Fast Place and Route Approaches for FPGAs

by

Russell G. Tessier

S.M., Massachusetts Institute of Technology (1992)
B.S., Rensselaer Polytechnic Institute (1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author . . . .
Department of Electrical Engineering and Computer Science
October 26, 1998

Certified by . . . .
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Fast Place and Route Approaches for FPGAs

by

Russell G. Tessier

Submitted to the Department of Electrical Engineering and Computer Science
on October 26, 1998, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

With recent advances in silicon device technology, a new branch of computer architecture, reconfigurable computing, has emerged. While this computing domain holds the promise of exceptional fine-grained parallel performance, the amount of time required to compile a program to a reconfigurable computing platform can be prohibitive for many applications.

A large portion of this compile time is typically spent performing device layout for field-programmable gate arrays (FPGAs), the core hardware components of most reconfigurable computing systems. In this thesis, an new integrated floorplanning and routing system for FPGAs, called Frontier, is detailed. This system has been designed to optimize FPGA layout time at the cost of modest increases in device logic and routing resources. Experimental results are presented which demonstrate an order of magnitude speedup over traditional layout approaches for an island-style FPGA architecture.

A key part of the Frontier system is a depth-first router that significantly reduces the search space required for FPGA routing and leads to decreased run time when compared to a traditional, breadth-first maze router. In the thesis, it is shown that for the depth-first case, the sparse nature of planar switchboxes, found in many island-style architectures, necessitates an additional localized search near net inputs, called domain negotiation, to aid in directing the route of each design net onto a set of routing resources most likely to lead to a successful route.

This router is tightly coupled with a macro-based floorplanner based on hierarchical, slicing approaches. The floorplanner takes advantage of a set of pre-placed and pre-routed macro-blocks that are commonly found in a broad range of computing applications. The depth-first router can be used to rapidly identify congestion in the floorplan and drive a feedback-driven placement relaxation phase. The use of an FPGA floorplanner provides an opportunity to evaluate techniques that isolate intra-macro routing from inter-macro connectivity in ways typically found in ASIC design styles. By following this design style, macro-sized pieces of a user design layout may be replaced without the need to re-place or re-route significant portions of the design.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my family for their help in guiding me to the completion this dissertation. Their undying support over the years has given me the strength to reach this achievement.

Many thanks go to Professor Steve Ward for supervising this work and for giving me the chance to join his research group. Through that initial opportunity, I have been able to take advantage of all the wonderful experiences MIT has to offer. Additionally, I am thankful for his help in enabling my promotion to Graduate Instructor. The inspiration that I have gained from teaching at MIT will hopeful lead to a successful career.

I am indebted to my thesis readers, Professor Anant Agarwal, Professor Srinivas Devadas, and Dr. Thomas Knight for reading and commenting on preliminary versions of this document. Professor Agarwal, in particular, has offered much-needed direction over the years in shaping my graduate research program. A special thanks to Dr. Chris Terman for being so generous with his time and advice over the past year. His help was instrumental in bringing this work to a conclusion.

I would like to give a special acknowledgment to Professor Jonathan Rose, Vaughn Betz, Jordan Swartz, and Yaska Sankar at the University of Toronto for their help in formulating the technical aspects of this work. Their contribution of ideas and software greatly aided in the development of my research.

I have been fortunate enough to make many friends during my stay at MIT. I thank Jonathan Babb for his insights and conversation, which have provided a much-needed diversion from work. A special thanks goes to my officemate, Luis Sarmenta, for listening to my comments and permitting me to watch *Seinfeld* in the office virtually every night for the past year. NuMesh group members Frank Honore, Chris Metcalf, Anne McCarthy, and David Shoemaker deserve recognition for their encouragement of this work.

No acknowledgment list would be complete without mentioning my friends from J-entry in MacGregor House. Thank you for letting me experience what a human place MIT can be.

Finally, I wish to thank all my colleagues from the 6th floor of Tech Square for providing such a stimulating learning environment over the past nine years. I hope that I am lucky enough to find such smart and interesting people at future stops down the road.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since their commercial introduction in the mid-1980's, field-programmable gate arrays (FPGAs) have revolutionized the way digital hardware has been designed and built. Over this period, these commodity digital parts have become invaluable system components due to their ability to implement many different logic functions efficiently and their ability to be easily reconfigured as system hardware requirements change. Progressive improvements in process technology has increased the logic capacity of these devices from the equivalent of a handful of simple TTL logic gates a decade ago to the capacity of a mid-sized application-specific integrated circuit (ASIC) today. Several commercial vendors have announced plans to introduce devices with capacities of more than one million logic gates before the end of this year.

With this available abundance of logic and routing resources, many new application areas for FPGAs have become feasible. Recent advances in logic emulation have made scalable hardware systems with hundreds of FPGA devices available for verification of prototype logic designs and for use as custom computing platforms for applications with large amounts of fine-grained parallelism. While hardware system capacity and capability has matured considerably in recent systems, to a large extent the underlying software systems needed to automatically map user designs and applications to hardware are still in their infancy.

Perhaps the greatest limitation to the use of contemporary multi-FPGA systems for computation and emulation is the amount of time required to place and route circuits inside the individual FPGA devices. For many systems, this compile time is on the order of hundreds of CPU hours as opposed to tens of minutes for typical microprocessor-based computing platforms. Such long turn-around time from conceptual development to physical implementation significantly limits the on-the-fly modification capability of existing FPGA computing applications. Almost all existing FPGA place and route systems are optimized to use as much of the logic and routing resources in a target device as possible. For many FPGA designers developing a single logic design over several weeks or months, compilation time measured in hours is tolerable and preferable to the greater expense of purchasing a larger device that will be only partially filled. For designers using FPGA devices for computing, however, compile times of several hours are unreasonable compared to compile times for microprocessors that typically require only minutes. Furthermore, much of this compile time is currently spent performing placement and routing on functional logic components, used across a set of FPGA computing applications, when a library of pre-placed and pre-routed macros in conjunction with a macro-based floorplanner could be used instead.

The use of field-programmable gate arrays in a given system is generally a tradeoff when compared to a possible ASIC implementation. A full-custom implementation of a circuit will always give higher performance than an FPGA but likely at a higher dollar cost due to reduced production volume compared to the commodity FPGA. In a same light it is possible to make tradeoffs in the implementation of designs inside the FPGA device. For a fixed sized device it is possible to vary the amount of time needed to place and route a design based on the layout algorithm chosen and the amount of design logic and interconnect targetted to the device. In this thesis, it is shown that typical fine-grained place and route approaches currently employed by FPGA vendors do not scale well in terms of compile time versus quality with increased device logic capacity and that there is a need for new layout approaches. While placement, floorplanning, and routing algorithms for VLSI layout have been widely studied for over thirty years, little work has been done in optimizing these algorithms to find a feasible solution quickly at the cost of a modest increase in required resources. Generally, work in this area has focussed on achieving the optimal layout solution, in terms of minimized required resources or optimal performance, at the cost of a significant increase in search evaluation time.

This thesis shows that the use of macro-based components to address placement and routing compile time issues greatly accelerates the placement process at the cost of decreased device logic utilization and, in some cases, slightly higher amounts of required device routing resources. To date, little work has been done in targetting macro-based placement and routing to FPGAs [53] [32]. Currently, no commercial FPGA vendor offers automated floorplanning of user designs or the capability to easily use a library of pre-routed macro components in their CAD tool flow. Additionally, little work has been reported in the research literature in these two areas. Through the use of a new integrated place and route system, this thesis explores tradeoffs between required device resources and place and route time for FPGA devices that have the same basic routing architecture as those currently found in commercial FPGA devices from Xilinx Corporation [4] and Lucent Technologies [3]. These architectures, known as island-style FPGAs [15], are characterized by a fine-grained array of logic cells surrounded by a collection of prefabricated routing segments interconnected by programmable switches. Rather than starting from the assumption that application designs originate as a netlist of fine-grained components at the size of the primitive logic cell, this new system, called Frontier, assumes the design is made of a group of macro-block components and attempts to take advantage of this regularity. Backoff steps are provided if layout cannot be successfully completed using only this novel macro-block assumption.

Recent trends in reconfigurable computing indicate that in many cases incremental changes in FPGA circuits may be required over the lifetime of an application [39]. By isolating placement and routing resources into specific regions of the device these changes can be made without re-placing and re-routing the large amounts of logic circuitry left unmodified by the change. This additional requirement of isolation requires special consideration in the layout process and additional cost since, in general, existing architectures are not designed to directly support this design style. The macro-based approach implemented in Frontier serves as a testbed for this evaluation.

1.1 Preview of Results

This thesis documents the development of a tightly-integrated set of layout tools for island-style FPGAs that have been optimized to reduce place and route time for large logic designs to under a minute in many cases. Much of this compile-time reduction comes from the assumption that the user design is structured as a set of register transfer level (RTL) macro-blocks that have predefined placement and internal routing structure. To support this design style, a macro-based FPGA floorplanner and fast router have been developed that can be tuned to tradeoff implementation quality for layout optimization time. In the course of this dissertation, this system is used to explore a range of tradeoffs between these two competing layout goals. In practice, the interaction between these tools can be varied depending upon tolerable tradeoffs specified by a system user.

A router, based on an A* search algorithm, has been developed and specifically optimized for the routing structures found in existing commercial FPGAs [59]. Through the use of a user-adjustable parameter, the router can be tuned to either aggressively search for a feasible route from net source to destination or rather search for the lowest-cost route using an exhaustive search. Route times of less than one minute for netlists containing 12000 logic blocks are achieved when this router is used in conjuction with devices with plentiful routing resources as compared to an order of magnitude longer route time for a previously reported version of the routing algorithm.

To achieve fast placement times, a macro-based floorplanner based on clustering and shaping has been developed. This floorplanner quickly packs design macros into the FPGA using a dynamic programming approach to break the floorplanning problem into a series of design subproblems. Macro-based designs of up to 12000 blocks are placed efficiently in less than 20 seconds using this floorplanner. The fast router is integrated with the floorplanner to determine if additional modifications to the floorplan should be made following initial placement.

Finally, the floorplanner is combined with the router to not only create feasible layouts in less than 30 seconds for designs of up to 6000 logic blocks, but also to create layouts that can be incrementally modified as user designs change. This integrated layout tool set allows for macro-sized pieces of circuitry to be replaced in the design layout without the need to modify design logic and net routes that are unchanged by the modification. While effective for small designs, the developed layout approach is shown to be non-scalable as device and design sizes increase for existing device architectures. Changes to existing island-style architectures are explored to make scalability feasible.

1.2 Thesis Organization

In the next chapter, the basic device architecture of island-style field-programmable gate arrays is reviewed. This is followed by a brief discussion of the types of computing applications that benefit from fast compilation. These applications typically consist of a collection of macro-blocks whose place and route characteristics can be defined in a library and used repeatedly. The synthesis flow for typical FPGA computing systems, including integration of the new Frontier tool set, is also discussed.

In Chapter 3, a routing algorithm is presented that significantly reduces the amount of time required for routing at the cost of slight increases in required routing resources. A modification is applied to an iterative maze router that achieves an order of magnitude speedup compared to the previously reported router implementation. For a set of benchmark designs, it is shown that routing time can be reduced to less than a minute for circuits containing thousands of logic blocks given approximately 30-40% more tracks per channel than the minimum needed to route the device. The routing algorithm that is used frequently requires multiple routing iterations to approach convergence to a successful route. Generally, the likelihood of eventual convergence of the router can be determined after a single routing iteration. It is shown that a single iteration of the router can be used as a predictor of eventual router convergence to allow for placement modification, if needed. Finally, it is observed that the fast router exhibits *linear* run time growth over a set of design benchmarks of increasing size. This relationship is explored using analytical techniques and it is found that the growth of routing time versus design size determined experimentally matches well with what is predicted from theory. This result helps motivate the use of pre-routed nets in macro-blocks as a means of amortizing route time in later experiments.

In Chapter 4, the growth rate of placement time versus quality for fine-grained FPGA placement approaches is evaluated for increasingly large benchmark circuits. From this analysis, it is seen that an exponential relationship exists between placement time and quality. This growth rate indicates the need for new placement methods that take design regularity into account. Through experimentation, it is shown that much of the time spent in fine-grain placement is expended creating locality that could be preserved through the use of macros.

In Chapter 5, a macro-based floorplanning algorithm is presented that has been designed to quickly identify a feasible floorplan for a design and allow for tight integration of the fast router as a means to evaluate routability. This floorplanner is designed to allow for high logic utilization of the FPGA device and to converge rapidly to a high-quality placement. The floorplanner is tunable to allow for a more exhaustive search in the quest for low-cost placements at the expense of additional computation time. Frequently, since the macros have been pre-placed without consideration to the communication patterns of a specific target design, the resulting floorplanned placement is not as routable as one that could be created using fine-grained placement. Following floorplanning, a single iteration of the fast router is used to identify if the design placement is likely to allow for successful route completion. If not, an additional low-temperature annealing step can be performed to further reduce placement cost.

In Chapter 6, the issue of incremental layout modification for island-style architectures is addressed. First, a hierarchical placement and routing approach that utilizes the macro-based floorplanner and an ASIC-style routing methodology is evaluated. While this approach converges very quickly for designs with small numbers of macros, it is in general not well suited to FPGA routing structures and is not scalable as device capacities increase. The cost of routing resource isolation is quantified over a collection of benchmark circuits.

We conclude this thesis by outlining additional directions FPGA designers could take to overcome layout compile time issues both by refining existing layout algorithms and also through architectural modifications.

1.3 Contributions of the Thesis

Combined together, the tools developed for this thesis represent the first set of layout tools specifically designed for fast place and route in island-style FPGA devices. Specific contributions include:

1. A tunable FPGA router that can achieve routes in seconds for FPGA devices with abundant routing resources (about 40% more than the minimum needed to route the device) or in a few minutes for devices that are routing-constrained.

2. A fast floorplanner that takes advantage of high-level structure in FPGA designs to quickly determine a high-quality placement inside an FPGA device in a matter of seconds for dozens of macro-blocks. A backoff strategy using fine-grained placement approaches can be used if the initial floorplan is determined to be unroutable.

3. A hierarchical routing approach that is integrated with the floorplanner to isolate portions of design placement and routing in specific planar sections of the FPGA device.

Chapter 2

FPGA Device and System Architecture Overview

Before describing the FPGA layout algorithms introduced in this thesis, it is necessary to explain the specific FPGA architecture that is under consideration and the typical application flow for designs targetted to reconfigurable computers which contain these devices. In this chapter, a detailed analysis is made of *island-style* FPGA architectures. First, an example commercial FPGA of this architectural type, the XC4000 family from Xilinx, is described with regard to logic functionality and routing structure. Since its introduction ten years ago, this basic architecture has been the subject of significant analysis and research. Following discussion of the specific commercial device, a generalized research model exhibiting the same basic routing structure as the commercial device is described. This model is used extensively as a test architecture for the algorithms developed in this thesis.

Subsequent to the discussion of device architecture, a review is made of a typical synthesis path for an FPGA-based reconfigurable computer. While most of this synthesis flow is shown to take advantage of the high-level circuit structure of the user application, FPGA-specific CAD approaches typically ignore this structure. As a result, much of the compile time limitations of using reconfigurable computing systems are drawn from biases against the use of high-level design structure made by FPGA layout tools. A change in focus toward the utilization of design structure for FPGA place and route forms the basis for the placement work described in succeeding chapters on floorplanning and macro-based design.

2.1 FPGA Architecture Basics

Most commercial SRAM-based FPGA architectures have the same basic structure, a two-dimensional array of programmable logic blocks, that can implement a variety of bit-wise logic functions, surrounded by channels of wire segments to interconnect logic block I/O [15] [60]. In most cases, FPGA logic blocks contain one or more programmable lookup tables, that can be programmed to perform any Boolean logic function of a small number of inputs (typically 4-5), a small number of simple Boolean logic gates, and one or more flip-flops. User-programmable switches control interconnection between adjacent wire segments and wire segments and logic blocks.

Three main classes of SRAM-based FPGA architecture have evolved over the past decade: cell-based, hierarchical, and island-style. Each architecture is defined by the amount of logic that can be implemented in an array logic block and the length and interconnection pattern of its channel wire segments. *Cell-based* FPGA architectures, such as those available commercially from Atmel Corporation [1] and National Semiconductor [5], consist of a two-dimensional array of simple logic blocks which typically contain two or three two-input logic structures such as XOR, AND, and NAND gates. Inter-logic block communication is primarily made through direct-wired connections from block outputs to inputs on adjacent logic blocks. Small numbers of wire segments that span multiple logic blocks offer a minimal amount of global communication but typically not enough to implement circuits with randomized communication patterns. These routing restrictions frequently limit the application domain of these devices to circuits with primarily nearest-neighbor connectivity such as bit-serial arithmetic units and regular 2-D filter arrays.

Devices with a *hierarchical* architecture, like those available from Altera Corporation [2], contain a 2-D array of complex logic blocks with many lookup tables and flip-flops (typically 8 or more) per block. Inter-logic block signals are carried on wire segments that span the entire device providing numerous high-speed paths between device I/O and internal logic. This architectural choice leads to an ideal implementation setting for designs with many high-fanout signals. These devices can effectively be used to implement many types of logic circuits exhibiting a variety of interconnection patterns.

Island-style devices provide an architectural compromise between cell-based and hierarchical architectures. As detailed in the next section, island-style devices are characterized by logic blocks of moderate complexity generally containing a small number of lookup tables (typically 2-4) per block. Routing channels with a range of wire segment lengths are available to support both local and global device routing.

2.2 Island-style FPGA Architectures

Perhaps the best known of all FPGA architectures is the Logic Cell Array architecture available from Xilinx Corporation [4]. This island-style architecture and those with similar routing structure from Lucent Technologies [3] contain a square array of logic blocks embedded in a *uniform* mesh of routing resources.

The logic block of the XC4000 Xilinx device, shown in Figure 2-1, contains three lookup-tables (LUTs), two programmable flip-flops and multiple programmable multiplexers. With this logic block

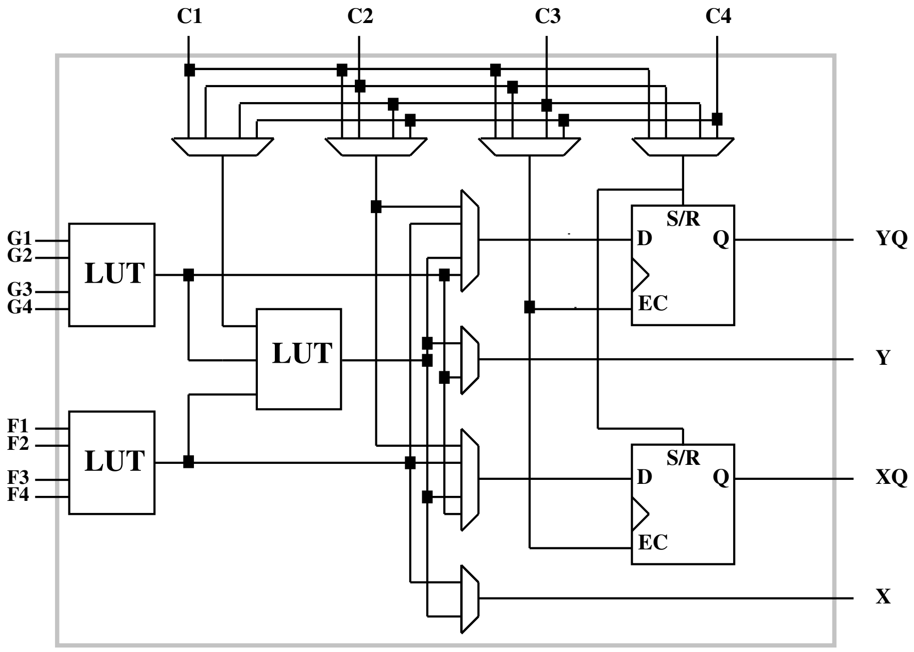

Figure 2-1: Xilinx XC4000 Logic Block

structure any function of five inputs (with **F** and **G** inputs identical), any two functions of four inputs (**F** and **G** inputs different), and some functions of up to nine inputs can be evaluated. The multiplexers can be used to route combinational results to either **X** or **Y** outputs or to flip-flops. The C inputs provide either a ninth data input for the 3-input LUT or direct inputs to the flip-flops.

As mentioned earlier, island-style routing architectures are generally characterized by their two-dimensional symmetry and their inclusion of wire segments that span one or more logic blocks. The percentage of segments of each length (or *segmentation*) in each routing channel along with the grain size of the logic block in terms of look-up tables and flip-flops defines a specific island-style family. The segmentation of wires allows for high-speed connectivity of signals, removing the need for signals to pass through an excessive number of routing switches.

Each logic block and adjacent routing segments is considered a routing *cell*. This single cell can be highly optimized in VLSI layout and then replicated both horizontally and vertically to form a uniform array, reducing the design time needed to create a new device family or facilitating the expansion of an existing family to larger logic array sizes. An illustration of the XC4000 routing cell is shown in Figure 2-2. Most interconnect in this family is in the form of single-length lines with additional connectivity provided by double-length lines and long lines which span the entire array. The small transparent squares in the figure represent programmable connections to allow for connectivity between intersecting segments or segments and logic blocks. In the next section the interconnection philosophy and physical implementation of segment to segment connectivity and segment to logic connectivity is discussed.

Other commercial segmented devices contain additional interconnect segments that spans four logic

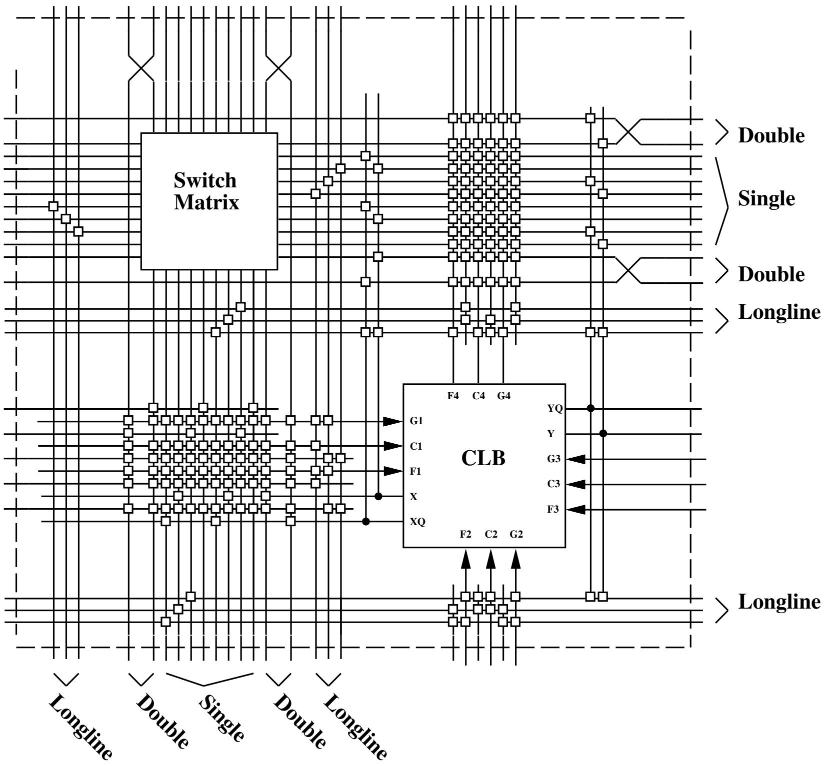

Figure 2-2: Xilinx XC4000 Routing Cell

blocks (XC4000X [4]) and five logic blocks (Orca 3C [3]) while fitting within the limitation of a single routing cell.

Generalizing the Model

The layout algorithms in this thesis are targetted to an architectural model that has the same basic routing structure as the XC4000 series of devices described above but includes a simplified logic block shown in Figure 2-3. This logic block simplification has been used in several other FPGA studies on routing [15] and routing flexibility [46]. In the course of the thesis, it will be shown that since the routing architecture, in terms of switch arrangement, stays the same, the fast placement and routing approaches developed in the dissertation can be applied equally to coarser-grained devices such as the XC4000 family.

The basic routing cell used in this thesis is shown in Figure 2-4. Since a variety of array sizes were used in experimentation, many reaching sizes far greater than existing devices, frequently it was necessary to scale the number of routing segments per channel in order to achieve a successful route. For this thesis, this scaling was always performed to maintain the same segmentation ratio of single length lines (44%), double length lines (22%), and long lines (33%).

