UNIVERSITY OF CINCINNATI

Date: March 3, 2006

I, RENQIU HUANG **,**

hereby submit this work as part of the requirements for the degree of:

PHD

in:

COMPUTER SCIENCE AND ENGINEERING

It is entitled:

PHYSICAL AWARE HIGH LEVEL SYNTHESIS AND INTERCONNECT FOR FPGAS

This work and its defense approved by:

Chair: RANGA VEMURI

HAROLD CARTER

KAREN TOMKO

WEN-BEN JONE

JINTAI DING

Physical Aware High Level Synthesis and Interconnect for FPGAs

A Dissertation submitted to the

Division of Research and Advanced Studies

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

by

Renqiu Huang

Thesis Advisor and Committee Chair: Dr. Ranga R. Vemuri

ABSTRACT

Reconfigurable computing (RC) is going mainstream where FPGA plays an essential role. Synthesizing the application from concept and prototyping onto reconfigurable FPGAs has emerged as one of the main challenges in design automation area. A large number of new applications show the huge potentials of synthesis strategy and architecture development for FPGAs. The work presented in this dissertation deals with the synthesis and novel architecture of FPGAs. In particular, it tries to address physical aware high level synthesis (PAHLS) methodology to ensure the synthesis integrity for FPGAs. Motivated by the study of PAHLS, a hybrid interconnect structure is proposed to increase the performance and reconfigurability for FPGAs or FPGA-like reconfigurable platforms.

We first present a performance-driven PAHLS where relational placement is combined with the macro generation strategy during high level synthesis. Second, we present an automated framework to integrate physical placement information into high-level synthesis that is believed to be the first on-line synthesis methodology for partially reconfigurable FPGAs. The presented synthesizer allocates the FPGA resources adaptively and is incremental in nature. The algorithm is designed to be linear in terms of the number of operations to ensure its on-line usage. We then present a transformation mechanism to extend the synthesis frontier to heterogeneous configurable architectures. We develop an automatic synthesis methodology which attacks both memory and logic assignments by interacting with behavioral synthesis. Next, we present a hybrid interconnect structure which takes advantages of both mesh and tree interconnect topologies. The presented architecture is investigated with a combinatorial analysis which examines the number of switches needed. Our evaluation demonstrates that the presented model has less switch accrued effects due to the introduction of tree networks. Finally we extend that hybrid interconnect structure to support multi-granular configuration. We also develop a fast evaluation tool to simulate on-line placement and routing effects by applying that interconnect on a run-time reconfigurable platform. The studies show the efficiency of the extended model in overcoming the fragmentation problem with a penalty of modest increase in the number of switches for the construction of that interconnect.

ACKNOWLEDGMENTS

I express my deep gratitude to Dr. Ranga Vemuri, my research and thesis advisor, without his guidances and supports this thesis would not have been possible. I am very grateful to Dr. Vemuri for his role in guiding me through my research. I clearly remember that in September 2003 Dr. Vemuri was working with us day and night to prepare the papers for DATE. Special thanks must go to Dr. Vemuri for his ideas, comments and suggestions during my work. His in-depth knowledge and dedicated attitudes toward research always inspire me to pursue the best.

My sincere thanks are due to Dr. Harold Carter, Dr. Jintai Ding, Dr. Wen-ben Jone, and Dr. Karen Tomko for being part of my dissertation committee and for their valuable suggestions. This work was sponsored in part by the Dayton Area Graduate Studies Institute (DAGSI) and the Air Force Research Laboratory (AFRL) research program under contract number IF-UC-00-07.

I also want to thank the reviewers of my papers during last several years. They are: Manish Handa, Jawad Khan, Madhubanti Mukherjee, Balasubramanian Sethuraman, Vikas Vijay, Harish Vutukuru and Jayanthi Rajagopalan. I had a good time working with the Reconfigurable Computing team: Bala, Jawad, Madhu, Manish, Vijay and Xin. The presentation and discussion of weekly seminar are memorable. Thanks also to the independent and anonymous reviewers outside for their comments and expert counsel.

I had an enjoyable time with all DDELites. It was good fun getting to know the DDEL folks: Madhubanti Mukherjee, Vijay Sundaresan, Glenn A. Wolfe, Jawad Khan, Manish Handa, Mukesh Ranjan, Raoul Badaoui, Anuradha Agarwal, Sunder Rajan Kankipati, Xin Jia, Mengmeng Ding, Huiying Yang, Vipul Patel, Vikas Vijay, Jayanthi Rajagopalan, Shyam Sundar Balasubramanian, Amitava Bhaduri, Balasubramanian Sethuraman, Harish Chander Rao Vutukuru. Life at DDEL was fun and it will be fondly remembered when I think of Cincinnati.

I acknowledge the love and support of my family who are more or less my counterpart. I especially would like to dedicate my work and everything I have achieved so far to them most of whom I have not seen for several years and was missing a lots when studied at University of Cincinnati.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As design cycle is shortening and design complexity is increasing, system designers have moved to higher levels of abstraction to enable larger systems to be described and more powerful computer-aided design tools to be applied. It is then imperative to develop a design methodology that automates design from conceptualization to silicon and helps designers to build the required complexity in short time-to-market.

