3664	3682.7	3666.7
LU230	18317.3		18375	18375.3	18382.2	18369.8	18294.9	18296.5	18296.5
des90	3919.7	3919.2	3919.8	3919.4	3916.7	3916.7	3901.1	3901.6	3903.2
neuron	2747.5	2748.3	2794.3	2798.5	2796.8	2787.4	2761.3	2752.5	2767.8
direct_rf	19531.8		20091.2	20073.9	19890.3	20090.9	19915.7	19882.1	19925.6
bitcoin_miner	13694		14175.6	14184.4	14161.9	14171.2	13947.7	13940.9	13940.4
dart	4198.8	4084.5	4224.6	4220.7	4177	4234	4163.2	4165.9	4155.2
minres	6672.1	6652.3	6711.3	6734.8	6737.4	6735.6	6702.1	6676.7	6705.2
openCV	5805.5	5473.7	5782.4	5780.1	5784.8	5782.3	5753.2	5755.4	5755.2
sparcT1_chip2	12673.7		13220.6	13223.5	13079.1	13272.5	12961.3	12975.8	12994
LU_Network	11096.4	11038.7	11470.6	11494.5	11504.7	11429.2	11188.4	11190.7	11441
sparcT1_core	2271.2	2205.1	2026.2	2034.2	2274.9	2318.5	2301.3	2299.2	2290.2
stereovision	2678.6	2680.7	2725.1	2733.9	2751	2738	2703.1	2701.8	2702.3
cholesky_mc	3087.3	3077	3148.7	3146.2	3129	3143.8	3098.6	3099.1	3099.2
gsm	9532	9581	9862.6	9862.2	9785.8	9860.8	9697.9	9694.2	9700.6
mes_noc	8881.8	8795.5		9309.2	9264.9	9333	9124.2	9133.4	9164.9
denoise	5836.1	5880.5	6129	6141	6096.11	6104.6	6013	5996.8	6016.9
sparcT2_core	5358.9				5633	5657.7	5570.7		
cholesky_bdti	5362.9	5310.3	5563.3	5490.9	5455.4	5487.4	5393.6	5451.9	5451.8
bitonic_mesh	6677	6710.4	6703.8	6706.8	6709.5	6704.1	6676.2	6672.6	6671.9
SLAM_spheric	2757.1	2753.2	2829.9	2827.6	2832.1	2843.5	2795.4	2783.9	2786.7
stap_qrd	4839.1	4848.9	4927.6	4982	4930.5	5023.3	4850.7	4856.2	4845.1

Table C.8: Memory Usage Results

Bibliography

[1] Ziad Abuowaimer et al. "GPlace3.0: Routability-Driven Analytic Placer for UltraScale FPGA Architectures". In: *ACM Trans. Des. Autom. Electron. Syst.* 23.5 (2018). ISSN: 1084-4309. DOI: 10.1145/3233244.

[2] S.N. Adya and I.L. Markov. "Fixed-outline floorplanning: enabling hierarchical design". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11.6 (2003), pp. 1120–1135. DOI: 10.1109/TVLSI.2003.817546.

[3] S.N. Adya et al. "Unification of partitioning, placement and floorplanning". In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* 2004, pp. 550–557. DOI: 10.1109/ICCAD.2004.1382639.

[4] E. Ahmed and J. Rose. "The effect of LUT and cluster size on deep-submicron FPGA performance and density". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.3 (2004), pp. 288–298. DOI: 10.1109/TVLSI.2004.824300.

[5] Pritha Banerjee, Susmita Sur-Kolay, and Arijit Bishnu. "Fast Unified Floorplan Topology Generation and Sizing on Heterogeneous FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.5 (2009), pp. 651–661. DOI: 10.1109/TCAD.2009.2015738.

[6] Christian Beckhoff, Dirk Koch, and Jim Torresen. "Go Ahead: A Partial Reconfiguration Framework". In: *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines.* 2012, pp. 37–44. DOI: 10.1109/FCCM.2012.17.