Island-style routing architectures can be generalized based on connectivity between adjacent wire segments and between segments and logic blocks. As illustrated in Figure 2-5, routing channels of width

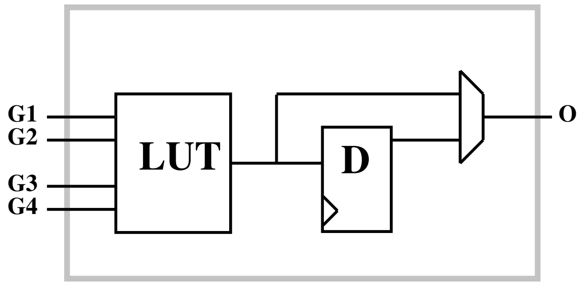

Figure 2-3: Thesis Logic Block

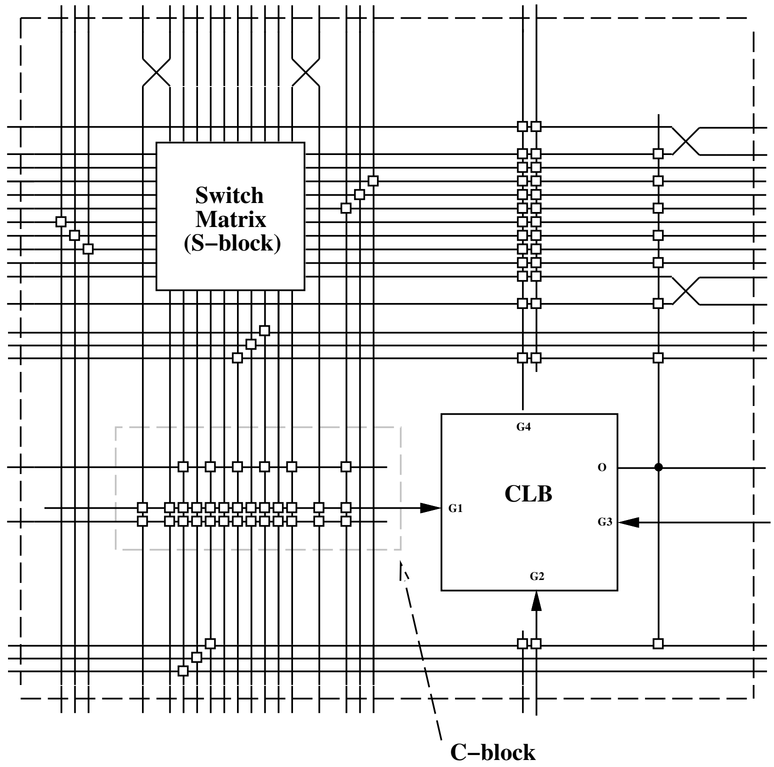

Figure 2-4: Thesis Routing Cell

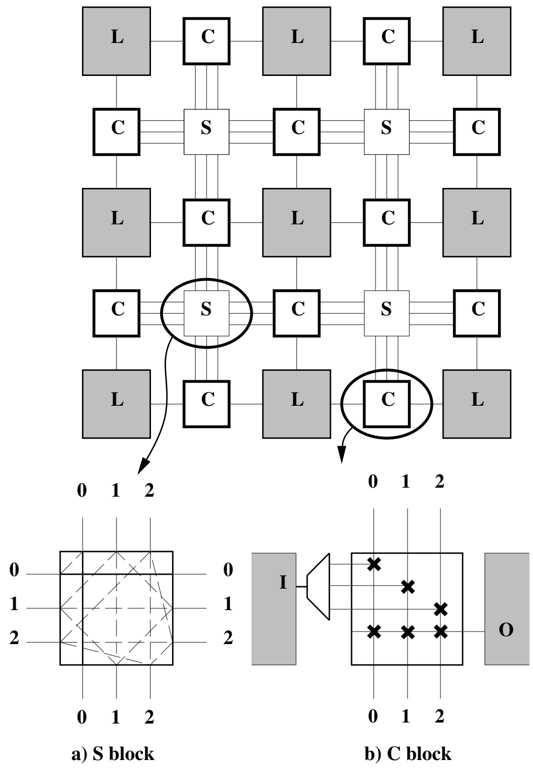

Figure 2-5: Generalized Island-style Model

W (for this figure 3) are connected to logic blocks through a set of programmable switches, referred to as connection or *C blocks*, at the intersection of logic block I/O terminals and channel tracks. In this model, it is assumed that the connection block is flexible enough to connect logic block I/Os to any routing track in the channel ($F_c = W$). A distinctive architectural feature of the FPGA is how each C block is constructed [37]. If each logic block input connection is implemented as a pass transistor, then two or more connections to the pin may be activated to permit a routing dogleg, where the pin and connected wires are shorted together to form a single electrical path. However, as seen in Figure 2-5b, if the input connections are implemented as a multiplexer, only one connection to the tracks can be made and doglegs are not possible. To maintain consistency with actual devices from Xilinx and Lucent that contain input multiplexers, the latter, no-dogleg case for logic block inputs is assumed.

Wire segments in routing channels span one or more logic blocks in the horizontal or vertical dimension. Switchboxes, or S-blocks, allow a predefined set of programmable connections between wires at the intersection of horizontal and vertical track channels. Figure 2-5a shows that each switchbox is sparsely connected so that each horizontal or vertical wire entering the switchbox can connect to only three possible destinations ($F_s = 3$). For wire segments that span multiple logic blocks, such as those labelled 0 in Figure 2-5a, wiring passes directly through the S-block and is represented as a solid line. Programmable S-block connections between segments are represented with dashed lines.

The limited connectivity of the switchbox topology divides routing tracks into disjoint routing sets or *domains*. As first noted in [61], a routing domain can be defined as follows:

> **Routing Domain:** A set of discrete wire segments in an FPGA device that can be connected together to form a routing path. If a path cannot be formed between two wire segments through switches or other segments, the segments are said to be in different domains.

Given the physical switchbox constraints for the architecture used in this dissertation, the number of FPGA domains always equals the number of wire segments in a device routing channel, W. In Figure 2-5, a total of three domains are indicated.

With the given S-block and no-dogleg restrictions, a net beginning in a given track domain at the net output pin is restricted to only wire segments in that domain, no matter which S-block switches it passes through or net input pins it touches. The sparseness of the switchbox coupled with the inability to switch tracks at logic block inputs leads to the constraint that routing domain changes can only occur at net *outputs* even for nets with high fanout.

The discretization of routing resources into domains plays an important role in optimizing routing algorithms for segmented routing. In the next chapter, a routing algorithm is developed that specifically takes routing domains into account to accelerate the routing search for feasible connections.

2.3 Typical Application Synthesis Flow

Reconfigurable computers based on FPGAs have shown impressive speedups for a number of computing applications by customizing the underlying logic of the computing platform to create exactly the hardware functionality required. Typically, due to the size of the circuit created to perform the computation, multiple FPGA devices are needed for design implementation. A number of recent projects [28] [10] [7] have used hundreds of FPGA devices in concert as a reconfigurable computing platform to solve computational challenges such as shortest-path search calculation, array sorting, FFT calculation, and special-purpose processor implementation. Advances in high-level compilation technology for these computing domains will likely lead to a rapid increase in the number of potential applications for reconfigurable computing.

As the complexity of reconfigurable computing applications and target platforms grows, the ability of application designers to map designs by hand to reconfigurable hardware becomes limited by the amount of time needed to analyze the complex variety of hardware implementation tradeoffs available. These limitations have given rise to automated high-level design flows for multi-FPGA reconfigurable computing platforms [44] [24]. While the specific details of individual systems vary, most follow the general synthesis flow that is outlined in Figure 2-6.

It will be seen that in the early stages of the compilation process the high-level structure of the design is exploited to aid in making implementation tradeoffs. Extending this model to FPGA layout provides the opportunity for experimentation studied in this thesis.

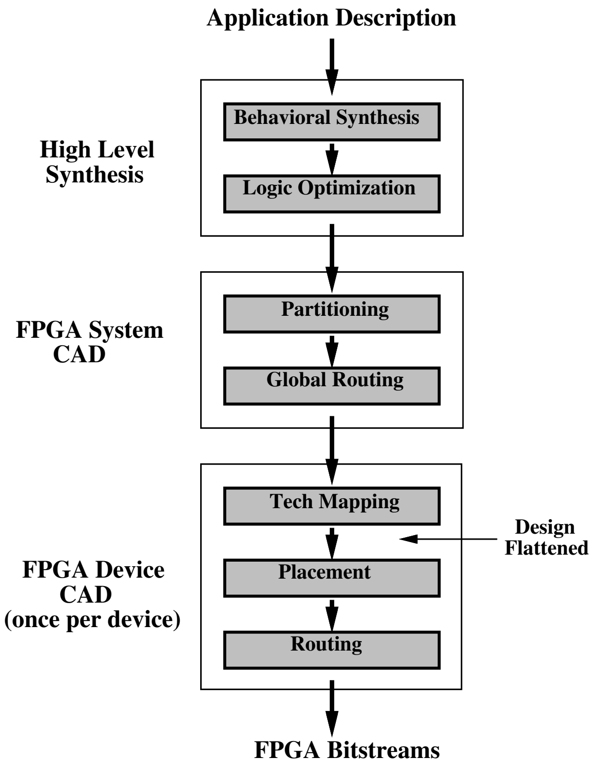

Figure 2-6: Reconfigurable Computing Synthesis Flow

2.3.1 High-level Synthesis

Recently, several projects exploring program specification for reconfigurable computing have resulted in the development of compilers that process the same procedural or object-oriented constructs used for microprocessor-based systems [44] [7]. A user algorithm is typically specified in a high-level language (such as C or C++) or in a behavioral hardware description language (VHDL or Verilog). This representation not only serves as a basis for synthesis but also can be simulated on a microprocessor for verification. Unlike microprocessor systems which require conversion of the textual representation to a sequence of simple processor instructions, reconfigurable computing systems require the generation of a complete hardware circuit. This synthesis step typically requires the *allocation* of datapath hardware resources in the form of high-level blocks such as ALUs, multipliers, and memory components, and the *scheduling* of communication between these components. Control of scheduled communication is maintained through the creation of control circuitry. This set of datapath and control structures form a register transfer level (RTL) representation of the application.

In the next step of the translation process, portions of the design in the control structure and datapath are optimized to a minimized set of Boolean logic gates through logic optimization [22]. Frequently, this optimization is the same for FPGAs as for other VLSI technologies, such as full-custom design, and involves evaluation of issues such as required design performance and available circuit area. The result of logic optimization is a structural netlist of gate-level components grouped within the coarse-grained datapath macro-blocks defined by high-level synthesis.

2.3.2 FPGA System CAD Flow

Following creation of a macro-based circuit representing application behavior, the circuit must be mapped to a hardware system consisting of multiple FPGA devices. The steps by which a specific reconfigurable computing software system performs this translation process varies somewhat from system to system but in general the macro-based netlist created by high-level synthesis must be *partitioned* into smaller netlists for each FPGA device and inter-FPGA signals must be globally *routed* using system-level routing resources. A comprehensive discussion of contemporary reconfigurable computing system architecture and CAD can be found in [30].

In the partitioning step, the netlist generated by logic optimization is subdivided into pieces of circuitry small enough to meet the logic and inter-chip communication capacities of the target FPGA devices. Bipartitioning algorithms such as mincut [25] are recursively applied to the initial netlist until appropriately-sized clusters of logic have been created. As part of the partitioning process, each cluster may be assigned to a specific FPGA to guarantee that specific system-level bandwidth requirements are met. The presence of high-level macro-blocks frequently is used to aid in the partitioning and placement process [44] and to allow for high-level analysis of data movement in the system.

Following assignment of logic clusters to FPGAs, inter-FPGA connections are assigned to specific pins on the FPGA device and inter-FPGA signals are routed using system-level routing resources.

2.3.3 Island-style FPGA CAD Flow

Technology Mapping

Earlier in this chapter it was stated that the basic element of an island-style FPGA architecture is a logic block which typically contains a small number of lookup tables and flip-flops. In the technology mapping step of FPGA compilation, the functionality of primitive logic gates, generated during logic optimization, is restructured into sets of these basic blocks. If a primitive gate has more inputs that a single lookup table, its functionality must be spread across several LUTs. Alternatively, if primitive gates contain too few inputs, small numbers of gates may be clustered together into groups for translation. Technology mapping for FPGAs has been widely studied [60] [15] and a number of effective heuristic approaches have been developed to perform design mapping.

FPGA Placement

After technology mapping, all design logic has been mapped into logic blocks at the quantization level of the basic block of the island-style device. The next step in the translation process is to assign the packed blocks of logic to specific logic block locations in the prefabricated two-dimensional array. The goal of placement for island-style FPGAs is to create a placed configuration of logic blocks that can be successfully interconnected in a subsequent routing step given the routing resources available. Ideally, it would be desirable to estimate localized routability in each subsection of the target device since failure at any specific point in a subsequent routing step leads to an overall mapping failure. In practice, given the distributed nature of interconnect and the dependencies created by segmentation, this becomes infeasible and the total design wire length of all design nets is used as an evaluation metric for quality of placement and routability.

While a thorough analysis of the following *fine-grained* placement algorithm is presented in Chapter 4, a brief overview is presented here. An initial logic block placement can be optimized by swapping pairs of blocks in an effort to find intermediate placement configurations that have lower overall cost. Greedy acceptance of cost improvements frequently leads to intermediate placements that, while locally optimal, are dependent on the order in which blocks are swapped and may be far from the globally optimal solution. For this reason, almost all island-style FPGA architectures use a variant of the iterative simulated annealing algorithm [50] for placement. Annealing algorithms are characterized by their acceptance of not only lower-cost permutations of logic blocks but also by their acceptance of a percentage of higher-cost permutations at various points in the progression of the algorithm to avoid premature convergence to local placement minima.

In order to allow the flexibility to swap most or all of the fine-grained logic blocks in a device, FPGA CAD software frequently flattens the structure of the input design netlist and considers the design as a random collection of interconnected logic blocks. In Chapter 5 this assumption is revisited to show that keeping design structure can aid in the placement process in many cases.

FPGA Routing

Routing is the process of identifying exactly which routing segments and switches should be used to create connected paths from net sources to net destinations for all nets in a circuit. In typical island-style CAD systems, routing is performed after logic block placement is complete due to the NP-complete nature of each problem. Routing for FPGAs is complicated by the fact that the amount of routing resources in the FPGA device is fixed. In general, routing resources in non-congested portions of the device will be wasted while resource overuse in congested parts may lead to failure to achieve a successful route.

By far the most popular routing algorithms for island-style FPGAs are *maze-routing* algorithms [36] based on Dijkstra's shortest path algorithm. This algorithm routes each net sequentially. The routing search starts at a net source and is followed by an iterative evalution of wiring segments, based on segment cost, in an effort to avoid congested resources. If all net routes are not initially successful, selected nets are ripped-up and rerouted in an effort to free contested resources. Additional information

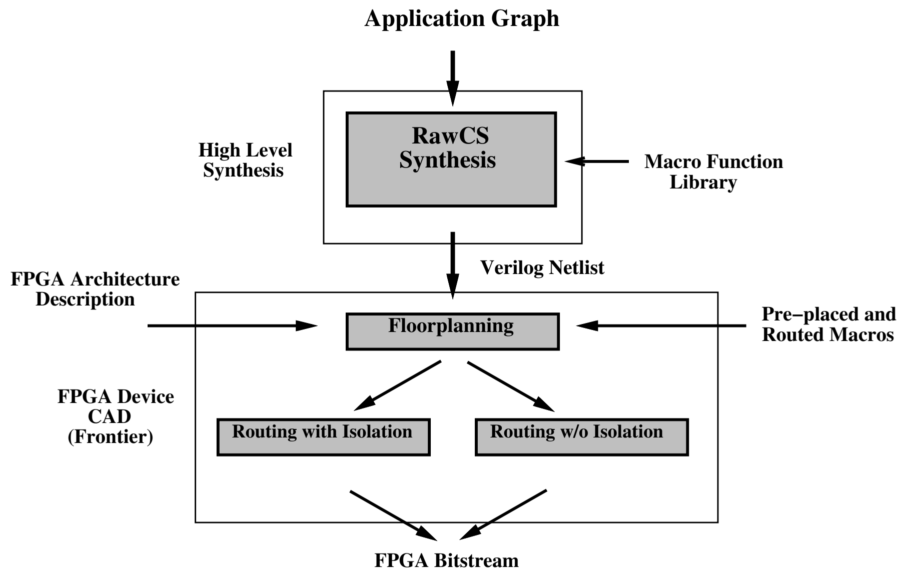

Figure 2-7: Frontier System Flow

on contemporary maze routing algorithms for FPGAs is presented in Chapter 3 along with a new, fast maze routing algorithm tuned especially for island-style FPGAs.

2.3.4 Design Flow Summary

Of the tasks listed in Figure 2-6, 90% of the compilation time for reconfigurable computing systems is typically spent performing FPGA place and route. This is due to the fact that while the other steps in the synthesis process typically optimize at the macro-block level, annealed placement of individual logic designs takes place at the grain size of the primitive logic blocks of the device. Not only does this approach require the placement tool to reconstruct locality information for a design which may contain high-level structure each time placement is performed, but also requires that all nets in the design be routed from scratch each time. This thesis focusses on techniques to use macro block information in a design to accelerate both the placement and routing process.

2.4 Frontier Place and Route System Flow

The basic flow of the Frontier Place and Route System used for experimentation in this dissertation is shown in Figure 2-7. Each step in this design flow is briefly described in subsequent subsections. Note that for this thesis, FPGA designs targetted to a single device were evaluated, thus eliminating the need for FPGA system CAD steps, such as design partitioning and global routing shown in Figure 2-6. If designs larger than a single device were synthesized in the high-level synthesis step, FPGA system CAD could be performed after high-level synthesis to create multiple device partitions, each of which could then be applied separately to the Frontier system.

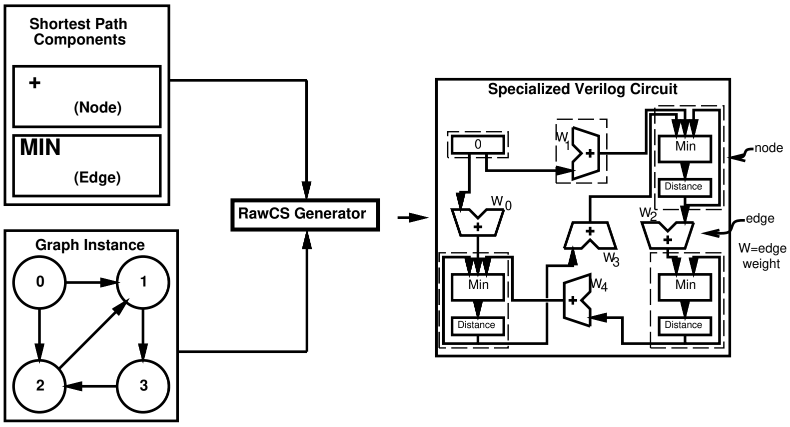

Figure 2-8: RawCS Flowchart for Shortest Path

Benchmark	Description
bheap	Binary Heap
bubble	Bubble Sort
fft	Integer Fast Fourier Transform
merge	Merge Sort
ssp	Single Source Shortest Path
spm	Multiplicative Shortest Path

Table 2.1: The RAW Benchmarks

2.4.1 The RAW Synthesis System and Benchmark Suite

In order to evaluate the benefits of considering design structure in the FPGA place and route process, a number of macro-based benchmark applications from the RAW Reconfigurable Computing Benchmark Suite [8] are used. These applications have been used previously in several reconfigurable computing studies [8] [18] and represent a range of applications currently applied to reconfigurable computing platforms.

A list of the benchmark applications used for experimentation appears in Table 2.1. Specific instances of the benchmark applications are created using Raw Computation Structure (RawCS) synthesis, a front-end compilation tool for reconfigurable computing [8]. Each application developed using the RawCS framework consists of a library of component macro-functions and a set of parameters specifying graphically how instances of these functions should be connected together to solve a specific computing problem. The RawCS generator uses these inputs to create a Verilog netlist of macro-functions that is used as the input to the Frontier system.

An example of this synthesis flow is shown in Figure 2-8. In this case, RawCS is used to create hardware circuits to solve shortest path graph applications. The RawCS generator takes as input a topological description of a graph instance specified as a text file. Based on this topology, the generator instantiates and interconnects components from a library of node and edge computation structures to

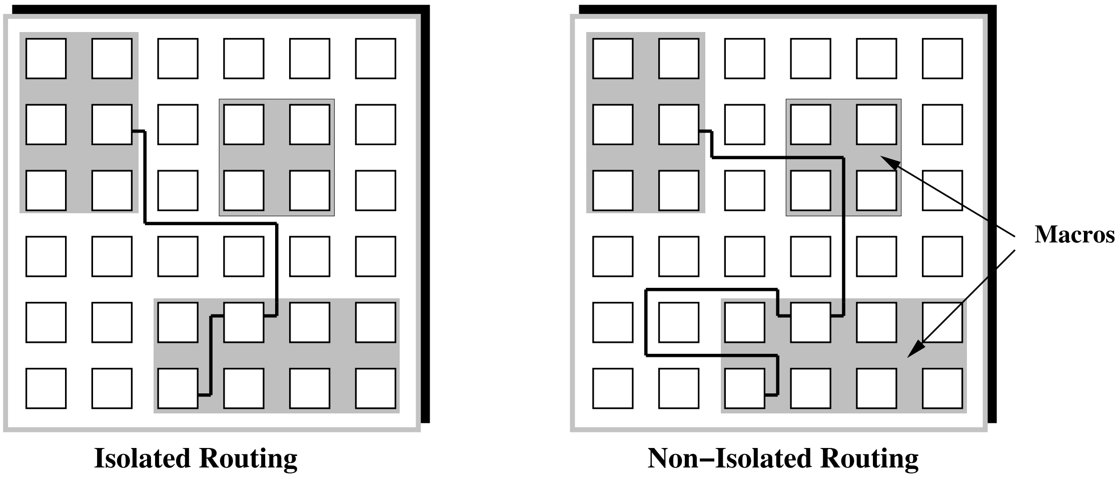

Figure 2-9: Isolated Versus Non-isolated Routing

create a final circuit description in Verilog. While experiments for this thesis are targetted for an individual FPGA, the RAW benchmark circuits could be thought of as pieces of a larger graph that take up multiple devices partitioned by a multi-FPGA CAD system.

2.4.2 Frontier Floorplanning

The result of synthesis by RawCS is a Verilog netlist consisting of multiple instantiations of functional macro-blocks. In the floorplanning stage, described in detail in Chapter 5, each of these macro-blocks is considered a rectangular shape consisting of FPGA logic blocks that have been previously technology mapped and stored in a library. The result of floorplanning is a placed circuit with each design logic block assigned to a specific device logic block. The non-uniformity of macro-block shapes typically precludes 100% logic block usage in a target device due to shape packing inefficiency, although logic utilization of up to 65% of blocks is common.

2.4.3 Frontier Routing

Following floorplanning, the placed FPGA design is routed using an iterative maze router described in Chapter 3. This router employs an A* search algorithm to quickly locate the first, best route from net source to destinations.

A key issue which guides how the routing algorithm assigns routing resources to individual nets is routing resource *isolation*. Much of the compilation speed for microprocessor-based systems is due to the modularity of the compilation process. For these systems, application progam modules are compiled individually and linked together to form a complete program object file. If a specific module in the high-level program changes, only the affected module need be recompiled and subsequently linked to unchanged objects to form a new object block. In this thesis, to promote fast, incremental compilation, analagous modular routing approaches for island-style FPGAs are considered. Internal macro-block routing is isolated on a pre-allocated set of routing tracks to avoid congestion from inter-macro nets and to allow for incremental recompilation following a logic or routing change in a specific macro.

To illustrate isolation, consider the two routing examples shown in Figure 2-9. In the left-hand example, all intra-macro routing is restricted to routing resources within the planar extent of the macros and inter-macro nets are routed around macros in regions not covered by macro logic. In contrast, the right-hand example shows intra and inter-macro nets routed without regard to macro boundaries (without isolation). While the latter case is more flexible, if there is a need to make a change inside the macro, non-local nets must be taken into account, thus making the modification process more difficult. In the former case, only internal macro routing need be considered. This partitioning of routing resources by macro boundaries is an example of *planar isolation*, one of the isolation approaches examined in Chapter 6.

Isolation of resources also has the added benefit of limiting the routing search space and can lead to very fast routes at the cost of reduced device logic utilization. It is shown in Chapter 6 that completed layouts for small designs of a few thousand logic blocks can typically be completed in less than 30 seconds.

2.5 Other Fast FPGA Layout Systems

Recently, researchers at the University of Toronto have started a fast layout project for island-style FPGAs that uses approaches that are similar to some of the ones used in Frontier. As of the writing of this dissertation, only limited results for the Toronto layout approaches have been published [56], but additional experimental results are expected to be reported soon [55] [49]. A brief high-level review of the Toronto system is presented here based on preliminary information obtained from University of Toronto researchers [54] [48].

The fast placement approach developed at the University of Toronto combines logic-block clustering with simulated annealing to reduce overall placement time while still achieving high-quality placement results [48]. Like the fine-grained FPGA placement method outlined in Section 2.3.3, the Toronto system commences layout with a input logic design that has been mapped to a number of fine-grained logic blocks. Small numbers of these fine-grained blocks are clustered together based on connectivity to form small logic block groups of approximately the same size. These groups are then assigned to equally-sized device regions and optimized through iterative swapping so that overall placement cost is minimized. Cluster-based placement has the benefit of region assignment that is not restricted by macro shape, a limitation of the floorplan case, but requires regeneration of design hierarchy already defined in RTL macro-blocks.

Like the Frontier router, the Toronto router is an accelerated multi-iteration maze router based on an A* search [56]. The Toronto router is not only optimized to find a feasible route quickly, but also to minimize longest path delay. More detailed differences between the Toronto router and the Frontier router are noted in Section 3.1.4.

Chapter 3

Fast FPGA Routing

Routing for FPGAs is a challenging problem considering the limited amount of routing resources typically found in many devices and the large numbers of nets that need to be routed. While, as noted in Chapter 2, the basic routing approach used by most CAD packages has evolved from heuristic maze routing, changes to the basic routing algorithm can be made to optimize for a specific goal or target device architecture. The router presented in this chapter has been optimized specifically to reach route completion quickly at the cost of a modest increase in required device routing resources. This performance is achieved by converting an exhaustive breadth-first maze route into a shorter depth-first one, effectively trading additional routing resources for decreased router run-time. For the depth-first case, it is shown that the sparse nature of routing switches in island-style FPGA architectures necessitates an additional localized search near net inputs called *domain negotiation* to aid in directing the route of each design net onto a set of routing resources most likely to lead to a successful route. This optimization is shown to have the greatest effect for designs that are difficult to route due to limited routing resources.

The routing algorithm presented in this chapter frequently requires multiple router iterations, each involving the rip-up and rerouting of each design net in a pre-specified order, to converge to a completed route with no overused resources. It is shown that the convergence of this algorithm can be accurately predicted after only one iteration of the router. This information can then be used to motivate placement changes to improve the chances of achieving a successful route.

3.1 Router Implementation

3.1.1 Basic Router Algorithm

The router described in this chapter is based on a maze routing algorithm [36] for finding a path between vertices on a planar rectangular grid. The pseudocode for routing a single net is shown in Figure 3-1. Each net route is started at the source logic block of the net and a search is performed through available

```
Put track segments attached to source onto expansion list.
Remove lowest cost track segment from expansion list.
While still more destinations to reach for this net.
    While a net input has not been reached.
        Put neighbors of current track under evalutation onto expansion list.
        Remove lowest cost track segment from expansion list.
    Endwhile
Endwhile
Empty the expansion list.
```

Figure 3-1: Basic Maze Routing Sequence for a Net

routing resources to find the lowest cost path from source to destination. Candidate wire segments or logic block pins for extending the existing partial route are kept in an expansion list (typically a priority queue) based on a pre-specified cost function. As seen in Figure 3-2, the planar scope of routing resources that are searched is typically limited to those within a bounding box that encompasses all routing resources in the rectangular area bounded by the source, all destinations of the net and a small border region.

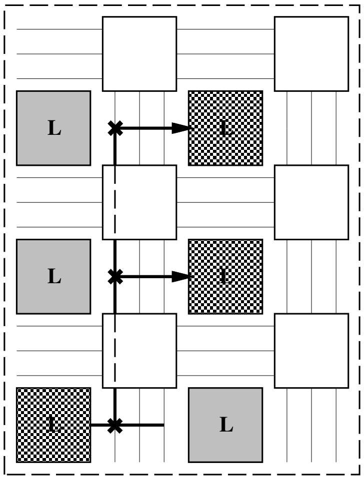

Figure 3-2: Bounding Box for Net with Fanout of 2

In general, maze routing may be defined as a graph problem [40]. Routing resources in an FPGA and their connections may be represented by a graph $G = (V, E)$ where V represents the routing *nodes* or tracks and E represents the connections between the wires or switches. Additionally, each node has an associated cost, c_i, which indicates its current usage, or *occupancy*, among nets targetted to the device. For successfully route completion, each node should have have an occupancy of at most one net.

In earlier results [56] [19], it was observed that Algorithm A* [42] may be applied to this routing problem by considering an evaluation function f at each node n_i in the partial route from a two-point net source to destination as:

$$f_i = g_i + d_i \tag{3.1}$$

where g_i is the cost of the path from the source through n_i and d_i is the *estimated* cost of the path from n_i to the destination.

Value g_i is represented in most maze routing algorithms [36] [40] as the total cost of the previous path f_{i-1} plus the cost of the next candidate node or:

$$g_i = f_{i-1} + c_i \tag{3.2}$$

Typically, the estimate of the path cost from node to destination d_i is ignored, giving $f_i = g_i$. Since maze routing algorithms proceed by expanding around the lowest cost path (f_i) under consideration, the net effect of considering only g_i is a *breadth-first* search, leading to a minimum-cost, shortest-path route. If instead, the preceding path cost (f_{i-1} in Equation 3.2) is ignored and the path cost estimate (d_i in Equation 3.1) is set to the Manhattan distance from node to destination, maze route expansion of the lowest cost path will lead to expansion of the lowest-cost node closest to the destination. By following this rule, a sub-optimal, but much faster, *depth-first* search is performed.

Searches between depth-first and breadth-first can be created by weighting the effect of g_i and d_i via a scaling factor α between 0 and 1:

$$f_i = (1 - \alpha) \times (f_{i-1} + c_i) + \alpha \times d_i \tag{3.3}$$

The node cost, c_i, is used to avoid the use of nodes occupied by previous routes. Since it is always necessary to take a least some congestion into account during maze routing to avoid obstacles, α cannot be set to exactly 1. In the remainder of this chapter, depth-first routing refers to routing with the above cost function biased toward distance rather than congestion ($\alpha > 0.5$). In upcoming results, it is shown that $\alpha = 0.6$ generated the best quality of routing results in the least amount of router run time.

3.1.2 The PathFinder Algorithm

The use of the cost function in Equation 3.3 with the algorithm in Figure 3-1 will frequently lead to nodes with occupancies greater than one after a single router iteration for all nets. To alleviate this congestion, at least some nets must be ripped up and rerouted in an attempt to find less congested paths. In general, maze routing is highly sensitive to the order in which nets are routed. Initial attempts at maze routing focussed on alleviating congestion by ripping up and rerouting small subsets of nets at

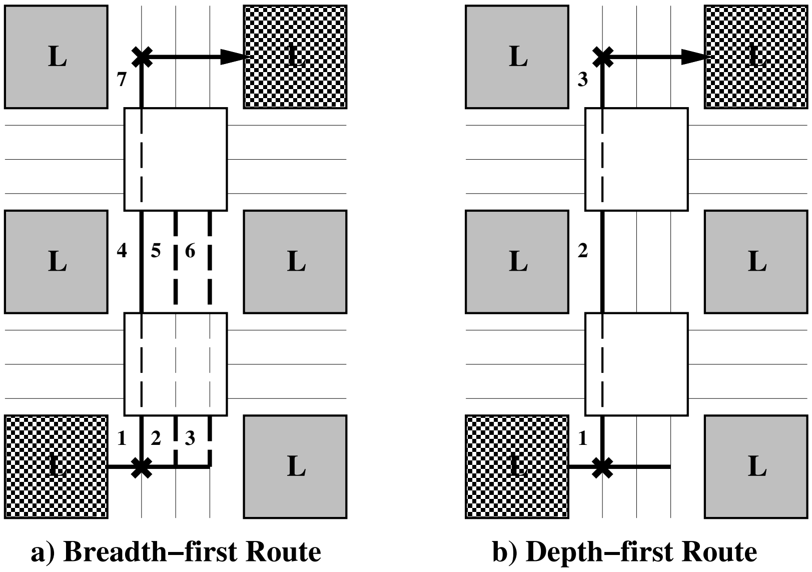

Figure 3-3: Routing Options

a time [20]. While this approach has met with limited success for FPGAs [26], the basic problem with the incremental approach is that the success of the route is not only dependent on the choice of which nets to route but also on the order in which the rerouting is done.

To overcome ordering dependencies, a version of maze routing that sequentially reroutes each net, even those that do not contain congested nodes, has been developed. In the past several years, the PathFinder negotiated congestion algorithm [40] has emerged as a desirable maze routing approach for island-style FPGAs, given its ability to achieve successful route completion, and its simplicity of implementation. For this algorithm, routing is completed in multiple routing iterations. During each router iteration, each net is ripped up and routed in a prespecified order. The cost of each routing node, c_i, in the routing grid is updated to reflect congestion not only after the route of each net, but also after an entire iteration in which every net is routed. This additional cost update allows for the migration of net routes away from congested areas of the device, to those more sparsely populated, through use of a non-decreasing cost factor assigned to each node. While the PathFinder algorithm has been shown to be very efficient in achieving successful routes, its initial implementation [40] involves a time-consuming exhaustive search of all possible routing paths for each net ($\alpha = 0$ in Equation 3.3). The new router, described in this chapter, shows that by reducing the routing search space, a much faster route using the base PathFinder negotiated congestion approach can be achieved with little or no loss of route quality.

3.1.3 Domain Negotiation

In Section 2.2, it was shown that the switchbox structure of island-style devices divides wire segments into disjoint routing *domains*, sets of wire segments that can be connected together to form a routing path. Due to limited inter-segment connectivity, each wire segment in a routing channel belongs to a different domain, indicating that the number of domains in a device is equivalent to the device channel width, W. The transition from a breadth-first route to a depth-first one requires that the disjoint nature of the routing architecture be taken into account. Consider a breadth-first route from a source to destination assuming that all routing nodes have the same cost c_i and span a single logic block. Figure 3-3a illustrates that node expansion takes place across *all* track domains at the same Manhattan distance from the destination prior to expansion of adjacent nodes due to the restriction of always expanding shortest paths. Thus, the final route is completed using tracks in a domain found to have the lowest cost along the source-destination path.

Alternately, in the depth-first case, the node closest to the destination is expanded first before additional points at the same Manhattan distance are expanded. As seen in Figure 3-3b, the net effect of this expansion approach and the disjoint nature of the routing switches is a directed route confined to the same track domain from net output to input. If routing along an initial domain fails, subsequent depth-first routes can be attempted on different domains until route completion is achieved.

A key issue in this depth-first route is deciding the domain order in which routes are attempted. Since domains can not be switched in the course of the route, it is desirable to first attempt routing in domains that have a high likelihood of successful completion so that expansion in additional domains will not be necessary. To perform this domain selection, the concept of *domain negotiation* is introduced. This action can be summarized as follows:

> **Domain Negotiation:** The action of rating the routability of routing domains in an FPGA device for a net based on routing congestion surrounding net destination logic blocks.

For negotiation, domains are ranked based on the occupancy of tracks adjacent to net input pins prior to routing each net. This localized search ranks domains as more likely to succeed in a depth-first route if the current track occupancy around the inputs in a domain is small and less likely to succeed if occupancy is high.

An example of where domain negotiation can be helpful can be seen in Figure 3-4. Here, a single net emanates from the logic block at the bottom, left of the array and has a sink at the top, right logic block. In this case, a depth-first route is initially attempted using tracks in domain 0 (the dashed segments). As the route approaches the destination, it is blocked by high-cost segments surrounding the destination that are already occupied (represented as thick, dashed lines). As a result of this congestion, a new depth-first route is started using tracks in domain 1 in an effort to find a low-cost, non-congested path. If domain 1 had been selected initially, the domain 0 search time could have been avoided. While this example illustrates wasted search time for a net routed with a fanout of 1, this inefficiency becomes more acute for nets with higher fanout.

Details of the domain negotiation algorithm are shown in Figure 3-5. Since logic blocks have multiple input pins that are logically equivalent, occupancy of a single domain track adjacent to a logic block input

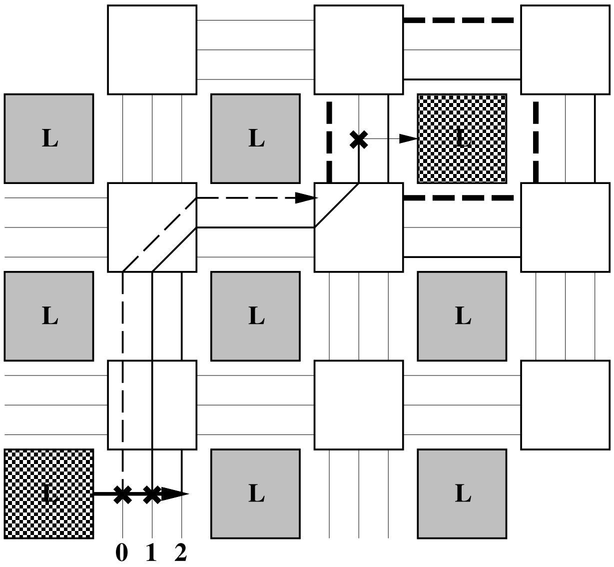

Figure 3-4: Domain Negotiation Example

pin does not prevent route completion in a given domain, but does make it more difficult. Competition for routing resources is reflected in a cost value C_d assigned to each domain. As each net input logic block is visited, domain C_d values are incremented with the occupancy of domain tracks adjacent to logic block input pins. If tracks for *all* logic block input pins in a given domain are occupied, the route cannot be completed in the domain without creating at least one track with occupancy greater than one, a non-feasible physical implementation. To reflect this undesirable situation, domain C_d values are incremented by a penalty factor P for each input logic block that has all domain nodes adjacent to input pins occupied by at least one net. A penalty value of $P > pins_{max}$, where $pins_{max}$ is the number of pins of the highest fanout net, is needed to minimize the number of node expansions required for routing completion and to create minimum track width routes.

As a first step in depth-first routing following domain negotiation, all domains are ranked based on their cost value C_d. The routing domain that has the minimum C_d value is given the smallest rank r_d, while the domain with the largest C_d value is given the largest r_d value. Other intermediate domains are labelled with rank values indicating their relative C_d value.

3.1.4 Depth-first PathFinder Implementation

The modified maze router differs significantly from the breadth-first original in its evaluation of multi-terminal nets. In the depth-first case, a specific target input must be specified to calculate the distance d_i in Equation 3.3. As a result, each input must be connected in a separate routing step. In an attempt to minimize overall wire length, the depth-first router orders target inputs based on distances found using Prim's shortest-path algorithm [45]. The first target input is chosen to be the one closest to the

Loop over track domains
  Initialize domain cost C_d to 0
  Loop over each destination input block
    Loop over tracks adjacent to input pins
      Add track occupancy to cost C_d
    End
    If all tracks adjacent to inputs have occupancy > 1
      Add penalty P to C_d
  End
End
Assign each domain a rank r_d based on cost C_d, $r_d = f(C_d)$

Figure 3-5: Algorithm: Domain Negotiation

net output. Subsequent targets are selected by choosing the input with the shortest path to the net output or to the inputs already chosen. In general, high fanout nets are easier to route when there is less existing routing congestion. As suggested in [56], nets are routed in order of decreasing fanout.

The details of the PathFinder algorithm modified for depth-first routing are shown in Figure 3-6. An expansion list is used to maintain a list of possible tracks for expansion and their related costs. For each net input, the expansion list is initialized to the existing route of the multi-fanout net including the output pin. If routing fails to complete in the domain used by previous net inputs, a new path back to the output pin of the the multi-pin net must be created to allow for a domain change.

Routing steps specifically devoted to domain negotiation have been italicized in Figure 3-6. The order in which domains are searched is controlled by the rank, r_d, of a given domain. As determined during the domain negotiation stage, domains with lower congestion will have a lower rank, r_d, thus promoting routing in less-congested domains first. This rank may be added to tracks attached to the net source by modifying the cost function in Equation 3.3 to include r_d:

$$f_i = (1 - \alpha) \times (f_{i-1} + c_i) + \alpha \times d_i + r_d \qquad (3.4)$$

All other tracks are added to the expansion list using the cost function in Equation 3.3.

Excluding the italicized domain negotiation steps, the routing algorithm shown in Figure 3-6 is similar to the one developed at the University of Toronto and discussed in [56]. This previous router was targetted to an enhanced FPGA routing architecture that did not contain the disjoint switchbox commonly found in commercial architectures such as the Xilinx XC4000 family. The Toronto router is currently being extended to take net delay information into account [55].

Order the sinks using Prim's Algorithm.
Perform *Domain_Negotiation.*
Target = sink closest to source.
Put *track segments attached to source onto expansion list with cost given by (3.4).*
Remove lowest cost track segment from expansion list.
While the net input has not been reached.
    Put neighbors of this track onto expansion list with cost given by (3.3).
    Remove lowest cost track segment from expansion list.
Endwhile
Empty the expansion list.
While still more sinks to route for this net.
    Target = next sink determined from Prim's Algorithm.
    Put whole net created to this point onto expansion list with $cost = \alpha \times d_i$.
    Put *track segments attached to source onto expansion list with cost given by (3.4).*
    Remove lowest cost track segment from expansion list.
    While the net input has not been reached.
        Put neighbors of this track onto expansion list with cost given by (3.3).
        Remove lowest cost track segment from expansion list.
    Endwhile
    Empty the expansion list.
Endwhile

Figure 3-6: PathFinder Iteration for a Multi-terminal Net

3.2 Uses of Router for Routability Prediction

In Chapter 5, an algorithm for fast floorplanning is presented. In this chapter, the use of a routability tool will be shown to be of critical importance in evaluating the quality of a candidate floorplan and to motivate localized changes in floorplan placement. Traditionally, determining the routability of island-style FPGA designs from placement has been very difficult even for designs containing a small number of logic blocks. For a 2-pin net with source and destination points separated by Manhattan distances X and Y, there are a total of $\frac{(X+Y)!}{X!Y!}$ possible routing paths. The varied distribution of wire segment lengths found in island-style devices and the limited connectivity of switches make design routability prediction even more difficult. In [16], statistical techniques based on average wire length were used successfully to identify whether small, placed designs were clearly unroutable, but were unpredictable for designs that were borderline unroutable.

Routability detection is clearly desirable for fast compilation to avoid long routing iterations on designs that eventually will be determined to be unroutable. Placements that can be evaluated as impossible to route or difficult to route need to be flagged early in the compilation process to allow either for high-level design changes or additional placement steps. Recently, Swartz and Rose [56] have proposed a promising new approach to routability prediction for island-style FPGAs that uses the total wire length of a placed design to estimate required channel width. From the calculated wire length, an estimate is made of W_{min}, the minimum channel width needed to route the placement in an FPGA device of the given routing architecture. If this value is within a specific channel width range of W_{FPGA}, the available channel width of the device, the routing difficulty level of the design is determined to be either low-stress, difficult, or impossible. While work is continuing on this approach, at this time results have only been reported for island-style devices containing routing segments that span single logic blocks [56]. Additionally, all designs tested to date have been placed in the smallest possible square device that will fit them, fully utilizing all device logic blocks and assuming an even distribution of routing demand.

Another possible approach, suggested by Swartz and Rose in private communication [57] and evaluated here, is to perform a single iteration of the PathFinder router in concert with dynamic evaluation of the amount of routing congestion currently in a design. This approach has the potential to be useful in evaluating candidate floorplans since congestion analysis can be restricted to only used routing segments for designs that partially fill the logic resources of the target device. To evaluate routability, a single iteration of the fast router outlined in this chapter is applied to a circuit design. A routability determination is made based on the amount of overused routing segments (occupancy > capacity) in the design during or immediately following this single iteration.

In order to evaluate the quality of a placement, two pieces of information from the initial routing iteration are needed. First, during the routing, a count is kept of the number of routing segments that are assigned to more than one net. Second, a note is made of the number of nets that have been completely routed (including those with some overused routing segments).

Through experimentation, it has been determined that if the number of overused segments exceeds a predefined threshold, S_{im} (determined experimentally to be when the number of overused segments divided by the number of nets is 0.03) and fewer than 96% of design nets have been successfully routed, the design will not subsequently achieve route completion even with additional router iterations (the **impossible** case). If, after all nets have been routed in a single iteration, greater than S_{df} (determined experimentally to be when the number of overused segments divided by the number of nets is 0.0075) segments are overused, it has been found that routing will eventually complete successfully but with a significant number of additional iterations and up to several minutes of route time (the **difficult** case). If only a few routing segments are overused, only a small number of additional iterations are likely (the **low-stress** case). Finally, no overused segments after one iteration indicates no further routing iterations are needed. This case summary is outlined in Table 3.1. The boundary between low-stress and difficult is somewhat arbitrary depending on whether the user is willing to make placement changes to eliminate routing congestion or whether additional waiting time for the difficult route to complete is acceptable. The key boundary is between the difficult and impossible cases since detection failure here leads to a long wait and the lack of a feasible route.

Overuse	Iteration 1	Nets completed	Status	Designation
$S_{ou} = 0$	Completes	100%	Done	Low-stress
$S_{ou} < S_{df}$	Completes	100%	Requires < 60 s	Low-stress
$S_{ou} > S_{df}$	Completes	100%	Requires > 60 s	Difficult
$S_{ou} > S_{im}$	Ends Early	< 96%	Unroutable	Impossible

Table 3.1: Routing Cases

Circuit	Source	Logic Blocks	DFS-neg Min. Tracks	BFS Min. Tracks
fft16	RAW	11860	12	12
ssp96	RAW	12041	13	12
spm16	RAW	6632	11	11
bubble	RAW	8453	8	8
frisc	MCNC	3556	15	15
s38417	MCNC	6406	11	11
s38584.1	MCNC	6447	11	12
clma	MCNC	8383	16	16
elliptic	MCNC	3849	13	13
pdc	MCNC	4631	21	21

Table 3.2: Benchmark Circuits Data

3.3 Results

Both the breadth-first and depth-first versions of the iterative router were applied to a number of large FPGA benchmarks. Each design was placed and routed in the smallest square FPGA which could contain it. Target FPGAs had the following track length distribution for each routing channel: 44% of channel tracks span one logic block, 22% span two logic blocks, and 33% span the entire array. This length distribution is the same as that found in devices from the Xilinx XC4000 family.

The first four benchmarks in Table 3.2 are from the RAW Benchmark Suite [8]. These benchmarks were placed using a fine-grained placer based on simulated annealing that will be detailed in Chapter 4. The remaining six benchmarks and associated placements are from the FPGA Challenge [14]. In only one case, ssp96, was the minimum track width that achieved a successful route, W_{min}, less for breadth-first routing than for depth-first routing with domain negotiation. The fact that the sum of the minimum track widths for breadth-first and depth-first routing was the same indicates the efficiency of the depth-first routing approach.

All run time results were obtained using a 140 MHz UltraSparc 1 with 288Mb of memory. Figure 3-7 illustrates the importance of domain negotiation in the depth-first routing of array-based architectures. In the non-negotiated case, the lack of domain selection caused many separate paths from the net output pin to input pins on different domains thus leading to the overuse of routing resources. For track widths

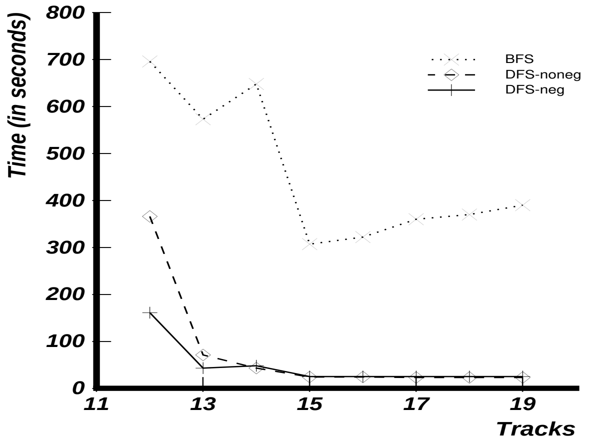

Figure 3-7: Route Time vs. Track Width - Example fft16

	Average Route Time (s)	
	Tracks: W_{min}	Tracks: W_{min}+40%
BFS	709	269
DFS-noneg	647	20
DFS-neg	333	18

Table 3.3: Average Route Times

near the minimum track width, depth-first routes with domain negotiation showed a speedup (as much as 2X) over routes performed without negotiation. In general, the effect of domain negotiation was less as the track widths were increased due to a large increase in possible routing paths for both cases.

Table 3.3 shows average route times achieved across the eight designs with the same minimum track width for all three routing test cases (DFS, DFS-noneg, BFS). It can be seen that route time decreases to under a minute for increased routing channels for the depth-first case but continues to be several minutes on average for the breadth-first case. This would indicate that if FPGA device manufacturers created devices with the same logic capacity but additional routing resources, depth-first routing could allow for device routing in less than a minute for the given placements. Results in Table 3.4 for depth-first routing with negotiation further enhance this point.

3.3.1 Routability Prediction

Table 3.5 shows the results of routability prediction for the ten benchmark circuits. In 32 out of 40 cases the correct difficulty was predicted after just one iteration in the amount of time indicated in the table. Most of the incorrect predictions (6 out of 8, indicated in italics) were of the low-stress/difficult variety where misprediction only costs some additional wait time for the user. Note that in all cases designs

Circuit	W_{min}	Time (s)	W_{min} +10%	Time (s)	W_{min} +20%	Time (s)	W_{min} +30%	Time (s)	W_{min} +40%	Time (s)
fft16	12	161	14	56	15	22	16	22	17	22
ssp96	13	363	15	34	16	34	17	34	18	34
spm16	11	222	13	13	14	25	15	13	16	14
bubble100	8	148	9	38	10	29	11	20	12	12
frisc	15	169	17	20	18	21	20	11	21	10
s38417	11	186	13	12	14	13	15	13	16	13
s38584.1	11	707	13	25	14	15	15	15	16	15
clma	16	1057	18	98	20	72	21	37	23	34
elliptic	13	535	15	60	16	20	17	20	19	11
pdc	21	388	24	57	26	27	28	27	30	22

Table 3.4: Negotiated Depth-first Route Times Versus Channel Width

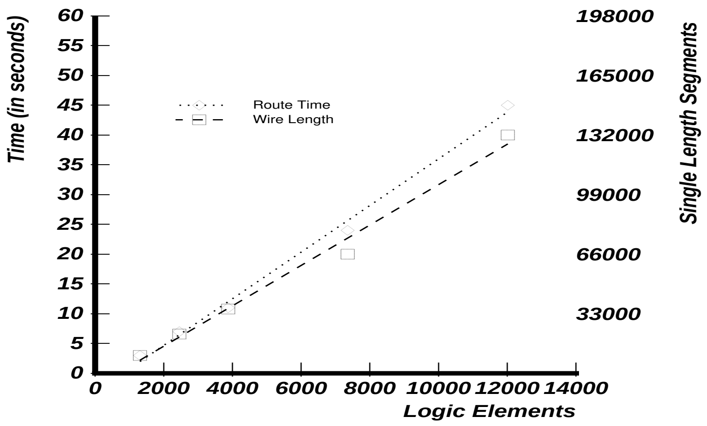

Figure 3-8: Shortest Path Graph Routing Statistics

that were impossible to route were correctly identified. All predictions were determined in less than one minute.

3.4 Estimation of Low-stress Routing Time Growth

Understanding the relationship between routing time and design size becomes increasingly important as both circuit designs and FPGA devices grow. To promote scalability, a growth relationship between these parameters with a slope of at most 1 (e.g., a doubling of design size no worse than doubles route time) would be desirable to maintain a predictable and bounded ratio of routing time to gate count. In this section, it will be shown that even in the low-stress routing case, for most circuits, the slope of the growth relationship between routing time and circuit size is *necessarily* greater than 1 for circuits routed in their entirety (e.g. with no pre-routed nets). Additionally, it will be shown that for a set of benchmark circuits of different sizes with similar basic internal interconnection patterns, this relationship can be

Circuit	W_{min}+2			W_{min}+1			W_{min}			W_{min}-1		
	Crct	Rpt	Time	Crct	Rpt	Time (s)	Crct	Rpt	Time (s)	Crct	Rpt	Time (s)
fft16	LS	LS	18	DF	*LS*	18	DF	DF	23	IM	IM	23
ssp96	LS	LS	38	DF	*LS*	40	DF	*LS*	37	IM	IM	37
spm16	LS	LS	11	LS	LS	13	DF	DF	15	IM	IM	12
bubble	LS	LS	9	LS	LS	9	DF	DF	9	IM	IM	8
frisc	LS	LS	10	DF	*LS*	10	DF	DF	11	IM	IM	11
s38417	LS	LS	10	LS	LS	11	DF	DF	13	IM	IM	12
s38584.1	LS	LS	10	LS	LS	9	DF	DF	11	IM	IM	13
clma	DF	*LS*	41	DF	DF	37	DF	*IM*	42	IM	IM	45
elliptic	LS	LS	10	DF	DF	12	DF	DF	10	IM	IM	9
pdc	DF	*LS*	30	DF	DF	28	DF	*IM*	33	IM	IM	33

Table 3.5: Routability Prediction for Benchmark Circuits

determined analytically without the need to place and route circuits.

The relationship between low-stress routing time and design size is developed below in three steps. First, an analytical evaluation of the relationship between wire length and design size is derived. Second, the relationship between wire length and routing time for maze routing is established. Finally, these two relationships are combined to form the relationship between routing time and design size.

3.4.1 Estimating Total Wire Length

A first step in evaluating the relationship between routing time and design size is the development of an understanding of the relationship between *wire length* and design size for circuits. Clearly, the connectivity (ratio of wires to logic blocks) of all logic designs is not the same. However, for benchmark designs that contain the same basic, scalable interconnection patterns such as FFT or physical implementations of shortest path graphs, it could be expected that interconnection requirements, such as circuit wire length, grow at a predictable rate, defined by generalizable parameters, as design sizes grow. Figure 3-8 illustrates empirically that this is the case for five shortest path graph (ssp) designs, detailed in Table 3.6, that are taken from the RAW benchmark suite [8]. It can be seen that in this case wire length appears to grow at a *linear* rate over a number of design sizes. To create these results, the circuits were first placed using simulated annealing and then routed using the fast router presented in this chapter. Next, it is shown that this information can also be determined analytically.