[7] V Betz, J Rose, and A Marquardt. *Architecture and CAD for Deep-Submicron FPGAs. 1999.* Kluwer Academic Publishers.

[8] Andrew Boutros and Vaughn Betz. "FPGA Architecture: Principles and Progression". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[9] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. "You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–23.

[10] Robert Brayton and Alan Mishchenko. "ABC: An academic industrial-strength verification tool". In: *International Conference on Computer Aided Verification.* Springer. 2010, pp. 24–40.

[11] Yun-Chih Chang et al. "B*-trees: A new representation for non-slicing floorplans". In: *Proceedings of the 37th Annual Design Automation Conference.* 2000, pp. 458–463.

[12] Deming Chen, Jason Cong, and Peichen Pan. *FPGA design automation: A survey.* Now Publishers Inc, 2006.

[13] Doris Chen and Deshanand Singh. "Line-Level Incremental Resynthesis Techniques for FPGAs". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* FPGA '11. Monterey, CA, USA: Association for Computing Machinery, 2011, 133–142. ISBN: 9781450305549. DOI: 10.1145/1950413.1950442.

[14] Tung-Chieh Chen and Yao-Wen Chang. "Modern Floorplanning Based on Fast Simulated Annealing". In: *Proceedings of the 2005 International Symposium on Physical Design.* ISPD '05. San Francisco, California, USA: Association for Computing Machinery, 2005, 104–112. ISBN: 1595930213. DOI: 10.1145/1055137.1055161.

[15] Yu-Hsiang Chen, Chih-Peng Fan, and Robert Chen-Hao Chang. "Prototype of Low Complexity CNN Hardware Accelerator with FPGA-based PYNQ Platform for Dual-Mode Biometrics Recognition". In: *2020 International SoC Design Conference (ISOCC).* 2020, pp. 189–190. DOI: 10.1109/ISOCC50952.2020.9333049.

[16] Lei Cheng and M.D.F. Wong. "Floorplan design for multi-million gate FPGAs". In: *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.* 2004, pp. 292–299. DOI: 10.1109/ICCAD.2004.1382589.

[17] Jason Cong and Yuzheng Ding. "Combinational logic synthesis for LUT based field programmable gate arrays". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 1.2 (1996), pp. 145–204.

[18] Mohamed A. Elgammal, Kevin E. Murray, and Vaughn Betz. "Learn to Place: FPGA Placement Using Reinforcement Learning and Directed Moves". In: *2020 International Conference on Field-Programmable Technology (ICFPT).* 2020, pp. 85–93. DOI: 10.1109/ICFPT51103.2020.00021.

[19] Yan Feng and D.P. Mehta. "Heterogeneous floorplanning for FPGAs". In: *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06).* 2006, 6 pp.–. DOI: 10.1109/VLSID.2006.96.

[20] Intel FPGA. *Intel Quartus Prime Handbook (QPS5V1).* 2018.

[21] Intel FPGA. *Intel Quartus Prime Pro Edition User Guide (UG-20135).* 2019.

[22] Intel FPGA. *Intel Quartus Prime Standard Edition User Guide (UG-20176).* 2021.

[23] Intel FPGA. *Intel Quartus Prime Standard Edition User Guide (UG-20179).* 2018.

[24] Brian Gaide et al. "Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture". In: FPGA '19. Seaside, CA, USA: Association for Computing Machinery, 2019, 84–93. ISBN: 9781450361378. DOI: 10.1145/3289602.3293906.

[25] Licheng Guo et al. "RapidStream: Parallel Physical Implementation of FPGA HLS Designs". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, 1–12. ISBN: 9781450391498. DOI: 10.1145/3490422.3502361.

[26] Pei-Ning Guo, Chung-Kuan Cheng, and Takeshi Yoshimura. “An O-tree representation of non-slicing floorplan and its applications”. In: *Proceedings 1999 design automation conference (Cat. No. 99CH36361)*. IEEE. 1999, pp. 268–273.