Design	blocks	Rent exponent (p)	two-pin nets
ssp8	1304	0.56	4032
ssp16	2450	0.53	7554
ssp32	3873	0.57	11793
ssp64	7351	0.58	22311
ssp96	12014	0.59	36664

Table 3.6: Shortest Path Design Statistics

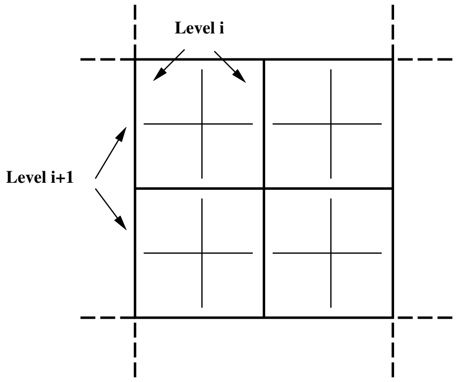

Figure 3-9: Calculation of Average Wire Length with Rent's Rule

A known relationship exists between the amount of logic (or number of logic blocks) in a region of a device and the number of wires leaving/entering the region. This relationship, Rent's Rule [35]:

$$\textbf{Rent's Rule}: \quad N = KG^p \tag{3.5}$$

where N is the number of wires emanating from a region, G is the number of circuit components (or logic blocks), K is Rent's constant, and p is Rent's exponent, characterizes the routing density in a circuit. Most circuits, except for linear arrays with primarily local communication, have been shown to have Rent exponents of $p > 0.5$ indicating that as a quantity of logic scales, the amount of interconnect emanating from it grows faster than its perimeter, which is directly proportional to $G^{0.5}$.

In [23], Donath developed a relationship between the Rent exponent, p, and an upper bound on the average wire length, $\overline{R}$, of a given circuit based on the number of internal logic blocks contained by the circuit. This relationship was determined for a circuit by recursively bipartitioning a design into hierarchical quadrants, as seen in Figure 3-9, and evaluating expected wire distances between quadrants at each level. These expected values were then averaged over all hierarchical levels to create the design average wire length. In closed form, for $p \neq 0.5$, this relationship was determined to be:

$$R_{max} = \frac{2}{9}\left(7\frac{G^{p-0.5}-1}{4^{p-0.5}-1} - \frac{1-G^{p-1.5}}{1-4^{p-1.5}}\right) \times \frac{1-4^{p-1}}{1-G^{p-1}} \tag{3.6}$$

Donath [23] showed that while the R_{max} value determined by Equation 3.6 for a circuit was often substantially larger than the actual value determined later through routing, the ratio of this value to the actual average wire length, $\overline{R}$, was consistant over a number of designs of different sizes or:

$$\frac{R_{max2}}{\overline{R_2}} = \frac{R_{max1}}{\overline{R_1}} = k \tag{3.7}$$

Since values in Equation 3.6 can be easily determined from the number of logic blocks in circuits, the relationship in Equation 3.6 can be used to estimate the average wire length of a larger circuit from a smaller one with the same Rent exponent as:

$$\overline{R_2} = \frac{R_{max2}}{R_{max1}} \times \overline{R_1} \tag{3.8}$$

and since Equation 3.6 is a non-decreasing function with respect to block count, R_{max2} must be greater than R_{max1} and $\overline{R_2}$ must be greater than $\overline{R_1}$.

There is a direct relationship between the average wire length of a circuit and its total wire length. The total wire length for a design can be determined by multiplying the average wire length, $\overline{R}$, by N_{nets}, the number of 2-pin nets in the design. As a result, the ratio of total wire lengths for the two designs is:

$$\frac{length_2}{length_1} = \frac{N_{nets2}\overline{R_2}}{N_{nets1}\overline{R_1}} \tag{3.9}$$

Substituting Equation 3.8 in Equation 3.9 and rearranging terms yields:

$$\frac{length_2}{length_1} = \frac{N_{nets2}R_{max2}}{N_{nets1}R_{max1}} \tag{3.10}$$

Equation 3.10 offers valuable insight into wire length growth as logic block counts increase for designs. For example, consider the case of two designs with the same Rent exponent, one containing *2 times* the number of logic blocks and nets as the other. In this case, the ratio of wire lengths $\frac{length_2}{length_1}$ *must be larger than 2*, since $\frac{N_{nets2}}{N_{nets1}}$ is 2 and $\frac{R_{max2}}{R_{max1}}$ must be greater than 1 due to the fact that Equation 3.6 is a non-decreasing function. Through similar analysis for any two designs of differing logic block counts, it can be shown that *wire length* must always grow at a faster rate than *logic block count* for designs with Rent exponents of greater than 0.5.

In order to take advantage of the predictable relationship between wire length and design, there must be a correlation between wire length and routing time. Maze routing algorithms, including the

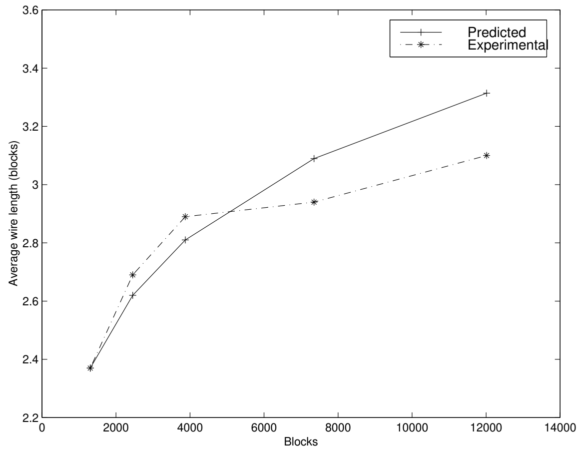

Figure 3-10: Average Wire Length

one outlined in this chapter, use an expansion list in the form of a priority queue to indicate the next, lowest-cost routing segment for evaluation. In the absence of congestion, the depth-first approach will always search a minimum number of routing segments along the shortest path from source to destination to complete a route. Therefore, it can be said that the *time* spent routing nets in the depth-first case is linearly proportional to the *length* of net shortest paths through a proportionality constant, ρ or:

$$routing\ time = \rho \times wire\ length \tag{3.11}$$

where ρ is constant across all designs. As a result, it can be said that the growth rate of routing time with regard to design size follows that of wire length with regard to design size. Results in Figure 3-8 confirm this observation experimentally [1]. In conclusion, since wire length was shown above to grow at a rate faster than logic block count and route time and wire length are linearly correlated through a constant, it can be said that routing time must grow at a faster rate than logic block count (e.g., doubling block count more than doubles route time) for circuits with Rent exponents of greater than 0.5, the common case for most circuits.

Validation of Analytical Approach

To verify the accuracy of the equations derived above and to demonstrate the quality of the fast router, a set of experiments were run on the designs detailed in Table 3.6. For the first experiment, the average wire lengths of all the designs were first determined experimentally using the fast router described in this chapter. Then, the estimated average wire length of the four largest designs were determined by scaling the known average wire length of the smallest design by the ratio in Equation 3.8 with R_{max}

[1] Designs were routed using only single-length segments for this case.

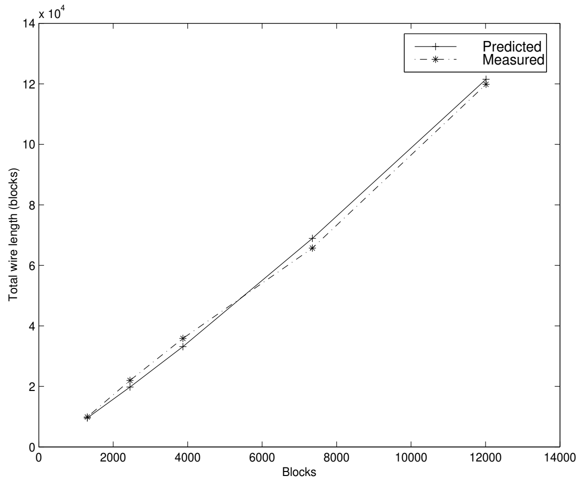

Figure 3-11: Total Wire Length

values determined from Equation 3.6. Results in Figure 3-10 show that values are similar for both experimental and analytical cases.

Known net count values, (N_{nets}), and average wire lengths determined above were then used to determine estimates for the total wire lengths of the four largest benchmarks listed in Table 3.6. In Figure 3-11, these results are compared to experimental wire length values determined from application of the fast router. It can be seen from the figure that the values determined analytically closely matched those determined experimentally.

Chapter 4

Analysis of Fine-grained Approaches

4.1 Fine-grained Island-style Placement

Before exploring the benefits and drawbacks of macro-based placement for FPGAs, an experimental evaluation is performed of existing placement approaches for FPGAs that consider designs as collections of fine-grained logic blocks rather than as structured components. These fine-grained approaches were first developed for mask programmable gate arrays (MPGAs) [51] and then applied to FPGAs as these devices became available. In general, fine-grained approaches achieve good placement results, but at the cost of a long run time due to the sheer quantity of logic block permutations that must be considered. The goal for this chapter is to quantify the tradeoffs between run time and quality for these approaches so that they may be compared with floorplanning approaches in the next chapter. It is shown through experimental results that required placement time to achieve a specific placement quality level for designs grows at a non-linear rate as design sizes increase, motivating the use of macro-blocks in placement.

The relative lack of routing hierarchy in island-style FPGAs requires placement to be formulated as a global optimization problem rather than as a series of smaller localized subproblems the results of which can later be combined together to form a complete layout. Since routing resources in island-style architectures are organized primarily as a flat mesh with limited segment-to-segment connectivity, local placement optimization that fails to take connections to other parts of the device into account generally leads to results that are far from optimal in terms of the number of routing tracks needed to complete routing successfully. This issue of local versus global placement optimization is revisited in the next chapter when the issue of pre-placed macro-blocks is addressed.

Numerous cell placement algorithms for gate array design styles have been developed and implemented. These approaches can be categorized as either constructive or iterative. Constructive placement [52] typically involves subdividing the total placement problem into smaller pieces through the use of clustering or recursive bipartitioning and then merging intermediate results together in a deterministic fashion to form a feasible[1] placement. This division of work allows constructive algorithms to converge quickly to a feasible placement, typically at the cost of placement quality. Given the large number of fine-grained elements typically found in an FPGA device, repeated application of only localized optimization generally results in an overly inefficient final placement. This problem is especially acute for island-style FPGAs given the sparsity of routing switches and tracks prevalent in most devices. As a result, placement approaches that have a more global perspective of the design are needed to achieve desired routability.

4.1.1 Iterative Simulated Annealing

In contrast to constructive techniques, iterative placement algorithms are designed to more fully search the placement implementation space to locate the best possible placement as determined by a predefined cost function. The most common iterative technique for island-style FPGAs (and for many other design problems) is simulated annealing [50]. The simulated annealing algorithm starts with a feasible placement, created either through random assignment of design logic blocks to physical logic blocks, or through the use of constructive approaches and then repeatedly generates placement perturbations in the form of logic blocks swaps. While it clearly makes sense to greedily accept perturbations that reduce overall cost, the search aspect that makes simulated annealing unique is its treatment of swaps that increase or have no effect on overall cost.

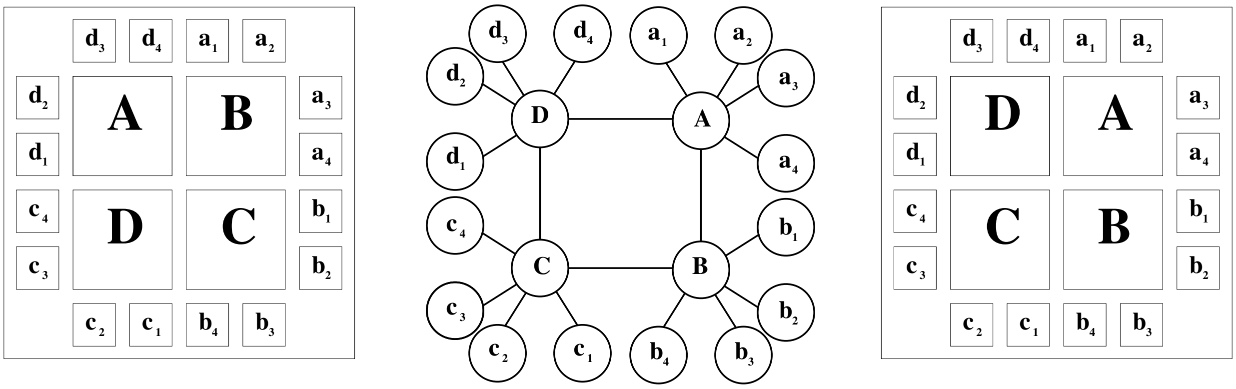

Figure 4-1: Example of a Local Placement Minima

As mentioned in Chapter 2, limiting swap acceptance for a design to only those swaps that improve placement cost tends to lead to a final placement that is far from optimal. Consider, for example[2], the circuit and sample placements shown in Figure 4-1. In this example, the circuit in the center of the figure is placed at left in a local cost minimum. No individual swaps of perimeter I/O pads or internal logic blocks in this configuration will result in an improvement in overall cost in terms of wire length,

[1]Feasible here indicates each physical logic block contains at most one design block.
[2]This example is taken from [30].

thus ending the iterative placement process if only swaps that reduce cost are accepted. This placement can be seen to be far from optimal, however, as the placement at the right clearly has lower overall wire length and is more likely to require fewer routing resources to achieve a successful route than the placement on the left. To avoid local cost minima, like the one shown in the figure, there is a need for simulated annealing to occasionally accept logic block swaps that increase overall cost. By accepting these moves, the global placement can be moved away from a local minimum enhancing the prospect that further cost-reducing swaps may find a more optimal final placement.

An important aspect of the simulated annealing algorithm is the determination of how frequently cost-increasing swaps are accepted. For most algorithms, this acceptance rate is determined based on a probability, $e^{-\frac{\Delta C}{T}}$, where ΔC is the swap cost increase and $\mathbf{T}$ is the *temperature*, a probability parameter which directly controls the acceptance rate. Initially, $\mathbf{T}$ is set to a high value so that almost all swaps, good and bad, are accepted. During progression of the algorithm, $\mathbf{T}$ is repeatedly reduced and fewer higher cost permutations are accepted, thus allowing convergence to a final result. Important factors that effect the run time and quality of simulated annealing algorithms are the determination of starting temperature $\mathbf{T}$, adjustment of $\mathbf{T}$, number of permutations attempted at each $\mathbf{T}$, and the ending criteria for the algorithm. These parameters have been the subject of a great deal of research and are reviewed in subsequent sections.

Following an elaboration of each of these parameters, an experimental analysis of the relationship between the run time of simulated annealing and FPGA routability is made. This analysis is performed by running simulated annealing for various lengths of time and then evaluating placement quality in terms of overall wire length and routability. It is shown that as design sizes scale, the amount of annealing time needed to achieve the same relative quality of placement increases non-linearly.

4.2 Simulated Annealing Formulation

While individual implementations of simulated annealing subtasks, such as starting temperature determination and temperature adjustment, vary, the flow of a typical annealing formulation, shown in Figure 4-2, is constant across most implementations. The italicized subtasks in the figure have been the subject of extensive research with regard to placement for FPGAs and other design problems. For the evaluation performed in this chapter, previously-tested subtask approaches are employed to create an understanding of the growth rate of annealing time versus design size for the five benchmark circuits previously used to evaluate the growth rate of routing time relative to design size. While a comprehensive review of all previous annealing implementations is beyond the scope of this chapter, a brief review of the important issues regarding annealing is provided subsequently.

T = *StartingT()*
Moves_per_iter = *MovesPerIter()*
While (*StoppingCriterion(T)* == FALSE)
    Move_count = 0
    While (Move_count < *Moves_per_iter*)
        Swap blocks
        Evaluate $\Delta\, cost$
        If $\Delta\, cost < 0$
            Accept swap
        Else if (random(0, 1) $< e^{-\frac{\Delta\, cost}{T}}$)
            Accept swap
        Else
            Reject swap
        Move_count++
        EndWhile
    T = *AdjustT(T)*
EndWhile

Figure 4-2: Simulated Annealing Algorithm

Cost Function

An important issue for simulated annealing is the cost function used to evaluate the quality of the global placement. In most cases, including the following experimentation, the overall placement wire length, determined from net bounding boxes, is used to judge placement quality. Several other cost metrics have recently been used with simulated annealing for FPGAs with varying success. In [12], a cost function was formulated which explicitly took into account the demand versus supply of routing resources for small regions of the target device. Interestingly, it was found that this explicit evaluation of routing congestion achieved very little improvement in reducing W_{min}, the minimum channel width needed to route the design, at the cost of more than an order of magnitude additional computation time. In [41], annealed placement and maze routing were combined into a single placement formulation. In this case, the cost function for annealed placement swaps was derived from the number of unrouted design nets and the desired minimum clock period of the circuit. While this approach created layouts with about 20% better performance than the discrete layout flow of placement followed by routing, run times for even small designs of 200 logic blocks measured over 3 hours, effectively making the approach non-scalable. In [60], Trimberger mentions that in addition to wire length, Xilinx uses an alignment cost metric to stack blocks driven by high fanout signals into rows and columns so that long lines may be more efficiently used. This optimization is not used in the experimentation described in this chapter but could be added in future experiments at the cost of additional placement evaluation time.

Start Temperature

For the following experiments, the start temperature (StartingT) is determined using an approach first developed by Huang [31]. An initial random placement of all design logic blocks and I/O pads (of total count N_{blocks}) serves as a start point for evaluation. Determination of starting temperature commences by performing N_{blocks} random pairwise swaps of these blocks and pads. The initial annealing temperature is then set to 20 times the standard deviation of the cost for these moves. This design specific procedure creates an initial temperature that accepts of high percentage of swaps ($> 98\%$) in the initial stages of the annealing algorithm.

Annealing Schedule

In [58], it was determined that it is desirable to keep the percentage of swaps accepted at each temperature, R_{accept}, as close to 0.44 as possible. In [14], a flexible annealing schedule (AdjustT) based on R_{accept} was developed. Changes in **T** based on this value are shown in Table 4.1. For acceptance rates around 0.44, the temperature is reduced by only a small amount to encourage additional searches in the current range. If most pairwise block swaps are accepted, the temperature **T** is reduced by half. Finally, if few swaps are accepted, the temperature is reduced moderately towards a final termination point.

Fraction of moves accepted R_{accept}	α
$R_{accept} > 0.96$	0.5
$0.8 < R_{accept} \leq 0.96$	0.9
$0.15 < R_{accept} \leq 0.8$	0.95
$R_{accept} \leq 0.15$	0.8

Table 4.1: Temperature update schedule [14]

Moves Per Iteration

The ideal default number of moves at each temperature was determined in [58] to be $10N_{blocks}^{4/3}$. In [14], Betz suggests that the leading constant, 10, can be scaled to tradeoff placer run time with quality. In the following section this approach for trading off run time for quality is compared with other approaches, such as reducing the annealing start temperature and changing the annealing schedule.

Annealing Stopping Criterion

As suggested in [14], annealing is terminated when the annealing temperature **T** is less than $0.005 \times \frac{Cost}{N_{nets}}$. Additional annealing iterations for temperatures less than this value have been shown to add little to cost improvement.

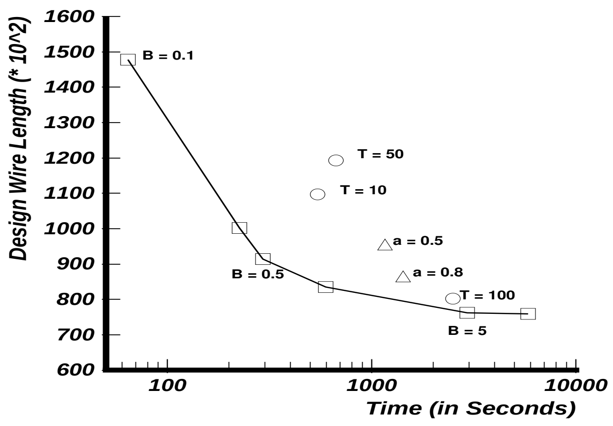

Figure 4-3: Variation of Placement Cost with Placer Run Time - Example ssp64

4.3 Experimentation

Simulated annealing has been widely studied to identify combinations of annealing parameters that create the best final quality result. Little work has been done, however, in quantifying how long it takes simulated annealing to converge to a given placement quality as design sizes scale. In this section, this growth rate is quantified experimentally for the same set of benchmarks that were evaluated in Chapter 3 with regard to wire length growth by modifying the annealed placer of the VPR toolset [14] to accept modified annealing parameter values.

Generally, longer simulated annealing runs result in improved placement quality relative to the specified cost function, in this case, design wire length. Annealing run time can be controlled by varying the annealing parameters that determine the number of logic block swaps and the swap acceptance rate. The key to evaluating the tradeoff between run time and placement quality is identifying the mix of parameter variations that result in the best run time/cost reduction curve. In an experimental evaluation, three annealing parameters, starting temperature **T**, annealing schedule, and moves per iteration were varied over a number of designs for a set of parameter values. A representative example of the results of this evaluation for design ssp64, detailed in Table 3.6, is shown in Figure 4-3.

All run time results were obtained using a 140 MHz UltraSparc 1 with 288Mb of memory. Unless otherwise stated, annealing parameters are set to the default values discussed in the previous section. Deviations from these defaults are represented by the following graph points:

1. Trials which involve the modification of the leading constant, β, of $\beta N_{blocks}^{4/3}$ moves per iteration are represented as squares.

2. Trials with constant starting temperature values **T** are represented as circles.

3. Trials with a fixed annealing schedule constant, α, are represented with a triangle.

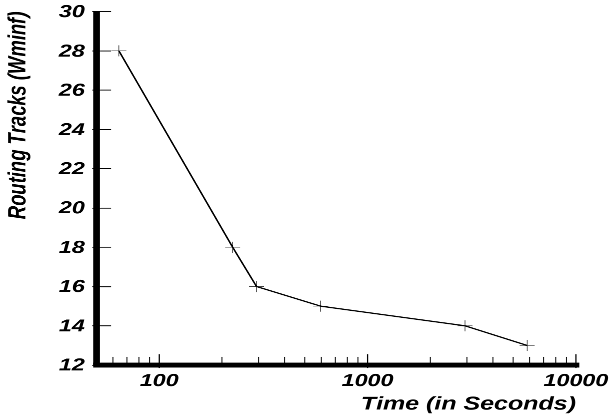

Figure 4-4: Variation of W_{minf} with Placer Run Time - Example ssp64

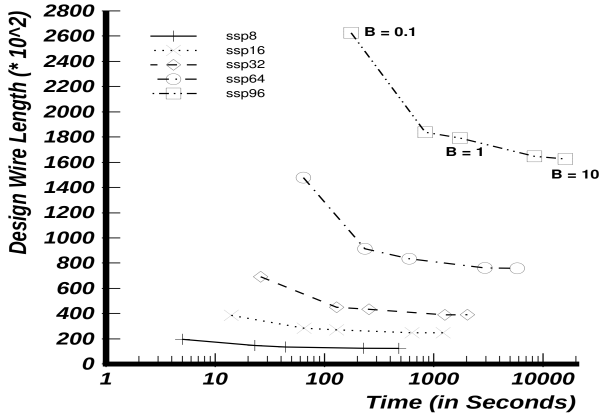

Figure 4-5: Variation of Wire Length with Placement Run Time - ssp Designs

It can be seen from the graph, that variation of the number of moves per iteration through modification of β is the most effective way to trade off placement run time for placement quality in terms of overall wire length. This observation makes intuitive sense since the starting temperature and annealing schedule are design dependent and are tuned dynamically during execution of annealing while moves per iteration remains the same across all designs. By modifying β, the search space at each temperature is narrowed, but roughly the same set of temperatures are visited as the more exhaustive case.

While evaluation of placement cost in terms of wire length gives some design quality insight, the real goal of extended placement time is reducing the amount of time needed to route a design or making design routing feasible for a target device. To measure routability in terms of track count, a second quality metric, the minimum track count that can be successfully routed in 60 seconds or less of routing, W_{minf}, is evaluated for various annealing trials. In Figure 4-4, it can be seen that the curve of W_{minf} generally follows the shape of the curve in Figure 4-3.

Now that a procedure has been established to allow for tradeoffs between annealing run time and

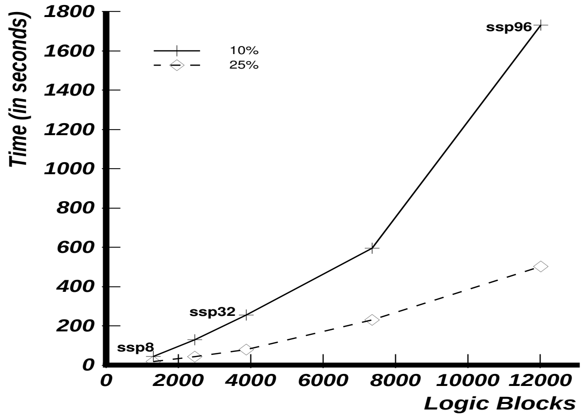

Figure 4-6: Anneal Run Time / Quality Tradeoffs

placement quality, it is possible to analyze the amount of placement time necessary to achieve a specific placement quality level across a range of designs. To perform this analysis, the five ssp benchmarks listed in Table 3.6 were placed using five different values for β, (0.1, 0.5, 1, 5, 10). The run time versus wire length results for the five benchmarks are shown in Figure 4-5. The nature of the curves indicate that increasingly long lengths of placement time are needed to achieve overall wire length results close to the best possible ($\beta = 10$) for the designs. In Figure 4-6, the result of experiments to quantify this placement time growth is shown. By varying β, each design was placed for the length of time necessary to achieve a wire length cost within 10% and 25% of the wire length cost that could be achieved with $\beta = 10$. The plots in the figure show that an increasingly large amount of time is needed to achieve these quality levels as designs sizes grow from small to large. This is not unexpected since the number of moves per iteration, $\beta N_{blocks}^{4/3}$, also increases exponentially for a given β value as the designs scale.