[27] Mathew Hall and Vaughn Betz. “From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation”. In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 56–65. DOI: 10.1109/ICFPT51103.2020.00017.

[28] Peter Jamieson et al. “Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research”. In: *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. 2010, pp. 149–156. DOI: 10.1109/FCCM.2010.31.

[29] Gunjan Joshi, P Rajendra Prasad, and Amardeep Singh. “FPGA implementation of channel emulator for testing of wireless air interface using VHDL”. In: *2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*. 2016, pp. 728–732. DOI: 10.1109/RTEICT.2016.7807920.

[30] Cindy Kao. “Benefits of partial reconfiguration”. In: *Xcell journal* 55 (2005), pp. 65–67.

[31] Vasilii M. Khvatov and Daniil A. Zheleznikov. “Development of an IP-cores Libraries as Part of the Design Flow of Integrated Circuits on FPGA”. In: *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. 2021, pp. 2686–2691. DOI: 10.1109/ElConRus51938.2021.9396219.

[32] Dirk Koch et al. “Partial reconfiguration on FPGAs in practice — Tools and applications”. In: *ARCS 2012*. 2012, pp. 1–12.

[33] B. Krill et al. “A new FPGA-based dynamic partial reconfiguration design flow and environment for image processing applications”. In: *2010 2nd European Workshop on Visual Information Processing (EUVIP)*. 2010, pp. 226–231. DOI: 10.1109/EUVIP.2010.5699127.

[34] Helena Krupnova and Gabriele Saucier. “FPGA-based emulation: Industrial and custom prototyping solutions”. In: *International Workshop on Field Programmable Logic and Applications*. Springer. 2000, pp. 68–77.

[35] Chris Lavin and Alireza Kaviani. “RapidWright: Enabling Custom Crafted Implementations for FPGAs”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2018, pp. 133–140. DOI: 10.1109/FCCM.2018.00030.

[36] Christopher Lavin et al. “HMFlow: Accelerating FPGA Compilation with Hard Macros for Rapid Prototyping”. In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. 2011, pp. 117–124. DOI: 10.1109/FCCM.2011.17.

[37] David Lewis et al. “The Stratix II Logic and Routing Architecture”. In: *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. FPGA ’05. Monterey, California, USA: Association for Computing Machinery, 2005, 14–20. ISBN: 1595930299. DOI: 10.1145/1046192.1046195.

[38] Zhifeng Lin et al. “An Incremental Placement Flow for Advanced FPGAs with Timing Awareness”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), pp. 1–1. DOI: 10.1109/TCAD.2021.3120070.

[39] Nan Liu, Song Chen, and Takeshi Yoshimura. “Floorplanning for high utilization of heterogeneous FPGAs”. In: *2011 12th International Symposium on Quality Electronic Design*. 2011, pp. 1–6. DOI: 10.1109/ISQED.2011.5770736.

[40] Jason Luu. *A Hierarchical Description Language and Packing Algorithm for Heterogenous FPGAs, MASc Thesis*. University of Toronto (Canada), 2010.

[41] Jason Luu, Jonathan Rose, and Jason Anderson. “Towards interconnect-adaptive packing for FPGAs”. In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 2014, pp. 21–30.

[42] L. McMurchie and C. Ebeling. “PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs”. In: *Third International ACM Symposium on Field-Programmable Gate Arrays*. 1995, pp. 111–117. DOI: 10.1109/FPGA.1995.242049.

[43] Yibai Meng et al. “elfPlace: Electrostatics-Based Placement for Large-Scale Heterogeneous FPGAs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.1 (2022), pp. 155–168. DOI: 10.1109/TCAD.2021.3053191.

[44] Hiroshi Murata et al. “Rectangle-Packing-Based Module Placement”. In: *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’95. San Jose, California, USA: IEEE Computer Society, 1995, 472–479. ISBN: 0818672137.