Chapter 5

Macro-based Placement and Routing

As stated previously, most large FPGA designs used in multi-FPGA systems consist of a collection of large macro-block components that can be used across a broad range of designs. While it is difficult to generalize across all applications, this design structure would appear to provide an opportunity to extract layout information from a pre-synthesized design library, potentially leading to significant layout time reductions. In Chapter 3, it was observed that by adding approximately 40% additional routing tracks to an FPGA device, it was possible to reduce routing time to less than 60 seconds for a number of large benchmark designs. The placements for these designs, however, had been created using simulated annealing iterations that required many minutes to complete. Through the use of a macro-based floorplanner described in this chapter, it is found that with about 50% additional tracks per channel, place *and* route times of approximately 60 seconds can be achieved with reasonable device logic utilization (about 65%) for design containing thousands of logic blocks.

The Frontier FPGA floorplanning system, developed for this dissertation, considers FPGA designs to be interconnected collections of macro-blocks. This floorplanner quickly achieves a high-quality placement, in which each physical FPGA logic block contains at most one design logic block and each design logic block is assigned to at least one physical block. To a large extent, the floorplan placement quality, in terms of the amount of routing resources needed to successfully route a design, varies significantly from design to design depending on design global communication patterns and the shape and construction of the macro-blocks. Just as it was necessary to consider tradeoffs between quantities of routing resources and routing time for the fast router in Chapter 3, it is necessary to consider tradeoffs between the amounts of routing resources, routing time, *and placement time* for the macro-based floorplanner. It will be shown in this chapter that an FPGA floorplanner will not necessarily achieve a more routable placement result than fine-grained simulated annealing unless a target application exhibits a regular communication pattern or contains macros with special internal structure that is difficult to find through annealing. A clear benefit of the floorplanner, however, is that it is possible to quickly achieve placements in a few seconds that, while not optimal, are comparable in quality to placements that require several minutes or more of fine-grained annealed placement.

In an effort to take advantage of design reuse and localized placement optimization, the floorplanning approach developed here considers each RTL component as a pre-placed library element with rectangular shape containing pre-placed logic blocks and (in some experiments) pre-routed internal nets. In general, input logic designs are considered to be collections of macro-blocks, interconnected in arbitrary two-dimensional communication patterns, rather than being limited to linear or two-dimensional arrays with primarily nearest-neighbor connectivity. Bin packing of rectangular shapes in a square region has been explored for some time with respect to full-custom ASIC design [52]. The floorplanning approaches implemented here are similar to ones previously developed, in many respects, with important additions to allow for integration in a place and route system that has the following goals:

1. The floorplanner attains a feasible placement in a few seconds of comparable quality to a placement attained through annealing that takes several minutes or longer.

2. The place and route system allows for a routability evaluation of an initial floorplan so that it can be determined if iterative routing is likely to converge quickly. It was shown in Chapter 3 that a single iteration of a fast router can serve this role. If routing will not complete successfully or will not converge quickly for the given channel width of the device, as a last resort, additional placement optimization can be made through low-temperature, fine-grained annealing.

3. The place and route system can optionally allow routing resources in each pre-placed macro-block to be *isolated* from the rest of the circuit so that local modifications to placement and routing inside a macro-block can be made without the need to change logic and routing resources in other parts of the design. To achieve isolation, during floorplanning, certain wire segments in the device are reserved to carry nets internal to macro-blocks, while others are reserved to carry inter-macro nets. In Section 2.4.3, an example of *planar* isolation was shown where intra-macro routing was restricted to wire segments in specific planar sections of the device. This approach and other isolation techniques will be detailed in the next chapter. In many cases, routing isolation can be used to create completed layouts in a few seconds at the cost of low device logic and routing resource utilization.

5.1 Floorplanning and Routing System Flow

To achieve the goals stated above, the floorplanner described in this chapter has been combined with the router of Chapter 3 in two distinct user-selectable paths shown in Figure 5-1. For the path containing shaded blocks, floorplanning is considered a substitute for annealed placement discussed in Chapter 4. In this flow, no restrictions are placed on routing track assignment so that any net may be assigned to

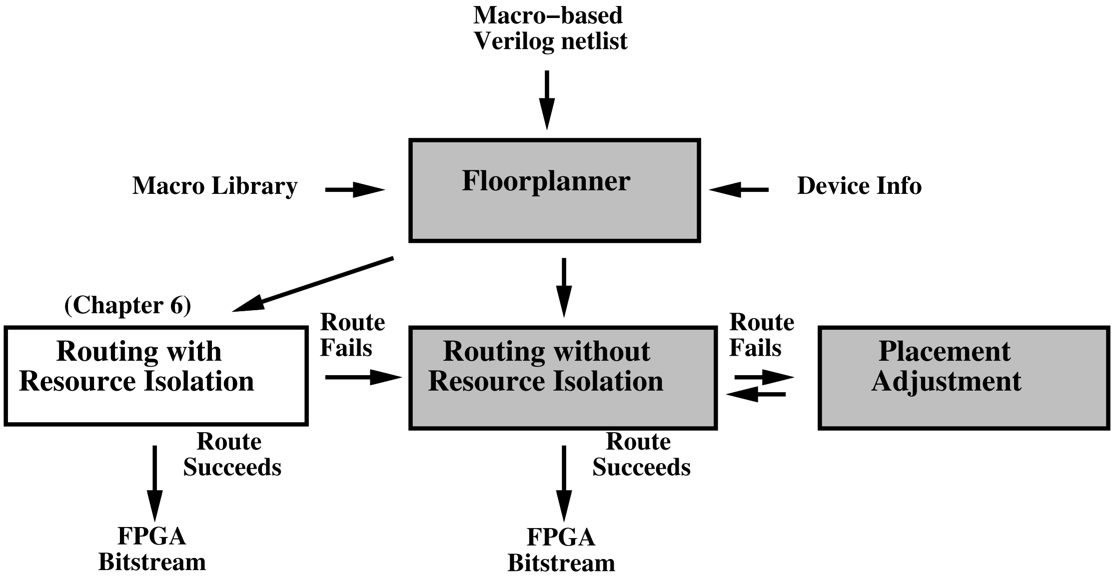

Figure 5-1: High-level Floorplanning and Routing Flowchart

any track. Included in this flow is a placement refinement step of low-temperature simulated annealing that can used in certain cases to improve design routability by eliminating localized routing congestion.

As mentioned previously, one of the benefits of using a floorplanned placement approach is that intra-macro routing may be isolated from inter-macro routing. This isolation requirement leads to additional routing constraints that are discussed in Chapter 6. The unshaded box in Figure 5-1 represents this specific step in the floorplanning and routing system.

5.2 Advantages of Macro-based Placement for FPGAs

Several specific island-style FPGA features make the pre-placement of macro-blocks a clear benefit in the overall placement process. In this section, these advantages are reviewed and their applicability to the floorplanning problem is discussed.

Local Minimization of Wire Length

In Section 3.4.1, the relationship between the number of logic blocks in a piece of a design and the number of wires emanating from the piece was determined through application of Rent's Rule. By Equation 3.5, this relationship was determined to be exponential for circuits with Rent exponents of greater than 0.5. Frequently, large macro-blocks, such as multipliers or ALUs have significantly less external I/O per macro-block than would be predicted via Rent's Rule. For these macros, optimized macro placement prior to floorplanning can be used to take advantage of the relative locality of communication inside the macro by achieving minimum-cost local placements.

Bit-slice Alignment

Many datapath circuits have a regular placement structure that is difficult to find using generalized algorithms, such as simulated annealing. Specific examples include bit-sliced arithmetic and logic functions (adders, comparators, etc) and memories. By stacking bit slices horizontally or vertically, control signals, such as clock enables, used by all individual bits, may be fanned out to a number of logic blocks via a single long line rather than an irregular maze of single-length segments. Additionally, neighboring datapath blocks with the same bit pitch can be easily aligned to minimize inter-macro wiring.

Most commercial FPGA devices contain special circuitry inside logic blocks, in addition to lookup tables and flip-flops, to perform functions such as fast generation of ripple carries [4] [3]. Frequently, for these features to be used, multiple bits of the datapath macro must be aligned in a specific linear pattern to allow for dedicated inter-logic block interconnection to occur. For example, in the Xilinx XC4000 series of devices, carry circuitry requires bit slices of an arithmetic macro to be aligned vertically to allow for near neighbor communication that is out-of-band from the general interconnect grid shown in Figure 2-2.

While small datapath blocks (like adders and ALUs) have bit-slice regularity, larger blocks such as complex multipliers do not, making it difficult to generalize bit-slice regularity into floorplanning. Several previous floorplanning approaches, outlined in Section 5.3.1, have restricted designs to only bit-sliced macros, greatly simplifying the placement problem, but limiting their applicability to more general macro-based designs.

Pre-routing of Nets

Application of simulated annealing to a flattened design destroys design regularity so that previously determined intra-macro routing information becomes useless. The use of macro-blocks isolates placement so that portions of intra-macro routing, stored in a design library, may be quickly identified and utilized. Effective manipulation of these *pre-routed* nets can reduce overall layout time by requiring fewer design nets be routed. An algorithm is presented later in the chapter to select internal macro-block nets for pre-routing in an effort to accelerate overall design route time.

5.3 Macro-based Floorplanning

General Macro-based Floorplanning Formulation

As seen in Figure 5-2, the floorplanning problem effectively adds shaping constraints to placement that were not present during the discussion of fine-grained FPGA placement in the last chapter. The floorplanning problem can be stated formally through a set of constraints [52]. Let $B_1, B_2, ..., B_n$ be the macro-blocks to be placed in the array. Each B_i has associated with it a height h_i and width w_i in

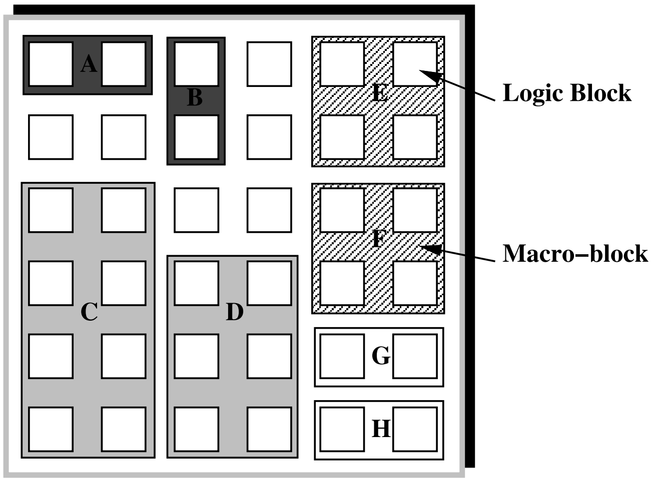

Figure 5-2: Macro-block Design Style

multiples of logic blocks. The set $N_1, N_2, ...N_m$ is the set of inter-macro nets that connect the macro-blocks, each having length $L_1, L_2, ..., L_m$. The placement problem may be defined as the need to find a placement rectangle $R = (R_1, R_2, ..., R_n)$ for each of the blocks in the array such that:

1. Each macro-block can be placed in rectangle R_i which has dimensions height h_i and width w_i.
2. No two macros overlap such that $R_i \bigcap R_j = \phi$.
3. The total area of the bounding rectangle R and total wire length $\sum_{i=1}^{m} L_i$ are minimized.

In general, the flat routing structure of island-style FPGAs makes the floorplanning problem for these devices similar to that previously addressed for full-custom VLSI design styles. A large number of approaches have been developed to pack rectangular macro-blocks into a square plane. Analagous to fine-grained FPGA placement, two basic types of floorplanning approaches exist, constructive and iterative [52].

Like the simulated annealing algorithm presented for fine-grained FPGA placement, iterative algorithms for floorplanning typically involve an extensive search of the implementation space to identify a near-optimal, feasible[1] floorplan with regard to wire length and performance objectives. All macros are considered simulataneously and an initial floorplan typically contains these blocks in a tight cluster that contains some macro-block overlap. As blocks are moved apart to eliminate overlap, additional cost factors such as performance and overall wire length are taken into account until the algorithm converges to a feasible floorplan with no logic block overlap. Iterative algorithms differ in the way new floorplan states are created. An typical iterative approach is force-directed placement [29], in which interconnectivity and overlap between blocks is represented by spring-like forces similar to those found in a mechanical

[1] Again, feasible here indicates that each physical logic block contains at most one design block.

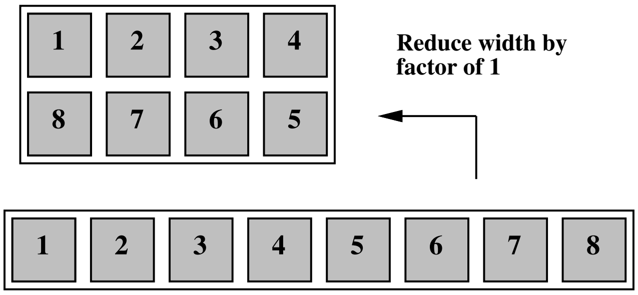

Figure 5-3: Macro Reshaping

network. As each block is moved to eliminate overlap and reduce wire length, these forces are adjusted. This process continues until a feasible, steady-state block configuration is found.

Other floorplanning approaches, based on constructive approaches, optimize small pieces of a floorplan locally and then combine partial floorplans together to form a final, feasible solution. Constructive approaches converge very quickly since not all primitive blocks are considered simulataneously by the algorithm. It is this class of algorithm that forms the basis of the floorplanning approach presented in this dissertation.

5.3.1 Previous Macro-based Floorplanning for FPGAs

Two previous approaches [32] [53] have been developed for island-style FPGAs, both targetted to the Xilinx XC4000 family. Each of the previous floorplanning algorithms takes significant advantage of data-bit alignment to converge to a placement quickly. Neither of these approaches, however, appears to be applicable to large designs containing macro-blocks that do not have this data-bit regularity, significantly limiting their application space. In [32], a floorplanning technique for one-dimensional datapath designs based on topological sorting was presented. This approach evenly distributes routing demand along the length of the datapath by abutting evenly-sized macros horizontally in a linear array. While this approach works well for regular designs, such as multipliers made of a series of adders, this approach is not extensible to two-dimensional designs.

In more recent work, [53] presented a method based on iterative force-directed floorplanning targetted to designs with irregular communication patterns. In this approach, macro-blocks are relaxed from an initial placement using force-directed analysis until design wire length is minimized in the placement plane, even if some macro-block overlap remains. This block movement step is followed by a subsequent *macro reshaping* step to eliminate overlap. As shown in Figure 5-3, bit-sliced stacks of datapath functions, like adders and comparators, are folded to reduce one macro shape dimension at the cost of the other. This reshaping process continues iteratively, along with local macro placement perturbation, until a feasible floorplan is reached. While this approach appears to work well for the small designs listed in [53], it is not clear how general the outlined approach is for macros that do not have internal bit-slice regularity. Additionally, the run time for just the force-directed portion of the algorithm was reported to be over 90 seconds for several benchmark designs and 5 hours for one of them [53], violating our goal of achieving a fast, feasible placement in less than a minute.

Both of the floorplanning techniques described above rely on specific design structure (a linear array of inter-macro connections for [32] and bit-sliced macros for [53]) in order to achieve fast, efficient placements through floorplanning. A goal of the floorplanning work in this thesis is to remove these limitations to explore the more general case of floorplanning designs with potentially unstructured inter-macro communication and arbitrary intra-macro interconnection. Previous, structured-interconnect designs can still be handled as special cases.

5.4 Limitations of Macro-based Floorplanning for FPGAs

It was shown in Chapter 3, that FPGA routing becomes much easier for island-style devices as the number of routing tracks per channel in the target FPGA is increased above W_{min}, the minimum number of tracks per channel needed to route the design. Given the limited VLSI area available in commercial FPGA devices, however, most commercial offerings set track counts narrowly, requiring multiple, long routing iterations to achieve a successful route for most designs and in some cases, requiring less that 100% logic utilization to achieve successful route completion. In general, as the size of a macro-based design is expanded to fill a greater number of logic blocks in an FPGA device, it becomes increasingly likely that the feasible placements created through floorplanning will be *less routable* than those that can be achieved using simulated annealing. This is due to the fact that possible macro-based floorplan configurations are limited by macro-block shapes and internal macro logic block placements that were created without specific knowledge of the communication patterns of the high-level circuit under consideration. A goal in performing fine-grained annealed placement was to avoid localized placement cost minima, in terms of wire length, in favor of a generally more routable global minimum. By packing pre-placed shapes together, frequently a localized placement cost minimum is achieved instead of the global minimum that could be achieved through annealing. Effectively, the benefits achieved though bit-slice macro structure and local minimization of wire length may be lost to increased inter-macro and total design wire length. A notable exception to this observation are applications with regular interconnection patterns of inter-macro communication, such as those that can be structured as a 2-D mesh or linear array. Since the circuits under consideration here are generally not regular graphs, typically, it is difficult to optimize inter-macro ports to appropriately fit all possible versions of circuits while still allowing macro-blocks to be packed in a space-filling manner. While many macros have a well-defined structure and exhibit limited inter-macro communication, some do not, potentially rendering their local placement non-optimal in regards to global circuit communication.

In an FPGA computing system, a target FPGA device has a fixed wire segment channel width

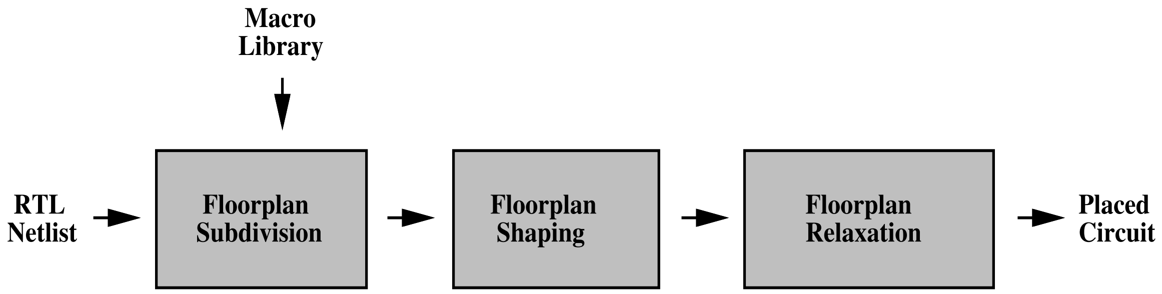

Figure 5-4: Frontier Floorplanner Flow

measured by W_{FPGA}. In our new CAD tool, Frontier, if no feasible floorplan can be found that will successfully achieve route completion at the target channel width of the FPGA, placement modification in the form of low-temperature annealing can be performed to allow for design smoothing at the cost of additional placement time. The evaluation of routability for the candidate floorplan triggering this step can be made quickly by using an iteration of the router, as described in Chapter 3.

5.5 Macro-based Floorplanning Implementation

To achieve the desired goal of fast convergence to a feasible placement, a constructive, macro-based floorplanning tool has been developed. This floorplanner consists of three distinct placement steps which are outlined in Figure 5-4.

In the first step, *floorplan subdivision*, either of two user-selectable approaches can be applied to hierarchically isolate small groups of macro-blocks that have high connectivity. Traditionally, for ASIC floorplanning, recursive bipartitioning has been used to perform this role at the expense of increased amounts of run time as problem sizes scale. In addition to bipartitioning, the Frontier floorplanner also supports macro-block clustering as an approach to subdividing macro netlists. This algorithm runs in $O(n)$ time and takes both size and connectivity of blocks into account to create results nearly as good as bipartitioning in a fraction of the time.

Subdivision is followed by a *floorplan shaping* step in which partial floorplans are created from the divided groups of the previous step by abutting macro-blocks or previously created partial floorplans together to form larger partial floorplans and eventually a final feasible design floorplan. A cost function based on wire length and macro-block area is used to evaluate candidate combinations. These costs, in combination with a novel technique for evaluating and storing candidate partial floorplans, is used to quickly prune out undesirable configurations.

As a final floorplanning step, an attempt is made to relieve congestion in areas between macro-blocks by identifying routing hot spots in the floorplan and opening additional routing pathways through *floorplan relaxation*. This optimization relieves localized congestion while maintaining the basic interconnection pattern of the floorplan.

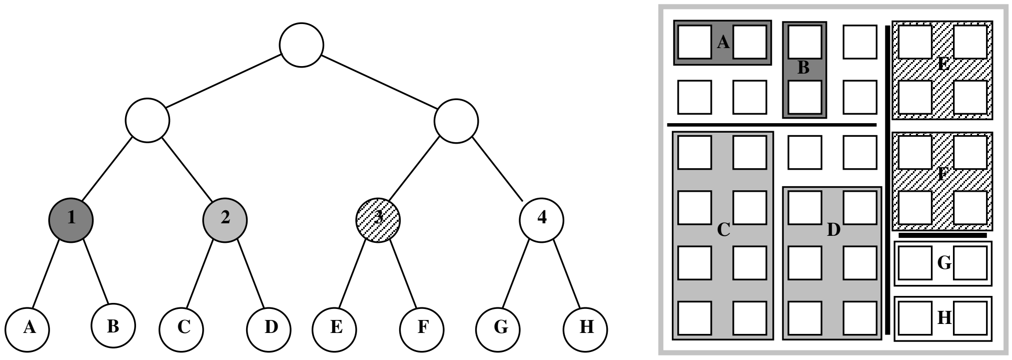

Figure 5-5: Floorplan Slicing Tree

Macro-block layout takes place prior to design floorplanning. Each design macro-block has a specific shape set by the designer. User-defined locations for inter-macro I/O signals and data busses along the border of the macro are used to guide internal alignment of bit-sliced busses either by hand or through the use of annealed macro-block placement. In subsequent experimentation, only one shape was considered for each macro, although many different sizes and shapes may be considered in the floorplanning algorithm during the shaping phase. A macro-block layout stored in a library may be rotated or mirrored, as necessary, in evaluating design floorplans.

5.5.1 Floorplan Subdivision

In this initial phase of floorplanning, the placement goal is to subdivide the initial user netlist of macro blocks into a *slicing tree* [43]. As seen in Figure 5-5, a slicing tree is effectively created by first identifying portions of the design that are tightly connected and then by isolating them hierarchically. Each node in the tree represents a partial floorplan. To allow for full utilization of logic resources, all nodes at given level of the tree should have approximately the same number of logic blocks to balance partial floorplan area.

Two different approaches to hierarchical isolation of macros are supported by the floorplanning system. The effect of using these two approaches are contrasted via results in Section 5.6.3.

Weighted Clustering

A fast, straightforward way to create a slicing tree is through the repeated application of clustering. Simply selecting blocks on the basis of connectivity, however, may lead to partial floorplans at the same slicing tree level that vary significantly in terms of size. A better way to ensure a balanced floorplan is to group together slightly less connected blocks of about the same size to balance the tree [62]. This is

B: Initial set of design macro-blocks.
M: Set of macros or macro clusters to be combined.
SizeM: Number of elements of M.
Add elements of B to M.
While SizeM > 1
  Loop over all M elements.
  Select pair M_i, M_j such that C_{ij} maximized.
  Remove M_i, M_j from M.
  Add macro cluster $M_i \bigcap M_j$ to M.
  Update connectivity.
EndWhile

Figure 5-6: Weighted Clustering Algorithm

accomplished through the use of a cost function weighted to take logic block counts and interconnectivity into account:

$$C_{ij} = \frac{B_{all}}{B_i + B_j} \times \frac{min(B_i, B_j)}{max(B_i, B_j)} \times \sum N_{ij} \tag{5.1}$$

where B_{all} is the total number of logic blocks in the circuit, B_i and B_j are the number of logic blocks in the macro-blocks i and j under consideration and N_{ij} are the nets connecting i and j. The first term in the cost function prevents a specific cluster from becoming too large in relation to the rest of the circuit. The second term prevents two macros with vastly different numbers of blocks from being connected together thereby creating area inefficiencies, and the last term measures connectivity. As shown in Figure 5-6, the above cost function is used as the cost metric of a recursive algorithm to create a bottom-up slicing tree.

Recursive Bipartitioning

An alternate way to create a slicing tree is to iteratively apply bipartitioning to a macro-based netlist until only single macro-block nodes remain. For the Frontier floorplanner, a partitioner based on the classical F-M mincut algorithm [33] [25] is used to create approximately balanced partitions in terms of fine-grained logic blocks while minimizing cut bandwidth. Generally, this top-down isolation approach has been considered preferable to bottom-up clustering since global communication patterns are considered earlier in tree creation for bipartitioning rather than later for clustering [38]. With clustering, locally-optimal choices made at the bottom of the tree may have adverse effects on minimizing communication at the top of the tree where inter-node bandwidth in generally greater.

Rather than simply minimizing the cut-width at each level of the tree, it would be preferable to

Mirror or rotate child node to create permutation, P.
Determine minimum dimension of P, *mindim*.
Determine maximum dimension of P, *maxdim*.
Determine wire length L of P from Equation 5.2.
Get candidate bin β associated with *mindim*.
Determine permutation cost, $c(P)$, from Equation 5.3.
Get maximum cost permutation, $maxP$, from β.
If $(c(P) < c(maxP))$
    Replace $maxP$ in β with P.

Figure 5-7: Evaluation for One Internal Node Shape Permutation

minimize the *maximum* cut-width across all levels so that the bandwidth inside the device becomes balanced to meet the routing density limitations of the target device. However, optimization for this objective has been found to be NP-hard even for simple graphs [38] and currently no efficient heuristics exist to address the issue.

5.5.2 Floorplan Shaping

In this step, a number of different floorplan configurations are created by performing a bottom-up, post-order traversal of the generated slicing tree. For each internal node of the tree, the shapes of its children are used to create new candidate partial floorplans. While the application of this type of dynamic programming approach to determining a feasible floorplan has been used in a number of systems [43] [38], the cost function and data structures used to evaluate partial floorplans and store intermediate results in Frontier differs from others due to the prespecified requirement of fast compilation. For most previous slicing-based systems [38] [62], a near exhaustive evaluation of possible node shapes is performed at each intermediate node to create a wide range of candidate partial floorplans. In the Frontier system, intermediate shapes are aggressively pruned to limit the floorplan search space to only those configurations most likely to lead to minimized placement cost.

In the new floorplanning system, shape and wire length are considered together in creating intermediate partial floorplan permutations for each internal node. Candidate permutations are created by abutting child shapes horizontally, each of which may be rotated or mirrored, to form up to 64 possible permutations per pair. Note that vertical abutment is not needed since it is created implicitly at the next, higher level node of the tree (the parent node of the current node under evaluation) when each horizontal child permutation is rotated by 90 degrees. Permutation wire length L is determined as:

$$L = \sum_{N_{ij}} q \times [bb_x + bb_y] \tag{5.2}$$

where N_{ij} are the number of nets interconnecting the two child shapes, bb_x and bb_y are the horizontal and vertical extents of the net bounding boxes and q compensates for the fact that the bounding box underestimates the wiring necessary to connect a net to a multi-fanout connection [17].

If all possible shape and wire length combinations were considered exhaustively, the number of possible shape permutations would grow exponentially as the tree is traversed bottom-up. To avoid this growth, pruning of inferior permutations is necessitated using steps outlined in Figure 5-7.

Partial floorplan cost information is partitioned into a series of bins keyed by the dimension of the shortest side of a candidate permutation. The linear cost function:

$$c(P) = \lambda \times L + \nu \times maxdim(P) \tag{5.3}$$

is used to compare candidate permutation cost with the cost of others already stored in the bin. Through experimentation, it was found that setting scaling factors λ and ν to 1 generated best results by emphasizing permutation shape for nodes near the bottom of the slicing tree and wire length for nodes near the root.

By following this bottom-up dynamic programming approach, a variety of feasible shape combinations can be considered and wiring- and area-inefficient combinations can be quickly eliminated. Only shapes that fit within the square perimeter of the target device need be evaluated for routability. The rate at which pruning of feasible shapes takes place at each internal tree node varies depending on the user-specified, maximum number of bins supported by the algorithm and the maximum number of permutations stored per bin.

At the end of the shaping process, a number of feasible final floorplans are available for consideration. The feasible floorplan with the smallest overall wire length is chosen for subsequent optimization and implementation.

5.6 Combined Floorplanning and Routing System

At the beginning of this chapter it was mentioned that two design flows are possible with Frontier, one that isolates intra-macro routes on certain device tracks (the **isolated** case discussed next chapter) and one that allows any net to be routed on any track (the **non-isolated** case discussed subsequetly in this chapter). For the latter case, floorplanning may be viewed as a substitute for fine-grained annealed placement. To evaluate the quality of the floorplanner, for the remainder of this chapter additional floorplanning and routing steps, specific to the *non-isolated* routing flow and shown in Figure 5-8, are presented and then analyzed through experimental results. A brief summary of these steps appears below:

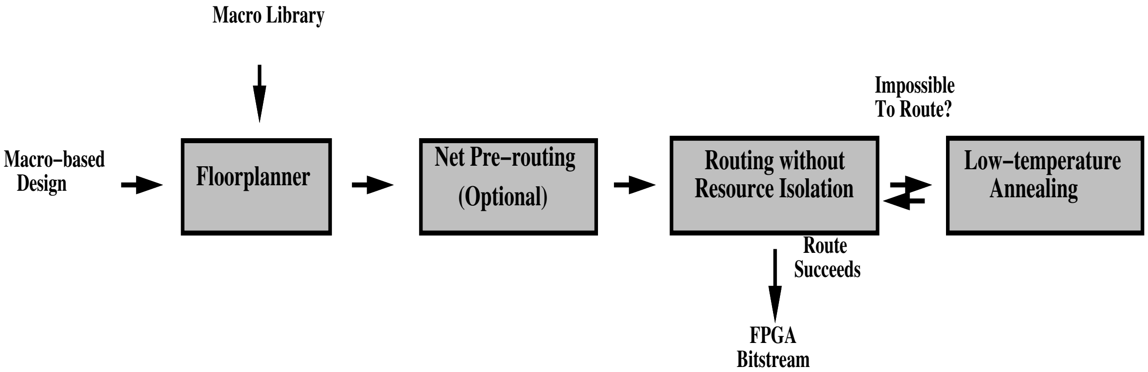

Figure 5-8: Integration of Floorplanning and Non-Isolated Routing

- **Floorplanning** - Frontier takes as input the logic block array size of the device, a netlist of instantiated macro-blocks, and a design library containing pre-placed macro shapes. As an initial step, the floorplanner quickly creates a feasible placement using the steps outlined in Section 5.5. A final floorplan relaxation step for the non-isolated routing flow is presented in the next section.

- **Net Pre-routing** - In this step, design nets internal to design macros are evaluated to determine if their net routes should be leveraged from routing information in a design library or if they should be routed from scratch in a subsequent design routing phase.

- **Design Routing and Route Evaluation** - In this step, a single iteration of the fast router is applied to the floorplanned design. As discussed in Chapter 3, at the conclusion of a single iteration of routing for the floorplan, the routing problem can be defined as either low-stress, difficult, or impossible. If the routing problem is either low-stress or difficult, the router is allowed to run to completion. Otherwise, a subsequent placement modification step, low-temperature annealing, is performed.

- **Low-Temperature Annealing** - In Frontier, if a floorplanned design is designated as impossible-to-route, the floorplan is used as an initial placement for low-temperature, fine-grained annealing. This floorplan modification step converts the impossible-to-route floorplan into a routable one, at the cost of additional placement time, by breaking up macros into their individual logic block components and performing pairwise logic block swaps.

5.6.1 Floorplan Relaxation

For both the fine-grained placement algorithm outlined in Chapter 4 and for the macro-based floorplanning algorithm presented in Section 5.5, overall design wire length is used as a metric to measure design routability. While this metric is easy to calculate and is needed to differentiate between numerous different floorplan permutations, a more desirable routability metric would measure *localized* congestion

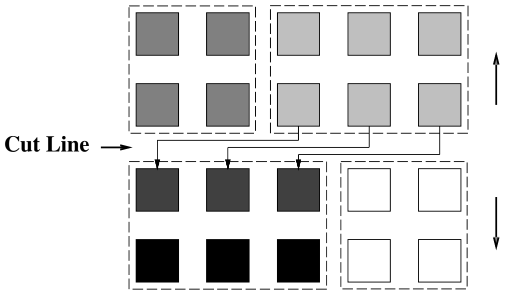

Figure 5-9: Inter-Macro Congestion

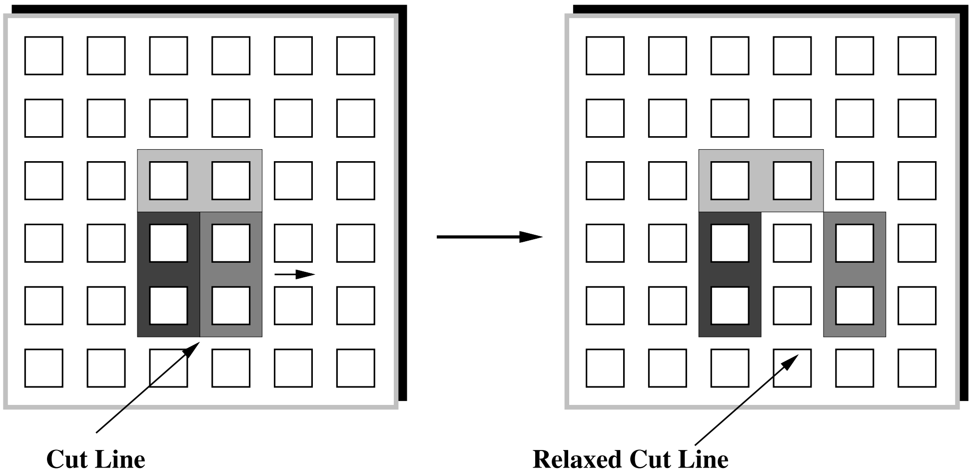

Figure 5-10: Floorplan Relaxation

in the floorplan since routing failure in even one small area of the FPGA device leads to overall routing failure. As an example, consider the partial floorplan shown in Figure 5-9. Here, a three-bit bus is sourced by logic blocks in the macro at the top, right and terminates at logic blocks in the macro at the bottom, left. In this case, for shortest path connections, all bits of the bus must use tracks in the routing channel along the cut line. If the macros are separated vertically by one block, the overall wire length of the design is increased, but the routability of inter-macro connections is improved, not worsened. Frequently, as seen in Figure 5-10, after floorplanning, unused logic blocks are located along the border of the device. As a means of promoting routability, additional space can be inserted between cut lines, broadening inter-macro channels and forcing macro-blocks into under-utilized area, while keeping relative macro positioning intact. Given the large number of wire segments in devices and the flat routing structure of devices, an evaluation of localized routing congestion between macros is very difficult to perform analytically, especially as devices scale.

One of the ideas explored with Frontier that met with limited success was the use of a single iteration of the fast router from Chapter 3, not only as a routability predictor, but also as a means to identify congested areas around cut lines. Where feasible, additional routing space could then be allocated between cut lines by moving macros apart by a single logic block, as shown in Figure 5-10. Routing

Order cut lines by congestion density.
While more cut lines to be relaxed.
  Select highest cost cut line from ordered list.
  Insert one block gap between cut line.
  If resulting floorplan exceeds device dimensions.
    Revert to floorplan before change.
EndWhile

Figure 5-11: Floorplan Relaxation Iteration

congestion along each cut line was determined following a routing iteration by evaluating the density of used routing resources (tracks used / tracks available) in a boundary region of five logic blocks parallel to and inclusive of the cut line.

Next, cut lines were ordered from most-congested to least-congested and relaxed using the steps of the algorithm shown in Figure 5-11. Through experimentation, it was determined that while an iteration of floorplan relaxation was helpful in creating a routable design, determining cut line ordering through congestion analysis was no more effective than ordering cut lines by *cut line length*. This latter approach was much faster than the evaluation of congestion density, since it did not require an iteration of fast routing, and led to insertion of additional routing bandwidth in the center of the device, typically the most congested area.

5.6.2 Understanding Macro Pre-routing

A desirable aspect of using macro-blocks in placement is the capability to leverage pre-routed intra-macro net information that has been stored in a design library for subsequent floorplan routing. During routing, a pre-routed net is considered to be fixed to a pre-determined set of routing resources and need not be routed or ripped-up. By reducing the total number of design nets to be routed and segments to be searched, in many cases, overall routing time can be reduced in comparison to a routing search started from scratch.

For most designs, the issue of macro-based net pre-routing is complicated by the fact that both inter and intra-macro nets use the same channel-based sets of wire segments. Simply pre-routing all intra-macro nets, without allowing for net rip-up and reroute over multiple router iterations, may lead to routing congestion, making the routing problem *more* time-consuming rather than *less*. To date, no work has been reported on acceptable levels of pre-routing for island-style FPGAs. In this section, an algorithm is developed that uses a tunable pre-route density level, W_{pre}, to select a subset of hard-to-route intra-macro nets for pre-routing. Through experimentation, it is shown that pre-routing a few critical intra-macro nets is effective in reducing overall route time and is preferable to blindly pre-routing

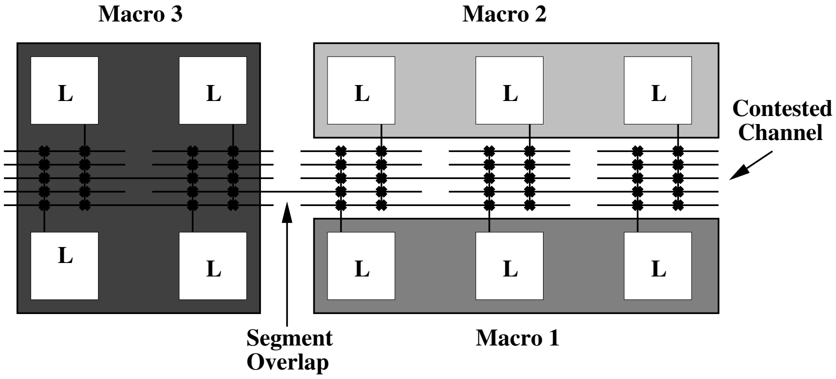

Figure 5-12: Inter-Macro Routing Contention

all internal macro nets.

Net pre-routing cannot be effectively performed without taking the architectural aspects and the algorithmic formulation of the router into account. These requirements lead to a set of tradeoffs regarding the number of nets that should be pre-routed for a given design and how these nets should be distributed inside macros to allow for sufficient inter-macro routing bandwidth.

Architectural Considerations for Pre-routing

In order to create a library of pre-routed, intra-macro nets for a pre-placed macro-block, the constraints of the segmented island-style routing architecture must be respected. Consider the FPGA snapshot shown in Figure 5-12. Since routing resources are evenly distributed in segmented channels, abutting pre-routed macros together can cause routing resource contention. Clearly, both Macro 1 and Macro 2 in the figure cannot have pre-routes that use the same contested tracks. As a result, pre-routed nets that collide with those from neighboring macro-blocks must be removed and rerouted. A second constraint arises from wire segments that span more than one logic block. These segments may overlap two or more macros depending on length, as indicated in the Figure 5-12. To avoid overlap, these segments must either remain unused for pre-routed nets or be removed if resource collisions occur. To avoid these types of resource collisions, in subsequent experiments, only single-length segments are used for pre-routing.

An Algorithm for Pre-routing

In selecting macro-block nets for pre-routing, it is desirable to only allow nets to fill each routing channel to a pre-specified channel occupancy[2], W_{pre}, to avoid the creation of routing congestion. Any intra-macro net selected for pre-routing that overflows this FPGA channel occupancy is not pre-routed and must subsequently be routed by the fast router from scratch, along with the inter-macro nets.

[2] Occupancy here indicates the number of tracks in the channel that are used.

W_{pre}: maximum pre-route occupancy of an intra-macro channel.
Loop over all macros.
  Order intra-macro nets by fanout.
  Loop over intra-macro nets.
    Mark net for pre-routing.
    Loop over each segment channel of intra-macro net.
      If channel occupancy + 1 > W_{pre}.
        Unmark net for pre-routing.
        Break loop.
    End
  End
End

Figure 5-13: Algorithm: Pre-route Intra-Macro Nets

An algorithm that selects nets to be pre-routed appears in Figure 5-13. For each macro, the intra-macro nets to be pre-routed are first sorted by fanout so that difficult-to-route high-fanout nets are given the highest likelihood of being successfully marked for pre-routing. Then, the routes of the pre-route candidates are traced from source to destination in step with dynamic evaluation of channel occupancy for each FPGA device channel. If a specific channel occupancy exceeds W_{pre}, the net is removed from pre-route consideration and must be routed from scratch in a subsequent floorplan maze routing phase. Otherwise, the net is marked as fixed (pre-routed) and during subsequent routing, the routing tracks and pins needed to connect source and destinations for the net are drawn from a design library.

5.6.3 Results

Quantifying the effectiveness of the combined floorplanner and router flow is difficult given the complex interaction between three parameters: place time, route time, and the number of routing tracks per routing channel in a target FPGA device. In this section, two seperate evaluations are made of the layout flow to quantify the quality of placements created by the floorplanner and to judge its effectiveness as a fast compile solution. In each of these experiments two of the three parameters mentioned above are varied while the third is kept constant.

In the first set of experiments, the quality of floorplanned placements is directly compared with the quality of fine-grained simulated annealing placements by targetting a number of benchmark designs to a set of fixed array-size devices, each of which contain routing channels with a different channel width. While, clearly, the track count of an in-system device cannot be changed, this analysis helps quantify the placement costs of floorplanning versus annealing for a given logic block array size. It will be seen that while annealing frequently achieves a more routable placement in terms of the number of tracks needed to route the design in a fixed amount of time, the amount of placement time needed to reach

Design	Data width (bits)	Macros	Logic Blocks	Ave. Macro Size (LBs)	% Intra-macro Nets
bubble32	32	63	5153	81	41
bheap5	32	46	8429	183	58
merge32	32	63	8875	140	64
fft16	9	48	11856	247	91
ssp32	16	79	3873	49	65
spm16	16	37	6632	179	90

Table 5.1: Benchmark Example Statistics

this quality point is often an order of magnitude longer, on average, than floorplanning.

In the second set of experiments, the track count of the device is held fixed while place and route times are allowed to vary. In some trials, a floorplanned placement is shown to be unroutable for a given device with a fixed track count given the shape limitations of the design and the internal layout of the macro-blocks. By using the routability evaluator of Chapter 3, this condition can be quickly identified and placement modification, through the use of low-temperature annealing, can be used to make the initial floorplanned placement more routable.

The benchmark circuits used with the floorplanner were taken from the RAW Reconfigurable Computing Benchmark Suite described in Chapter 2. Design information about the specific benchmarks used is shown in Table 5.1.

Experiment 1: Floorplan Quality

In this section, a comparison is made between placement quality that is achieved with floorplanning and placement quality that is achieved with simulated annealing. Specifically, given a feasible floorplan, an evaluation is performed of the amount of time needed by annealing to reach a placement that is as routable as the floorplanned design. While design wire length could be used to evaluate quality in this comparison, a more complete analysis requires a post-placement routing step, so that the combined total of place *and* route time may be considered.

Earlier in the chapter, it was stated that the motivation for exploring macro-based placement approaches is a desire to develop CAD techniques that avoid the long delays associated with simulated annealing. Considering this goal, it makes sense to evaluate the floorplanner in the context of fast placement followed by *fast routing*, defined in Chapter 3 to be route completion achieved in less than 60 seconds. As an initial set of trials to evaluate floorplan quality, six benchmarks were evaluated using the following steps:

1. The six benchmark circuits were floorplanned using weighted clustering and mincut into the smallest FPGA arrays that would hold them of sizes indicated in Table 5.2. For most circuits, logic block utilization of approximately 60-65% was achieved.

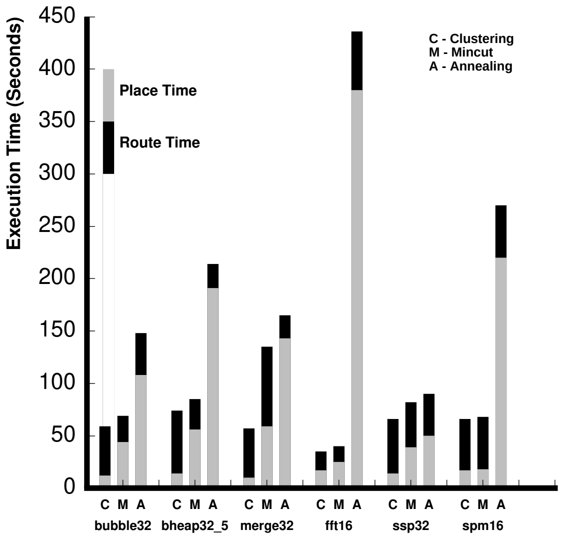

Figure 5-14: Comparison of Floorplanning and Annealing at W_{minf}

Design	Array Size	Fplan		Annealing Effort						
		w/Clustering		Low ($\beta = 0.2$)		Medium ($\beta = 0.5$)		High ($\beta = 1.0$)		
			place		place		place		place	
	(LBs)	W_{minf}	time (s)	W_{minf}	time (s)	W_{minf}	time (s)	W_{minf}	time (s)	W_{min}
bubble32	90x90	17	12	22	88	15	219	16	438	12
bheap5	118x118	25	14	25	180	18	404	18	797	15
merge32	118x118	24	10	24	181	17	444	16	903	12
fft16	140x140	18	17	24	305	19	797	19	1663	13
ssp32	80x80	16	14	13	48	13	178	12	383	11
spm16	102x102	16	8	21	135	18	336	14	675	12
total		116	75	129	937	100	2378	95	4859	75

Table 5.2: Floorplan Array Sizes and Low-stress Minimum Track Counts

2. The floorplans were then routed, using the fast router of Chapter 3, using a number of different channel track widths, until the minimum track width that allowed routing in less than 60 seconds was found. This track count is designated as the *low-stress minimum track count* or W_{minf} for the placement. In this first set of experiments, no pre-routed nets were used during the routing phase (e.g. all nets were routed from scratch).

3. Placement was then restarted from a random logic block placement using simulated annealing. Fine-grained placement was performed on the designs until placements were located that would achieve a successful route time of 60 seconds or less for devices with the same minimum track counts (W_{minf} values) and array sizes found in Step 2. Annealing run time was controlled by varying β, the number of moves per iteration, as outlined in Chapter 4.

Design	W_{winf}	W_{minf}
	clustering	mincut
bubble32	17	16
bheap5	25	25
merge32	24	23
fft16	18	18
ssp32	16	15
spm16	16	16
total	116	113

Table 5.3: Low-stress Minimum Track Counts for Mincut and Weighted Clustering

All run time results were obtained using a 140 MHz UltraSparc 1 with 288Mb of memory. In Figure 5-14, it can be seen that for the six large benchmark circuits considered, low-stress minimum track count values (W_{minf}) determined by floorplanning are achieved, on average, an order of magnitude faster for weighted clustering than annealing, and a factor of five faster for mincut-based floorplanning than annealing.

A second set of annealing and floorplan trials were performed to judge the quality of floorplan placement relative to the *best* quality placement that could be achieved by annealing runs of extended periods of time. Annealed placement run time was once again controlled by varying β, the number of moves per annealing iteration. For each placement (both annealed and floorplanned), the minimum track count that could be successfully routed in 60 seconds was determined. In Table 5.2, results summarizing the length of time for placement trials and the resulting low-stress minimum track counts are shown. These results indicate that the floorplan quality created by weighted clustering falls within the low to moderate range of annealing quality.

In general, as designs increased in size, the benefits of floorplanning were more apparent. This is due to an exponential increase in number of moves needed to achieve the same quality level for annealing. Note that W_{min}, the minimum routable track count that can be achieved with essentially unbounded place and route time, is shown for quantitative comparison with fast compile results.

As seen in Table 5.3, the use of mincut instead of weighted clustering for floorplan subdivision resulted in slightly lower W_{minf} values. This would indicate that for devices with small numbers of tracks per channel, mincut would have a higher chance of successfully creating a routable floorplan.

Consideration of Pre-routed Nets

The numbers presented for the annealing and floorplan results shown in Table 5.2 were generated by routing FPGA placements from scratch. In this section, the effect of pre-routing critical nets inside the macro-blocks is considered to determine if a device with a *reduced* track count, in comparison with the previous results, can be successfully routed in 60 seconds of route time. To evaluate the effect of

Design	W_{minf}	W_{minf}	W_{pre}
	w/o pre-routes	pre-routes	(in tracks)
bubble32	17	16	1
bheap5	25	23	1
merge32	24	23	1
fft16	18	16	1
ssp32	16	16	1
spm16	16	14	2
total	116	105	

Table 5.4: Reduction of W_{minf} Through Pre-routing

pre-routing, the low-stress minimum track counts shown in column 3 of Table 5.2 were recalculated by performing floorplan routing with some nets pre-routed. Prior to each floorplan route, the algorithm shown in Figure 5-13 was applied to the macros for various levels of pre-route density. As mentioned in Section 5.6.2, pre-route density for a macro is defined by the maximum occupancy of pre-routed nets in a routing channel, W_{pre}. The lower the value of W_{pre}, the smaller the number of internal macro nets that are pre-routed.

The floorplans evaluated earlier without pre-routing were rerouted using W_{pre} levels ranging from 1 to 6. As summarized in Table 5.4, the use of pre-routing further reduced the track count that could be achieved with routing time of 60 seconds, W_{minf}, for all floorplans except one. Suprisingly, the best routability results were achieved with only minimal pre-routing for most designs. This indicates that for floorplanned macro-block designs, macro-block pre-routing at a pre-route density of 1 prior to routing will typically enhance the likelihood of rapid route convergence. Consider the graph shown in Figure 5-15 of route time at various pre-route densities for a fixed track count of 16. As W_{pre} is increased from 1 to 6, routing time increases from a minimum at 1 due to intra-macro route congestion. Note that W_{pre} of 0 indicates no pre-routing is performed.

Experiment Summary

Most current FPGAs are currently designed with track counts that just barely support routing even for multiple, long router iterations. Therefore, for the designs used in this experiment, it is appropriate to contrast the number of tracks per channel needed to achieve fast place and route in about 60 seconds, W_{minf}, with the minimum track counts that could be achieved with unbounded place and route time, W_{min}, to determine the resource cost of fast compilation. Column 3 in Table 5.4 and column 11 in Table 5.2 indicate that the resulting totals across all designs for the trials explained above are 105 tracks for W_{minf} and 75 tracks for W_{min}. Just as the results in Chapter 3 indicated that fast routing could be achieved with about 30-40% additional tracks per FPGA channel, this new result indicates that fast floorplanning *and* fast routing could be achieved with between 40-50% additional routing tracks per

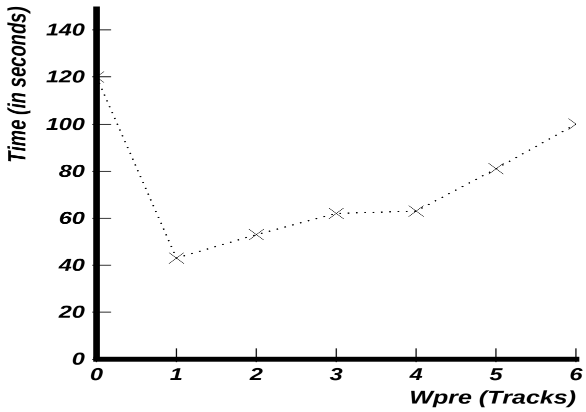

Figure 5-15: Routing Time versus W_{pre}, FFT16, $W_{FPGA} = 16$

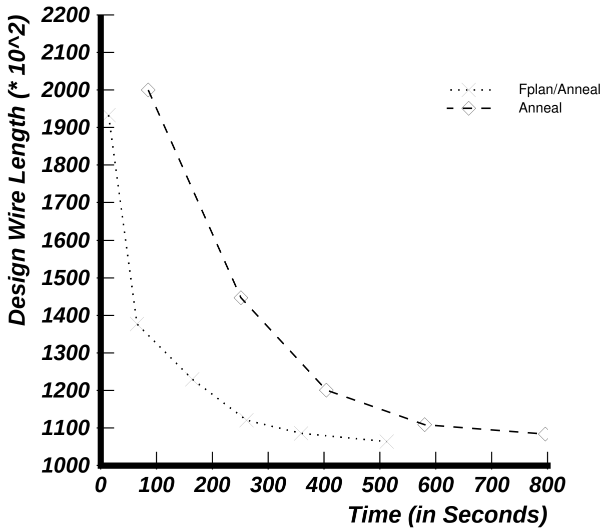

Figure 5-16: Low-temperature Anneal Tradeoffs - Example Bheap32

FPGA channel.

Experiment 2: Combined Floorplanning and Annealing

In an FPGA computing system, the track count of a target FPGA is fixed, not flexible. Here, it is shown that if a floorplan is determined to be unroutable, low-temperature annealing can be applied to the placement to reduce wire length cost to more routable levels.