[45] Kevin E. Murray and Vaughn Betz. “HETRIS: Adaptive floorplanning for heterogeneous FPGAs”. In: *2015 International Conference on Field Programmable Technology (FPT)*. 2015, pp. 88–95. DOI: 10.1109/FPT.2015.7393136.

[46] Kevin E. Murray and Vaughn Betz. “Tatum: Parallel Timing Analysis for Faster Design Cycles and Improved Optimization”. In: *2018 International Conference on Field-Programmable Technology (FPT)*. 2018, pp. 110–117. DOI: 10.1109/FPT.2018.00026.

[47] Kevin E. Murray, Sheng Zhong, and Vaughn Betz. “AIR: A Fast but Lazy Timing-Driven FPGA Router”. In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 338–344. DOI: 10.1109/ASP-DAC47756.2020.9045175.

[48] Kevin E. Murray et al. “Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap between Academic and Commercial CAD”. In: 8.2 (2015). ISSN: 1936-7406. DOI: 10.1145/2629579.

[49] Kevin E Murray et al. “Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8.2 (2015), pp. 1–18.

[50] Kevin E. Murray et al. “Titan: Enabling large and complex benchmarks in academic CAD”. In: *2013 23rd International Conference on Field programmable Logic and Applications*. 2013, pp. 1–8. DOI: 10.1109/FPL.2013.6645503.

[51] Kevin E Murray et al. “Vtr 8: High-performance cad and customizable fpga architecture modelling”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 13.2 (2020), pp. 1–55.

[52] Kevin Edward Murray. *Divide-and-Conquer Techniques for Large Scale FPGA Design, MASc Thesis*. University of Toronto (Canada), 2015.

[53] Razvan Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10 (2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[54] Tuan D.A. Nguyen and Akash Kumar. “PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’16. Monterey, California, USA: Association for Computing Machinery, 2016, 149–158. ISBN: 9781450338561. DOI: 10.1145/2847263.2847270.

[55] Ralph HJM Otten. “Automatic floorplan design”. In: *19th Design Automation Conference*. IEEE. 1982, pp. 261–267.

[56] Dongjoon Park et al. “Case for Fast FPGA Compilation Using Partial Reconfiguration”. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 235–2353. DOI: 10.1109/FPL.2018.00047.

[57] Andrew Putnam et al. “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services”. In: *IEEE Micro* 35.3 (2015), pp. 10–22. DOI: 10.1109/MM.2015.42.

[58] J. Rose and S. Brown. “Flexibility of interconnection structures for field-programmable gate arrays”. In: *IEEE Journal of Solid-State Circuits* 26.3 (1991), pp. 277–282. DOI: 10.1109/4.75006.

[59] Jonathan Rose et al. “The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’12. Monterey, California, USA: Association for Computing Machinery, 2012, 77–86. ISBN: 9781450311557. DOI: 10.1145/2145694.2145708.

[60] Ali Sadeghi, Mina Zolfy Lighvan, and Paolo Prinetto. “Automatic and Simultaneous Floorplanning and Placement in Field-Programmable Gate Arrays With Dynamic Partial Reconfiguration Based on Genetic Algorithm”. In: *Canadian Journal of Electrical and Computer Engineering* 43.4 (2020), pp. 224–234. DOI: 10.1109/CJECE.2019.2962147.

[61] David Shah et al. “Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 1–4. DOI: 10.1109/FCCM.2019.00010.

[62] Amit Singh and Malgorzata Marek-Sadowska. “Efficient Circuit Clustering for Area and Power Reduction in FPGAs”. In: *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*. FPGA ’02. Monterey, California, USA: Association for Computing Machinery, 2002, 59–66. ISBN: 1581134525. DOI: 10.1145/503048.503058.

[63] Amit Singh, Ganapathy Parthasarathy, and Malgorzata Marek-Sadowska. “Efficient Circuit Clustering for Area and Power Reduction in FPGAs”. In: 7.4 (2002), 643–663. ISSN: 1084-4309. DOI: 10.1145/605440.605448.

[64] D.P. Singh and S.D. Brown. “Incremental placement for layout-driven optimizations on FPGAs”. In: *IEEE/ACM International Conference on Computer Aided Design, 2002. ICCAD 2002.* 2002, pp. 752–759. DOI: 10.1109/ICCAD.2002.1167616.

[65] Love Singhal and Elaheh Bozorgzadeh. “Heterogeneous Floorplanner for FPGA”. In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. 2007, pp. 311–312. DOI: 10.1109/FCCM.2007.31.

[66] Love Singhal and Elaheh Bozorgzadeh. “Novel multi-layer floorplanning for Heterogeneous FPGAs”. In: *2007 International Conference on Field Programmable Logic and Applications*. 2007, pp. 613–616. DOI: 10.1109/FPL.2007.4380729.

[67] Ali Asgar Sohanghpurwala et al. “OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 228–235. DOI: 10.1109/IPDPS.2011.146.

[68] *Speedster7t network on chip user guide (UG089)*. 2019.

[69] Ian Swarbrick et al. “Network-on-chip programmable platform in VersalTM ACAP architecture”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 212–221.

[70] Stephen M. Steve Trimberger. “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore’s Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation”. In: *IEEE Solid-State Circuits Magazine* 10.2 (2018), pp. 16–29. DOI: 10.1109/MSSC.2018.2822862.

[71] Dries Vercruyce, Elias Vansteenkiste, and Dirk Stroobandt. “Liquid: High quality scalable placement for large heterogeneous FPGAs”. In: *2017 International Conference on Field Programmable Technology (ICFPT)*. 2017, pp. 17–24. DOI: 10.1109/FPT.2017.8280116.

[72] Kizheppatt Vipin and Suhaib A. Fahmy. “FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications”. In: 51.4 (2018). ISSN: 0360-0300. DOI: 10.1145/3193827.

[73] WC3. *Extensible Markup Language (XML)*. 2016. URL: https://www.w3.org/XML/.

[74] Qingcheng Xiao and Yun Liang. “Towards Agile DNN Accelerator Design Using Incremental Synthesis on FPGAs”. In: FPGA ’22. Virtual Event, USA: Association for Computing Machinery, 2022, 42–48. ISBN: 9781450391498. DOI: 10.1145/3490422.3502351.

[75] Yuanlong Xiao, Syed Tousif Ahmed, and André DeHon. “Fast Linking of Separately-Compiled FPGA Blocks without a NoC”. In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 196–205. DOI: 10.1109/ICFPT51103.2020.00035.

[76] Yuanlong Xiao et al. “Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks”. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 153–161. DOI: 10.1109/ICFPT47387.2019.00026.

[77] Xilinx. *Hierarchical Design Methodology Guide (UG748)*. 2013.

[78] Xilinx. *Partial Reconfiguration Tutorial (UG743)*. 2012.

[79] Sadegh Yazdanshenas and Vaughn Betz. “Interconnect Solutions for Virtualized Field-Programmable Gate Arrays”. In: *IEEE Access* 6 (2018), pp. 10497–10507. DOI: 10.1109/ACCESS.2018.2806618.

[80] Jun Yuan et al. “LFF algorithm for heterogeneous FPGA floorplanning”. In: *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.* Vol. 2. 2005, 1123–1126 Vol. 2. DOI: `10.1109/ASPDAC.2005.1466538`.

[81] Yong Zhan, Yan Feng, and S.S. Sapatnekar. “A fixed-die floorplanning algorithm using an analytical approach”. In: *Asia and South Pacific Conference on Design Automation, 2006.* 2006, 6 pp.–. DOI: `10.1109/ASPDAC.2006.1594779`.

[82] Yuan Zhou et al. “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs”. In: FPGA ’18. Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, 269–278. ISBN: 9781450356145.