In Figure 5-16, the dashed curve displays anneal time/placement cost tradeoffs created by varying β, the number of moves per annealing iteration, between 0.1 and 10 for different annealing placement runs. For these trials, random logic block placements were used as an initial placement for subsequent iteration. The dotted curve shows results achieved when annealing is applied to an initial placement created by the floorplanner. For these trials, a constant start temperature **T** of 0.3 is specified. Here, a low **T** value is used to limit the number of swaps that increase cost since the initial floorplan already has a great deal of locality and only minor placement perturbations are needed. It can be seen in the figure that floorplanning can be an effective way to create an initial placement for simulated annealing for low track count devices, as well as an approach to create a placement quickly for devices with abundant routing resources.

Chapter 6

Isolation of Resources

Much of the compilation speed for microprocessor-based systems is due to the modularity of the compilation process. If a specific source file is changed, incremental compilation can be limited to the affected module assuming that the interface to other program modules remains the same. In this chapter, modular layout approaches for island-style FPGAs are considered. In addition to macro-block floorplanning, which isolates logic in rectangular-shaped regions of FPGA devices, internal macro-block nets are routed on a specific set of routing tracks that are free of inter-macro nets. This new limitation forms a physical routing boundary and facilites internal change to macro-block layouts. In this chapter, it will be shown that by following a structured *routing* methodolgy, completed layouts can be constructed quickly (in a few seconds for small designs) for macro-based designs and incremental design modification can easily be performed by using both logic and routing resources devoted only to individual macro-blocks.

In the course of describing the two layout flows for isolation developed in this chapter, it will be shown that island-style architectures are generally not well architected to support modular layout. The first isolation approach that is presented leads to significantly reduced utilization of device logic and routing resources (only 10-20% of total device resources used) compared to layout styles discussed in previous chapters. In design situations where device utilization is not a significant issue, this approach can be applied directly to existing commercial devices to achieve fast layout that can be easily modified. In contrast, the second layout approach allows for high device logic utilization at the cost of additional required tracks per FPGA routing channel relative to the number commonly found in contemporary commercial devices.

While the developed approaches have limitations for existing island-style devices, they offer insights into how device architectures should be changed to better support modular compilation. A number of these suggested architectural changes are noted in the next chapter and offer a motivation for future work.

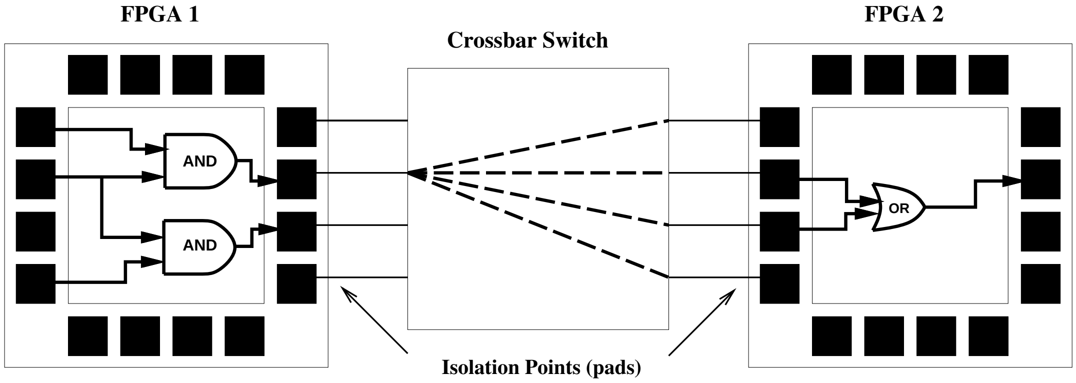

Figure 6-1: Isolation of Routing Resources

6.1 An Example of Routing Resource Isolation

Before reviewing the two types of modular layout presented in this chapter, the term *resource isolation* is better defined through the use of an example. Generally, it can be said that a set of design logic and routing resources are *isolated* from the rest of the physical design if the layout has been created in such a way that intra-macro resources can be changed internally, without the need to change any of the layout for other parts of the design. As an example of isolation at the multi-FPGA system level, consider the two FPGA devices connected with a programmable crossbar switch shown in Figure 6-1. Each wire that emanates from a pad on FPGA 1 carries a single inter-FPGA design net that can be routed through a programmable crossbar from FPGA 1 to FPGA 2. The pad in this case serves as an *isolation point* between the logic and routing inside FPGA 1 and net routing in other parts of the system. Clearly, this system configuration provides flexible *inter-device connectivity*, since any wire from one FPGA can connect to any wire in the neighboring device. As long as the assignment of nets to isolation points (pads) remains the same, logic and routing changes inside the FPGAs can be made without requiring any other system change, effectively isolating the FPGA to a set of logic and routing resources and a list of isolation points. In this example, isolation has been facilitated by the architecture of the system and the fact that inter and intra-FPGA resources have well-defined boundaries.

In contrast, island-style FPGAs do not have explicit routing boundaries in hardware that can be used to isolate routing resources. In this chapter, two approaches to isolated layout for island-style FPGAs are presented that attempt to build a hierarchy of routing isolation on top of the flat routing architecture of the device. The first approach follows a design style that is similar to the one used in full-custom, macro-block layout for ASICs. As seen in Figure 6-2, for this macro-block design style, macro logic and routing resources are isolated in specific *planar* regions of the device. Inter-macro routing takes place in channels around the border of the macro-blocks forming distinct functional boundaries between macro-blocks and the remainder of the circuit. For full-custom technology, this discretization is necessitated by a limited number of metal routing layers in process technology which causes the physical

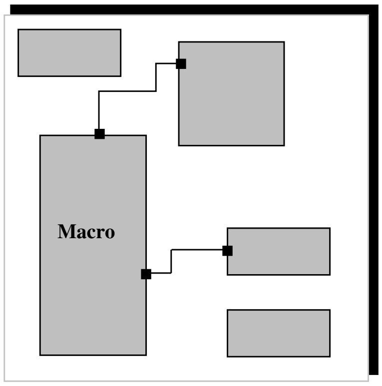

Figure 6-2: Full-custom Macro Design Style

implementation of the macro-block to be a hard boundary to external macro-block routing. In the next section, a similar approach for modular FPGA layout is developed in which all intra-macro nets are pre-routed on routing resources inside the planar scope of the FPGA macro and additional space is left between macros in the form of free routing resources to allow for inter-macro communication. While this *planar* approach to modular layout is shown to converge very quickly for small island-style designs and is directly applicable to existing commercial FPGA devices, it wastes increasingly large amounts of logic and routing resources as design and device sizes scale. This inefficiency is demonstrated to be the result not only of device architecture but also of the basic wirability properties of most macro-based circuits. In upcoming analysis, it will be shown that planar isolation approaches for island-style FPGAs are provably not scalable with design size for circuits with Rent exponents of greater than 0.5.

The lack of scalability for planar approaches motivates evaluation of an additional hierarchical approach that is scalable and takes advantage of the segmented nature of island-style routing. This second approach uses placements created by the Frontier floorplanner to isolate inter and intra-macro routing segments not by planar region but rather by *routing domain*. For this approach, routing tracks in each FPGA channel are assigned to either inter or intra-macro routing. This *domain-based* isolation approach has the advantage of not only allowing for routing isolation, but also, with sufficient routing resources, of allowing for increased device logic utilization as designs scale.

6.2 Planar Isolation of Resources

In this section, an integrated placement and routing technique is developed that allows for the isolation of intra and inter-macro routing resources based on hierarchy defined by a floorplan slicing tree. This layout design style closely follows the general flow of full-custom, macro-based layout. First, macro-blocks are assigned to disjoint planar regions of the FPGA device. Then, routing resources surrounding the macro-blocks are used to interconnect inter-macro nets. As seen in Figure 6-3, to create an inter-macro routing area, some logic blocks in the FPGA device must be left unutilized. This necessary loss of logic

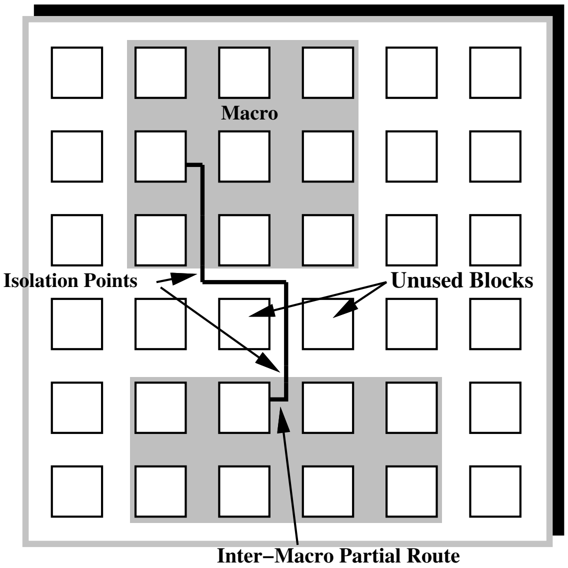

Figure 6-3: Planar Isolation Design Style

utilization is a key factor in evaluating the scalability of planar isolation approaches. Subsequently, it will be seen that while isolated layouts can be completed quickly for circuits, the mandatory loss of logic utilization due to characteristics of the routing architecture makes this type of approach prohibitive as designs scale. An analysis using parameters derived from Rent's Rule also confirms this lack of scalability.

The layout approach presented in this section builds upon the hierarchical floorplanning algorithm presented in the Chapter 5 through the integration of *hierarchical routing*. During floorplanning, the user netlist is first recursively subdivided into a slicing tree structure and then nodes of the partial floorplans are combined together to form a final, feasible floorplan. For the new, integrated floorplanner and router, an additional step of *routing* is added to floorplan shaping for each partial floorplan to quickly create a new partial floorplan macro-block that is both placed and routed. By recursively routing around each of the newly formed macro-blocks and then treating the result as a single unit, a hierarchical layout can quickly be formed.

6.2.1 Limitations of Approach

The flat organization of routing resources in island-style devices presents several limitations to the development of a planar isolation approach. In addition to routing resource constraints regarding the use of wire segments that span multiple logic blocks mentioned in Section 5.6.2, additional constraints related to device architecture are noted here. Following discussion of practical implementation issues, an analytical analysis of planar isolation shows that planar-based routing approaches are provably not scalable for island-style FPGAs. This finding motivates a different isolation approach based on routing domains discussed later in the chapter.

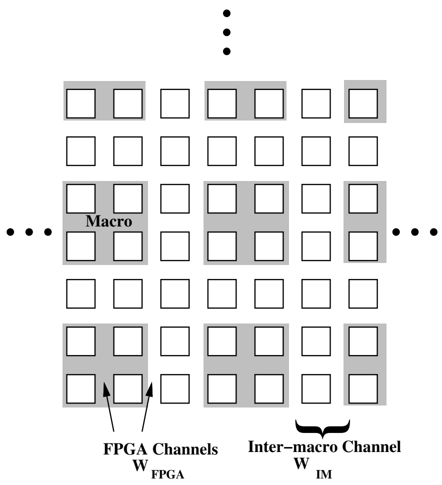

Figure 6-4: Analysis of Planar Isolation

Domain Limitations

As shown in Figure 6-3, to create a macro that supports planar isolation it is necessary to route an inter-macro net partial route from a logic block in the interior of a macro-block to an isolation point at the macro-block perimeter so that it may be subsequently connected to inter-macro routing. Ideally, this isolation point would be provided by logic block input and output pins to limit the length of partial routes. In most cases, however, the number of inter-macro I/O signals emanating from the macro is greater than the number of perimeter blocks, necessitating isolation points at *wire segments* along the border of the macro. As reviewed in Section 2.2, the switchbox topology of island-style FPGAs divides routing tracks into routing *domains*. Thus, the partial net leading up to the isolation point must be constrained to a specific track domain. It has been mentioned previously in Section 2.2 that the only location on the device where routing domains for a net route may be changed is at the source logic block for the net, which in this case is embedded inside the source macro for the net. As a result, to form a feasible connection between the partial net of the macro at the source of the net and the partial nets at destination macros, *wire segments in channels between macros and wire segments internal to macros must use segments in the same routing domain as the partial route in the source macro* or no feasible connection can be made at isolation points. In comparison to the example of Figure 6-1, inter-macro connectivity is quite limited. Instead of inter-macro connectivity provided by a full crossbar, connectivity has been reduced to wire segments in one specific domain, a small fraction of routing resources available on the device.

Rent's Rule Limitations

The limitation mentioned above is specific to the XC4000-style routing architecture and could be eliminated by increasing switchbox flexibility to ease domain limitations at the cost of additional routing switches. Interestingly, a more fundamental limitation to the scalability of planar isolation for island-style FPGAs exists and is outlined here via analytical analysis. In this section, it is shown that a design methodology that isolates macro-block logic and routing resources in planar regions and then routes inter-macro nets in channels around the regions is non-scalable for circuits with Rent exponents greater than 0.5. This technique is also shown to cause an increasingly large amount of resource wastage as designs and devices scale **regardless of segmentation or switchbox flexibility** for a uniform grid of logic blocks surrounded by fixed sized channels of routing resources.

Consider the FPGA array shown in Figure 6-4. In this figure, FPGA routing channels, each with channel width W_{FPGA}, are split into two groups, those located within macros to allow for intra-macro nets and those external to the macros that have been grouped into inter-macro channels for use in inter-macro routing. From the figure it is apparent that there are two situations that might arise that would lead to logic and routing resource wastage in the FPGA device. First, if W_{FPGA} is excessively large compared to the channel width needed to route intra-macro nets, large amounts of potential device *routing* bandwidth will go unused inside the planar region of the macros since it cannot be used by inter-macro routing. Second, a possible point of *logic* loss is located in the inter-macro channels surrounding the macros. It can be said that the width in tracks of the inter-macro routing channel, W_{im}, is

$$W_{im} = \gamma \times W_{FPGA} \tag{6.1}$$

where m is the number of FPGA routing channels per inter-macro routing channel ($m = 2$ in the figure) and W_{FPGA} is the track width of each FPGA channel. If W_{im} is large, the number of unused logic blocks in the channel must be large, leading to reduced *logic* utilization for the device.

Intuitively, it would appear that as design sizes, in terms of numbers of macros, and device sizes scale, the amount of resources lost due to these inefficiencies could increase due to increased required bandwidth in channels between macros to support larger circuits. Next, it is shown that, in fact, the rate at which resource loss occurs can be determined analytically to be exponential by calculating the expected size of W_{im} as designs scale. To simplify the analysis of this growth rate, a few simplifications are made. First, it is assumed that all macros have the same square shape, have been placed in a square array, much like the organization of logic blocks in the FPGA array, and have an equal number of I/O pins, λ. As the design scales, additional macros are added such that the square shape of the array is preserved.

W_{im} may be thought of as a *minimum required* inter-macro channel width in the *device* needed to route inter-macro connections. To achieve route completion, this value should be same as the *maximum* inter-macro channel width of the routed *circuit*. The issue of inter-block channel width for square block-based circuits has been addressed previously by El Gamal. In [27], it was determined that inter-block channel widths for block-based circuits in an array exhibit a post-route width distribution, W, that is

Poisson. Additionally, it was shown that the *average* channel width, $\overline{W}$, of the distribution is:

$$\overline{W} = \frac{\lambda \overline{R}}{2} \tag{6.2}$$

where λ is the number of I/O pins per macro-block and $\overline{R}$ is the average wire length of the design in terms of macro-blocks. Given this information, the *maximum* inter-macro channel width, of the macro-based circuit (and the minimum required channel width of the device, W_{im}) is related to the average channel width, $\overline{W}$, and width standard deviation, $\sqrt{W}$, [11] as:

$$W_{im} \approx \overline{W} + \sqrt{\overline{W}} \tag{6.3}$$

In Section 3.4.1, it was established that for a typical design with Rent exponent p greater than 0.5, $\overline{R}$ grows at a rate proportional to $N_{macros}^{p-0.5}$ by Equation 3.6. As a result, it can be said that due to Equation 6.2:

$$\overline{W} \propto N_{macros}^{p-0.5} \tag{6.4}$$

and by Equations 6.3 and 6.4:

$$W_{im} \propto N_{macros}^{p-0.5} \tag{6.5}$$

Hence, there exists an *exponential* relationship between the number of macros in the design, N_{macros}, and the width of the inter-macro channels, W_{im}, needed to route the design if the design Rent exponent is greater than 0.5, the common case for most circuits.

In light of this relationship, let us reconsider the two cases mentioned previously regarding the two possible areas of resource wastage, unused routing tracks inside macros and unused logic blocks in inter-macro channels, to evaluate their growth rates as design sizes increase. Consider two square macro-based array layouts (Design 1 and Design 2), similar to the one shown in Figure 6-4, of different sizes ($N_{macros2} > N_{macros1}$) that contain same-sized macro-blocks. By Equation 6.1, the inter-macro channel width, W_{im1}, of Design 1, targetted to an FPGA with channel width W_{FPGA1} is:

$$W_{im1} = \gamma \times W_{FPGA1} \tag{6.6}$$

Since Design 2 has more macro-blocks than Design 1, by Equation 6.5, the required inter-macro channel width for the second design must proportionally increase ($W_{im2} > W_{im1}$). Considering this increase, two implementation choices exist for Design 2. The design could be targetted to an FPGA with an exponentially larger FPGA channel ($W_{FPGA2} > W_{FPGA1}$) so that the increase in W_{im} could be absorbed without increasing γ, the number of FPGA channels per inter-macro channel. However, since the size of each macro, in terms of logic blocks and the FPGA channel width needed to route intra-macro nets, has not changed as the design has scaled, any additional *routing* resources added within the macro ($W_{FPGA2} - W_{FPGA1}$ per FPGA channel) must be wasted.

An alternate approach to avoid routing resource wastage inside macros would be to target the larger design to a device with the same FPGA channel width, W_{FPGA1}, and to increase the size of inter-macro channels to include additional FPGA channels or:

$$W_{im2} = \gamma_2 \times W_{FPGA1} \tag{6.7}$$

where $\gamma_2 > \gamma$. This approach results in an increasingly large loss of *logic* resources following the exponential increase of W_{im} in Equation 6.5.

6.2.2 Implementation Steps

The layout approach described in this section not only generates layouts with isolated routing resources, but also effectively trades off device logic utilization for reduced place and route time. By limiting inter-macro routing to regions between macros, the routing search space of the design can be reduced and faster route times can be achieved at the cost of lower device logic utilization.

In this section, a set of constructive layout steps are presented that quickly create a hierarchical layout based on planar isolation. These steps necessarily have low complexity to allow for rapid layout convergence, although several potential optimizations are noted during the description of the layout algorithms in subsequent sections. Layout for planar isolation is performed through the following steps:

- Initially, the clustering-based floorplanner from the previous chapter is applied to a macro-block design to create an approximately square floorplan with minimized design wire length and to determine relative macro-block positioning and shape.

- A bottom-up traversal of the slicing tree formed by floorplanning (shown in Figure 5-5) is performed to complete design routing. For each two macro-block partial floorplan, inter-macro wiring is completed by routing around the perimeter of the macros. Isolation points to other parts of the floorplan design external to the macro pair are located along the rectangular perimeter of the new partial floorplan and routing is extended to these points. In effect, this creates a new placed and routed partial floorplan macro.

- After each two-macro partial floorplan in the slicing tree has been routed, as a final step, net

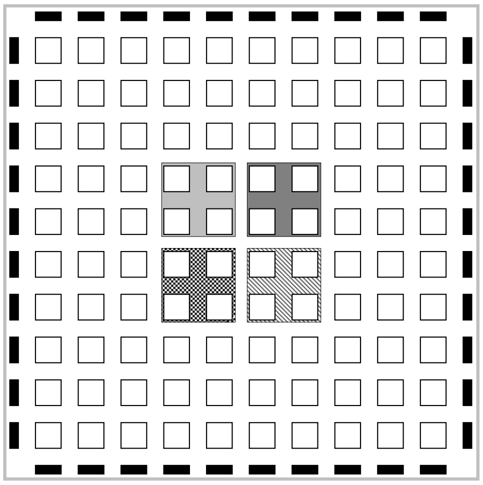

Figure 6-5: Step 1: Floorplan Based on Weighted Clustering

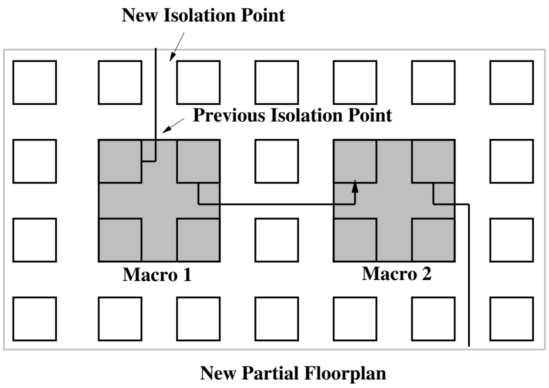

Figure 6-6: Step 2: Perimeter Routing

interconnection to device pads is completed by routing in channels along the perimeter of the device.

The result of these steps is a placed and routed circuit that can be easily modified at the macro level. Each of the above three steps are detailed in following sections.

Initial Floorplan

Before hierarchical routing takes place, an initial floorplan is created to isolate macros that are tightly connected and to identify macro shapes and rotations that minimize wire length. As seen in Figure 6-5, the floorplanner based on weighted clustering from the previous chapter is used to create a densely-packed initial floorplan of approximately equal height and width. This condensed floorplan provides an initial point for inter-macro routing. As described in the next section, routing space is inserted between blocks to allow for needed bandwidth as part of the routing process.

Inter-Macro Routing

A distinctive feature of this planar isolation approach is the division of circuit routing into a series of smaller routing subproblems. As seen in Figure 6-6, in this stage, two macro-blocks with isolation points located around their perimeter are combined together to form a new macro-block with net connections to other circuitry in the floorplan located around its perimeter. As mentioned in Section 6.2.1, the architecture of the FPGA device requires that each net in the partial floorplan be routed in only one routing domain. This is a significant limitation that restricts the routing resources that can be used to route each net inside macros and affects how net isolation points along the perimeter of the new macro-block are allocated. The discretization of routing in partial floorplans allows for local net routes to be completed quickly and for routing congestion in inter-macro channels to be easily diagnosed and alleviated. The *global* extent of nets that extend beyond the dual-macro pair can be taken into account by treating the isolation point selection problem as a pin assignment problem.

Inter-macro routing for each partial floorplan takes place in three steps:

1. **Isolation Point Selection** - In this step, isolation points for each of the nets that extend beyond the two children of the new partial floorplan are assigned to specific single-length wire segments along the partial floorplan perimeter.

2. **Partial Floorplan Routing** - In this step, the fast router detailed in Chapter 3 is used to quickly route nets between the two child macros and to the isolation points along the perimeter of the partial floorplan.

3. **Floorplan Relaxation** - In this step, routing congestion in the partial floorplan is evaluated. If the partial floorplan route completes with overused routing resources (those having occupancy $>$ 1) in inter-macro channels, additional routing channels must be added to the partial floorplan. This is accomplished by stretching the rectangular box of the partial floorplan to include more FPGA channels and unused logic blocks. If there are no overused resources, the partial floorplan route has completed successfully.

The first step in forming a routed partial floorplan from two child macro-blocks is to assign isolation points to specific wire segments along the partial floorplan perimeter for each inter-macro net that connects to other parts of the slicing tree. Physically, this isolation point serves as a bridge between routing inside the partial floorplan and routing for the remainder of the circuit. The assignment of nets to isolation point wire segments along the perimeter of the partial floorplan is a constrained placement problem similar to the pin assignment problem [52] for flexibly-shaped macros in ASIC floorplanning. In the following formulation, nets are first assigned to isolation points in a random fashion, and then greedily swapped, based on overall net wire length, to quickly optimize placement. In selecting isolation points for specific nets, domain limitations must be respected. As an example, consider the logic block

Loop over all inter-macro nets.
Identify net domain, d of selected net.
While unoccupied isolation point not found.
Select isolation point wire segment with domain d.
If isolation point associated with domain d unoccupied.
Assign net to isolation point.
End
End
Determine total wire length cost, C, associated with isolation points.
Swap isolation points until C is a minimum.

Figure 6-7: Algorithm: Determination of Inter-Macro Isolation Points

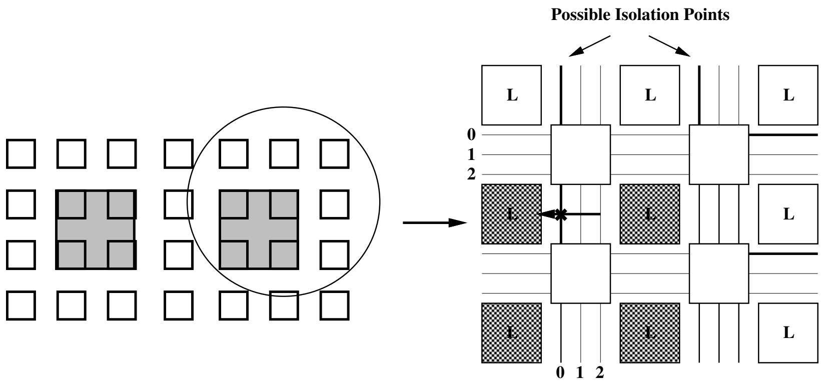

Figure 6-8: Inter-macro Isolation Point

input shown in Figure 6-8 that connects to floorplan points outside the partial floorplan. Here, it can be seen that the net connects to the input using a wire segment in routing domain 0. As a result, only the boldface wire segments along the perimeter of the partial floorplan that are also in domain 0 can be considered as isolation points. Following initial selection of isolation points, greedy swapping is used in an effort to minimize wire length between macros and isolation points. Details of the isolation point selection algorithm can be seen in Figure 6-7.

Following assignment of inter-macro isolation points, routing of the partial floorplan takes place using the fast router of Chapter 3. If the router completes with overused track segments in inter-macro channels, additional bandwidth must be allocated to the channels to allow for a successful route. Following routing, an evaluation of congestion is made in each of the five border regions surrounding the two child macro-blocks (top, bottom, left, right, middle), based on the ratio of overused wire segments, S_{ou}, to available wire segments, S_{av}. As a result of this evaluation, one additional row or column of FPGA routing resources (and unused logic blocks) is added to the affected region by extending the horizontal or vertical extent of the partial floorplan. This loop of evaluation followed by relaxation continues until a successful route is completed with no overused tracks. Considering the small size of

Design	Data width (bits)	Macros	LBs	Ave. Macro Size (LBs)	# Inter-macro Nets
ssp4	16	11	573	52	206
ssp8	16	18	1304	72	326
ssp16	16	46	2450	53	790
bsort16	16	23	973	42	533
spm4	16	11	2204	200	206
spm16	16	37	6632	179	646

Table 6.1: Benchmark Example Statistics

Design	Array Size (blocks)	Logic Utilization	Floorplan (s)	Routing (s)	Total (s)
ssp4	66x66	13%	4	6	10
ssp8	99x99	13%	6	10	16
ssp16	130x130	14%	9	18	27
bsort16	96x96	11%	6	16	22
spm4	100x100	22%	4	10	14
spm16	166x166	24%	8	20	28

Table 6.2: Planar Isolation Results

most partial floorplans, this evaluation can be usually be performed quickly.

Following application of partial floorplan routing for each two-macro pair, connections between the highest-level partial floorplan, which includes all macros amd associated routing, and the I/O pads of the device are made. If the resulting floorplan and routing will not fit within the target device, routing without isolation, as detailed in Chapter 5, can be applied.

6.2.3 Planar Isolation Results

The building block style of floorplanning and routing outlined in preceding sections was applied to six benchmarks from the RAW benchmark suite detailed in Table 6.1. The target device for these designs contained 8 single-length wire segments per routing channel, the same number found in devices from the Xilinx XC4000 family. All routing both inside and between macros was performed using the single-length lines of the device.

All run time results were obtained using a 140 MHz UltraSparc 1 with 288Mb of memory. Results were generated by targetting the smallest FPGA device that would hold the placed and routed design. From Table 6.2, it can be seen that the price of isolation in terms of logic utilization is high due to the number of blocks wasted in inter-macro channels. As expected, the two designs with the largest average macro size (spm4, spm16) had the best logic utilization. In addition to providing isolation, the hierarchical routing approach was very fast, leading to place and route times of less than 30 seconds for all designs.

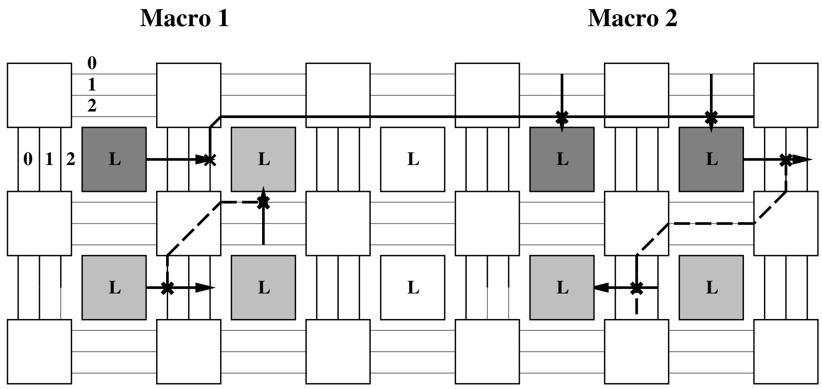

Figure 6-9: Domain-based Isolation

6.3 Domain-based Isolation

While an approach that achieves 40-50% device utilization is acceptable for many end users of FPGAs, an approach the achieves only 10-20% is prohibitively expensive for many. In Chapter 5, increased routability was achieved by adding routing tracks to each routing channel. In this section, a similar technique is explored to promote isolation of routing resources while maximizing the logic utilization of the island-style device.

A second means of creating routing hierarchy in an island-style architecture is to isolate routing resources not by planar region, but rather, by track domain. In Chapter 2, it was noted that wire segments in each routing channel are assigned to a specific domain by the switch organization of routing switchboxes. By designating each track in a routing channel as capable of routing *either* inter-macro or intra-macro nets, a partitioning of resources can be achieved.

An example of domain-based isolation can be seen in Figure 6-9. In the figure, an inter-macro net, represented as a solid line, is routed on tracks in domain 2 from Macro 1 to Macro 2, while intra-macro nets, represented as dashed lines, are routed on tracks in domains 0 and 1. Note that while the planar isolation approach offered a *single point* of isolation for inter-macro routes, in Figure 6-9, the inter-macro net connects to Macro 2 at two points, both input pins to logic blocks. As a result, changes to the internal logic and routing structure of a macro can be made independently of the rest of the circuit as long as both darkly shaded blocks, which serve as an interface to the remainder of the circuit, remain in place.

In Chapter 5, an evaluation was performed of the amount of routing resources needed to successfully route a floorplanned circuit. In this section, a similar evaluation is performed for designs routed with domain-based isolation.

For each of the trials performed using domain-based isolation, the first six tracks in each routing channel were reserved for intra-macro nets. Prior to routing, intra-macro nets were pre-routed using these resources. During routing, inter-macro nets were excluded from the first six tracks in each routing channel to determine the absolute minimum track count needed to achieve design route completion,

Design	Array Size (blocks)	W_{min} non-isolated	W_{min} isolated
bubble32	90x90	13	21
bheap5	118x118	18	26
merge32	118x118	15	23
fft16	140x140	12	17
ssp32	80x80	12	17
spm16	102x102	11	17
total		81	121

Table 6.3: Comparison of Isolated and Non-Isolated Minimum Track Count

W_{min}, with isolation. The same floorplans based on weighted clustering that were used for Experiment 1 in Chapter 5 were used for these trials. It can be seen from Table 6.3 that about 50% additional tracks were needed to complete routing for the isolated case versus the non-isolated case, in which all nets could be assigned to any track.

Chapter 7

Summary and Future Work

In this dissertation, a collection of placement and routing tools for island-style FPGAs have been described and evaluated. Contributions of this tool set include:

1. A multi-iteration, fast router for island-style FPGAs that achieves routes of the same quality as existing routing algorithms, but in much less time. This router has been specially tuned to the routing structure of island-style FPGAs through a technique called *domain negotiation*. This technique aids in directing the route of each design net onto a set of routing resources that are most likely to achieve route completion.

2. It is shown analytically using Rent's Rule and experimentally using the fast router, that the routing time of FPGA circuits with Rent exponents of greater than 0.5 grows at a faster rate than circuit size.

3. A macro-based floorplanner based on slicing approaches has been developed for island-style FPGAs and is shown to achieve feasible placements for macro-based designs in seconds that are of the same quality as those achieved in minutes with simulated annealing. On average, about 50% additional tracks per channel above the minimum track count needed to route the circuit are required to achieve 60 second place and route time for designs containing thousands of logic blocks. This routing resource penalty is considerably less for designs with large macro-blocks exhibiting limited inter-macro connectivity.

4. Two routing methodologies that isolate routing resources inside FPGA macros and allow for incremental design modification have been implemented and evaluated. It is found that, while current FPGA devices are not well-suited to resource isolation, successful design placement and routing can be achieved in less than 30 seconds if low device utilization is tolerable. In this chapter, changes to FPGA architecture are suggested to make routing resource isolation less costly for future generations of FPGAs.

7.1 Future Directions

Fundamentally, the use of FPGAs requires tradeoffs. In this thesis, through the use of a tool set that takes advantage of design regularity, it has been determined that fast placement and routing can be achieved in many cases with modest increases in device resources. In coming sections, the lessons that have been learned are integrated with suggestions for future research in the area of fast compilation for FPGAs. First, several additional software changes to island-style FPGA CAD flow are suggested to aid the place and route process and to allow for enhanced device routability. Finally, more substantial deviations from island-style architecture and CAD flow philosophy are discussed as viable implementation alternatives for research and implementation over the next decade.

7.1.1 Additional Directions for Island-Style FPGAs

It was shown in Chapter 3 that for island-style FPGAs not only does the *number* of wires emanating from a piece of logic increase at a faster rate than a corresponding increase in logic, but also that *wire length* and *routing time* increase at a faster rate than logic growth for designs with Rent exponents of greater than 0.5. To address this issue in the future, FPGA device and CAD tool designers will likely be faced with two choices as devices scale:

1. Device designers could continue to add routing tracks to FPGA devices at a rate bounded by Rent parameters to accomodate additional required device bandwidth.

2. CAD tool designers could reduce design bandwidth inside FPGA devices by assigning multiple design nets to the same physical routing resources inside the device and *scheduling* communication on the resources at statically-determined times.

Dehon [21] has indicated that currently about 90% of silicon area in FPGAs is devoted to routing resources. Given this observation, it would appear that the first of the two choices above may not be the best one to follow. Thus, in addition to the floorplanning and routing approaches discussed in this dissertation, software and hardware techniques for *scheduling* inter-macro communication between macro-blocks deserves consideration in an effort to reduce overall device bandwidth.

In Chapter 5, a design path similar to the one outlined above in Figure 7-1 was noted in which low-temperature annealing was used as a floorplan modification step to reduce wire length and to allow for the creation of a more routable design. In the flow shown in the figure, a potential design flow enhancement is the replacement of a set of macro-blocks in an unroutable floorplanned design with new macro-blocks exhibiting reduced external I/O due to inter-macro datapath bit-slicing.

One possible implementation for this floorplan modification step is outlined here. For the proposed flow, placed and routed macros not only have fixed size and shape, but also have a *fixed communication schedule* with respect to other macros with which they communicate. A simple example of a macro

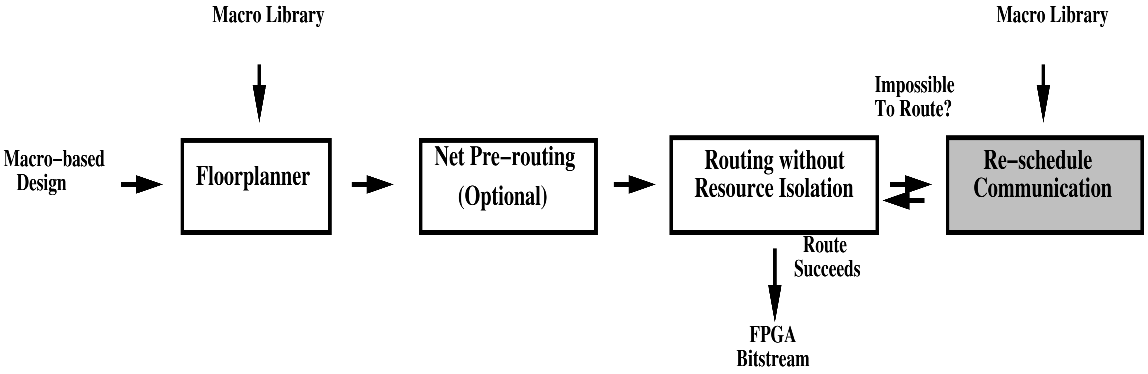

Figure 7-1: Design Flow with Communication Re-scheduling

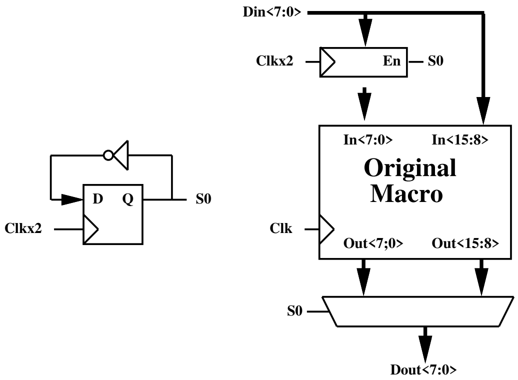

Figure 7-2: Macro-block with Bit-sliced Communication

whose I/O communication has been bit-sliced by a factor of 2 appears in Figure 7-2. In this case, data is communicated at twice the rate of computation inside the macro since the clock **Clkx2** runs at twice the frequency of **Clk**. The specific data value that is received or transmitted by the macro in a communication cycle is determined by state bit S0. If all macros across the circuit are bit-sliced in the same fashion, the function of the circuit is maintained at the cost of a reduction in computation clock rate by a factor of two compared to the maximum supported by the FPGA technology (**Clkx2**). Multiplexers and enabled registers to perform scheduled communication can be synthesized out of FPGA look-up tables and flip-flops and placed and routed as part of macro library layout.

A similar approach of performing scheduled communication between blocks of FPGA resources using logic synthesized from the basic technology of the FPGA was explored in the Virtual Wires project [9]. The motivation for that project, however, was not compile time speed for a single FPGA device, but rather, overcoming pin limitations of individual FPGA devices in a multi-FPGA system.

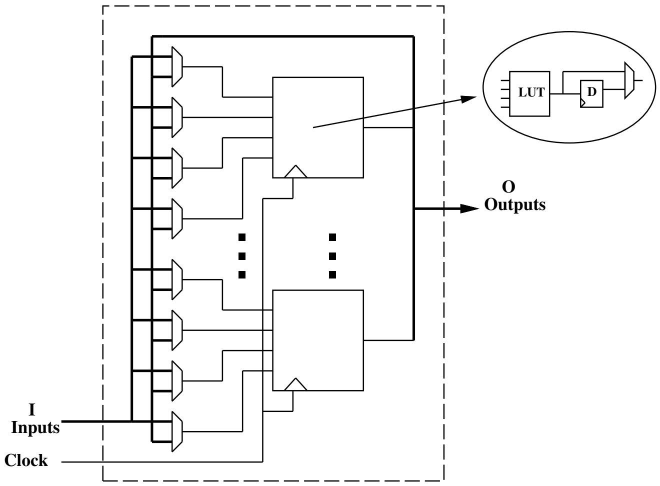

Figure 7-3: Coarse-grained Logic Block

7.1.2 FPGA Architectural Modifications

This thesis has focussed on the development of a fast place and route system for existing island-style FPGAs. As mentioned in Chapter 2, the fundamental characteristic of this architecture is the logic and routing *cell*, a basic block which is highly optimized at both the VLSI and architectural level. This cell is uniformly replicated in a two-dimensional array to form a regular logic and routing grid. While this architectural regularity simplifies fine-grained placement (simulated annealing) and subsequent interconnection (maze routing), in many situations that have been identified in this thesis, the inherent lack of architectural hierarchy inside the FPGA complicates macro-based FPGA layout.

Partitioned Architectures

The lack of architectural hierarchy was particularly noticable as techniques for routing resource isolation were developed in Chapter 6. There, the lack of well-defined hardware boundaries between macros and the flat routing structure of the device made isolation approaches non-scalable and difficult to implement. Recently, as shown in Figure 7-3, commercial FPGA companies such as Xilinx [4] and Altera [2] have attempted to address the macro-block issue at the architectural level by increasing the logic block grain-size of devices to include multiple look-up table/ flip-flop pairs. This architectural approach has limitations, however, as Betz [13] has shown that devices with logic blocks containing more than 4 lookup-table, flip-flop pairs per block become prohibitively expensive in terms of device area compared to implementations with finer-grained logic blocks and a distributed routing plane.

A potential architectural path for next-generation FPGA devices includes the development of *partitioned* island-style architectures [47]. As seen in Figure 7-4, these architectures overcome many of the design impediments to fast compilation discussed in this thesis by quantizing a large, homogeneous array of logic and routing resources into a tree of island-style sub-arrays. This type of implementation has the

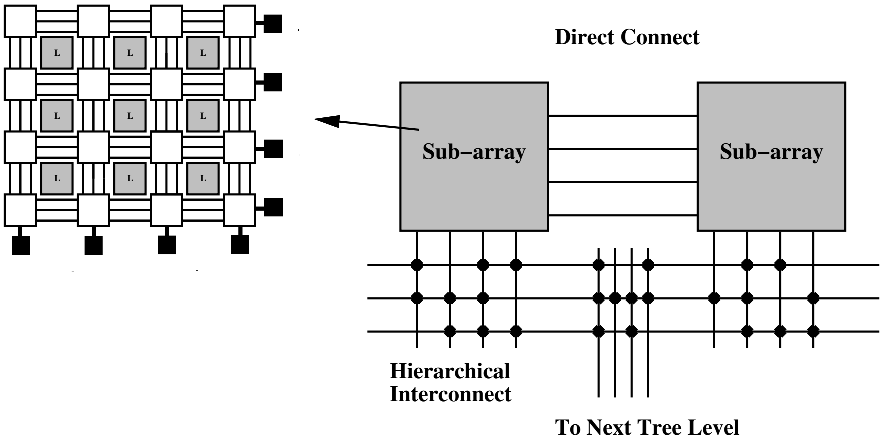

Figure 7-4: Partitioned Island-style Device

following potential benefits for fast compilation:

- Existing island-style technology can be leveraged by mapping macro-blocks to sub-arrays. If a macro-block requires more than one sub-array, *direct connect* lines can be used to provide near-neighbor, inter-array bandwidth.
- Since there are well-defined architectural boundaries between sub-arrays, issues such as contention between macros for specific wire segments at macro boundaries and contention for wire segments that span multiple logic blocks, as discussed in Section 5.6.2, no longer exist.
- Tree-like interconnect provides opportunities for isolation between macros. Each signal that emanates from a sub-array can connect to multiple inter-array wires, reducing the domain matching issues addressed in Chapter 6. Hierarchical isolation of a portion of the circuit can be achieved at each internal node of the routing tree.

Initial work on hierarchical routing architectures for FPGAs [6] [34] has focussed on tree-like interconnection structures with small numbers of logic blocks (4-16) at tree leaves in an effort to minimize FPGA delay. For these structures to be effective for macro-based designs, coarser-grained sub-arrays will be needed.

Hardware Support for Scheduled Communication

As mentioned earlier, a fundamental issue affecting place and route time for any FPGA architecture is the growth rate of inter-macro interconnect due to Rent's Rule limitations. As shown in [21], hierarchical interconnect structures are particularly limited by wire growth due to a high-density switching bottleneck at each tree node. To reduce this bandwidth, scheduled communication between macros becomes a necessity. In general, using look-up tables and flip-flops in technology native to the FPGA to create hardware support for scheduled communication limits feasible bandwidth to the same clock rate as the base FPGA logic design. By implementing special purpose, time-multiplexed structures at sub-array I/O points, rapid inter-array communication can likely be achieved and fewer inter-array wires will be needed.

Bibliography

[1] *Configurable Logic Design and Application Book.* Atmel Corporation, 1994.

[2] *Altera Data Book.* Altera Corporation, 1996.

[3] *Field-Programmable Gate Arrays Data Book.* Lucent Technologies, 1996.

[4] *The Programmable Logic Data Book.* Xilinx Corporation, 1996.

[5] *NAPA 1000 Adaptive Processor.* National Semiconductor Corporation, 1998.

[6] A. Aggarwal and D. Lewis. Routing Architectures for Hierarchical Field-Programmable Gate Arrays. In *Proceedings IEEE Custom Intgerated Circuits Conference*, 1994.

[7] J. Babb and M. Frank. Solving Graph Problems with Dynamic Computation Structures. In *SPIE Photonics East: Reconfigurable Technology for Rapid Product Development and Computing*, Boston, Ma, Nov. 1996.

[8] J. Babb, M. Frank, V. Lee, E. Waingold, and R. Barua. The RAW Benchmark Suite: Computations Structures for General Purpose Computing. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1997.

[9] J. Babb, R. Tessier, M. Dahl, S. Hanono, and A. Agarwal. Logic Emulation with Virtual Wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 609–626, June 1997.

[10] P. Bertin, D. Roncin, and J. Vuillemin. Introduction to Programmable Active Memories. *DEC Paris Research Lab Technical Report No. 3*, June 1989.

[11] V. Betz. Personal communication.

[12] V. Betz and J. Rose. On Biased and Non-Uniform Global Routing Architectures and CAD Tools for FPGAs. *University of Toronto Department of Electrical Engineering, Technical Report*, June 1996.

[13] V. Betz and J. Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. In *Proceedings, IEEE Custom Integrated Circuits Conference*, pages 551–554, 1997.

[14] V. Betz and J. Rose. VPR: A New Packing, Placement, and Routing Tool for FPGA Research. In *Proceedings, Field Programmable Logic, Seventh International Workshop*, Oxford, UK, Sept. 1997.

[15] S. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publishers, Boston, Ma, 1992.

[16] P. Chan, D. Schalg, and J. Sien. On Routability Prediction for Field Programmable Gate Arrays. 1993.

[17] C. E. Cheng. RISA: Accurate and Efficient Placement Routability Modeling of Macro Cells. In *ICCAD*, 1994.

[18] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object Oriented Circuit-Generators in Java. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, Ca, Apr. 1998.

[19] G. W. Clow. A Global Routing Algorithm for General Cells. In *Proceedings, ACM/IEEE 21st Design Automation Conference*, 1984.

[20] W. Dees and R. Smith. Performance of Interconnection Rip-up and Reroute Strategies. In *Proceedings, ACM/IEEE 18st Design Automation Conference*, 1981.

[21] A. Dehon. *Reconfigurable Architectures for General-Purpose Computing.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1996.

[22] S. Devadas. *Logic Synthesis.* McGraw-Hill, New York, NY, 1994.

[23] W. E. Donath. Placement and Average Interconnect Lengths of Computer Logic. *IEEE Transactions on Circuits and Systems*, CAS-26(4), Apr. 1979.

[24] A. Duncan, D. Hendry, and P. Gray. An Overview of the COBRA-ABS High Level Synthesis System. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, CA, Apr. 1998.

[25] C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Design Automation Conference*, May 1984.

[26] J. Frankle. Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing. In *Proceedings, ACM/IEEE 29st Design Automation Conference*, 1992.

[27] A. E. Gamal. Two-Dimensional Stochastic Model for Interconnections in Master Slice Integrated Circuits. *IEEE Transactions on Circuits and Systems*, CAS-28, Feb. 1981.

[28] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweeney, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *Computer*, 24(1), Jan. 1991.

[29] S. Goto. An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout. *IEEE Transactions on Circuits and Systems*, CAS-28, Jan. 1981.

[30] S. Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1995.

[31] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An Efficient General Cooling Schedule for Simulated Annealing. In *ICCAD*, 1986.

[32] A. Koch. Structured Design Implementation - A Strategy for Implementing Regular Datapaths on FPGAs. In *International Symposium on Field Programmable Gate Arrays*, Monterey, Ca., Feb. 1996.

[33] B. Krisnamurthy. An Improved Min-Cut Algorithm For Partitioning VLSI Networks. *IEEE Transactions on Computer-Aided Design*, CAD-33:438–446, May 1984.

[34] Y.-T. Lai and P.-T. Wang. Hierarchical Interconnection Structures for Field Programmable Gate Arrays. *IEEE Transactions on VLSI*, pages 186–196, June 1997.

[35] B. Landman and R. Russo. On a Pin Versus Block Relationship For Partitions of Logic Graphs. *IEEE Transactions on Computers*, C-20(12), Dec. 1971.

[36] C. Lee. An Algorithm for Path Connections and its Applications. *IRE Transactions on Electronic Computers*, Sept. 1961.

[37] G. Lemieux, S. Brown, and D. Vranesic. On Two-Step Routing for FPGAs. In *Proceedings: International Symposium on Physical Design*, Napa, Ca., Apr. 1997.

[38] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, New York, NY, 1990.

[39] W. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mercer, J. Morris, K. Palem, V. Prasanna, and H. Spaanenburg. Seeking Solutions in Configurable Computing. *IEEE Computer*, 30(12):38–43, Dec. 1997.

[40] L. McMurchie and C. Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *International Symposium on Field Programmable Gate Arrays*, Monterey, Ca., Feb. 1995.

[41] S. Nag and R. Rutenbar. Performance-Driven Simulataneous Place and Route for Island-Style FPGAs. In *ICCAD*, 1995.

[42] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing, Palo Alto, Ca., 1980.

[43] R. Otten. Efficient floorplan optimization. In *Proceedings of International Conference on Computer Design.* IEEE Computer Society Press, 1983.

[44] J. Peterson, R. O'Connor, and P. Athanas. Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures. In *Proceedings, IEEE Workshop on FPGA-based Custom Computing Machines*, Napa, CA, Apr. 1996.

[45] R. Prim. Shortest Connecting Networks and Some Generalizations. *Bell Syst. Tech. J.*, 1957.

[46] J. Rose, R. Francis, D. Lewis, and P. Chow. Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid State Circuits*, 25(5), Oct. 1990.

[47] J. Rose and D. Hill. Architectural and Physical Design Challenges for One-Million Gate FPGAs and Beyond. In *5th International Workshop on Field-Programmable Gate Arrays*, Monterey, Ca, Feb. 1997.

[48] Y. Sankar. Fast Placement for FPGAs. In *Proceedings: University of Toronto FPGA Research Review*, Peterborough, Ontario, June 1997.

[49] Y. Sankar. Ultra-Fast Placement for FPGAs. Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, 1998. in preparation.

[50] C. Sechen. *VLSI Placement and Global Routing Using Simulated Annealing.* Kluwer Academic Publishers, Boston, Ma, 1988.

[51] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid State Circuits*, 20:510–522, 1985.

[52] N. Sherwani. *Algorithms for Physical Design Automation.* Kluwer Academic Publishers, Boston, Ma, 1992.

[53] J. Shi and D. Bhatia. Performance Driven Floorplanning for FPGA Based Designs. In *International Symposium on Field Programmable Gate Arrays*, Monterey, Ca., Feb. 1997.

[54] J. Swartz. Rapid Routing for FPGAs. In *Proceedings: University of Toronto FPGA Research Review*, Peterborough, Ontario, June 1997.

[55] J. Swartz. A High-Speed Timing-Aware Router for FPGAs. Master's thesis, University of Toronto, Department of Electrical and Computer Engineering, 1998. in preparation.

[56] J. Swartz, V. Betz, and J. Rose. A Fast Routability-Driven Router for FPGAs. In *6th International Workshop on Field-Programmable Gate Arrays*, Monterey, Ca, Feb. 1998.

[57] J. Swartz and J. Rose. Personal communication.

[58] W. Swartz and C. Sechen. New Algorithms for the Placement and Routing of Macro Cells. In *ICCAD*, 1990.

[59] R. Tessier. Negotiated A* Routing for FPGAs. In *Proceedings: Fifth Canadian Workshop on Field-Programmable Devices*, Montreal, Quebec, June 1998.

[60] S. Trimberger. *Field-Programmable Gate Array Technology.* Kluwer Academic Publishers, Boston, Ma, 1994.

[61] Y.-L. Wu and D. Chang. On the NP-completeness of Regular 2-D FPGA Routing Architecture and a Novel Solution. In *ICCAD*, 1994.

[62] T. Yamanouchi, K. Tamakashi, and T. Kambe. Hybrid Floorplanning Based on Partial Clustering and Module Restructuring. In *Proceedings, ACM/IEEE 33rd Design Automation Conference*, 1996.