Synthesis is viewed as a process of design conversion and enhancement, which, while satisfying the given design constraints, provides much more detail implementation message for the original description. There are normally three level synthesis stages in the design flow hierarchy: behavioral level, logic level and layout level synthesis Fig.1.1(b). Each synthesis step puts in an additional level of decisions and adjustments which influence the final accomplishment and offer information considered necessary for the next level of synthesis or for fabrication of the design.

High level synthesis, also called as behavioral synthesis or architectural synthesis (henceforth we will use high level synthesis and behavioral synthesis interchangeablely), has been a very hot research issue over the pass two decades, which is defined as translation process from a behavioral specification into a structural or register-transfer-level (RTL) description. Each component in the structural description is in turn defined by its lower-level elaboration. The primary advantage of applying high level synthesis is to shorten the design cycle with exploring more design space. This

is benefited from the feature of HLS: ease of specification and verification of design. There are an extremely large number of potential implementations even for a moderately complex design for todays very large scale integrated circuits (VLSI) technology, and further, large applications like as multimedia, networking applications require enormous amounts of simulation to verify their functionality. Since at RTL level simulation is painfully slow and finite state machines (FSM) for memory or control subsystems are becoming extremely complex, HLS could out-perform it in exploring architectural trade-off because different designs can be generated and multiple hardware implementations can be evaluated quickly.

Despite more than twenty years of research and many applications, high level synthesis lacks a general acceptance in the electronic system design community and it is still not moved into mainstream design practice. The main reason is largely due to the low quality of designs produced by these tools. This low quality is implied through two characterizes: *uncontrollability* and *unpredictability* which were introduced from HLS, but shown in the final layout realization. The former is due to the independent relationship between HLS and layout synthesis stages. There is no concern on how we could make a decision to guide the layout feasible as we dont know whether the decision made at HLS achieves better place and route result or not. The latter is owed to the constraints are met until at the end of the time-consuming phase of place and route. If a problem occurs later, we dont know which judgment or decision made at HLS causes that problem.

During high level synthesis, in order to simplify the HLS algorithms and be flexible for technologies, generic abstract models of hardware, which have little or no physical basis information, are used. Lacks of physical implementation information and layout effects obviously make the design synthesized poorly from the start. In deep sub micrometer (DSM) processes, where layout effects traditionally thought about as second-order have got to require much consideration, the problem will get worse without considering physical design in high level synthesis.

In order to overcome the inefficiency mentioned above, incorporating the physical information into high level synthesis is desirable Fig.1.1(c). Generally, the higher levels of the design cycle can make more profound impacts on the qualities of the design than the lower levels can. So, by

making high-level decisions that are more consistent with the final implementation, a high quality solution may be produced while still in the behavioral format and the convergence to a desired solution would be faster. For example, the loop, denoted by dotted line in Fig.1.1(c), may not be necessary. For HLS algorithms to make effective decisions that eventually result in high-quality layouts, we need to incorporate physical design information during HLS. We must account not only for place and route effects, but global considerations as well, such as RT wiring, register organization, component styles, aspect ratio, floorplanning, and the combination of "all of the above" [2]. Every step of HLS should take layout into account [3].

Since we are interested in the synthesis integration, we concentrate our focus on the physical aware high level synthesis for FPGAs. The primary characteristics of synthesis flow are described in the next section. We then survey the recent approaches for synthesis integration with various strategies: design representation, optimization metrics, estimation technologies and FPGA's applications. Finally, the last part outlines the main works in this proposal.

1.1 Synthesis Issues

Similar to the compilation of a high-level language program in C or Pascal into an assembly program, HLS performs "hardware compilation", which generates hardware circuits from a high-level input description. However, the result of high-level synthesis is no transistor layout yet, but rather a so called register-transfer description which again has to be processed by subsequent synthesis steps. Normally there are three subtasks in HLS: scheduling, allocation, and binding Fig.1.1(a). Scheduling assigns operations of the behavioral description into control steps, sometimes it determines which operations are to be performed in which clock cycle because a control step usually corresponds to a cycle of the system clock. The responsibility of high-level synthesis is to perform optimization steps which would be carried out through resource allocation and operation binding. Allocation chooses functional units and storage elements from the component library and binding assigns operations to functional units, variables to storage elements, and data transfers to wires or

buses. Due to resource limitation and scheduled steps, it is critical to decide how many modules and which kind of modules are needed to achieve the better area performance tradeoff without violating the resource limitations. Further, program transformations like loop unrolling or function inlining may also be performed by high-level synthesis.

In most methodologies, the generated RTL netlist from HLS is then submitted to logic synthesis for gate-level optimization. The goal of logic synthesis is to produce a circuit that satisfies a set of logic equations, occupies minimal silicon area and meets the timing constraints. Most logic synthesis systems currently available split this task into two phases: technology independent phase and a technology dependent phase. In the first phase, transformations are applied on a Boolean network to find a representation with the least number of literals in the factored form. Additional timing optimization transformations are applied on this minimal area network to improve circuit performance. The role of the technology dependent phase is to finish the synthesis of the circuit by performing the final gate selection from a target library. The technology-dependent phase is, to a large extent, constrained by the structure of the optimized Boolean network. It is assumed that wiring optimization can be handled efficiently in the physical design phase.

Physical design converts a circuit description into a geometric description. This description is used to manufacture a chip. The physical design cycle consists of partitioning, floorplanning, placement and routing, compaction as well. Partitioning decomposes a complex system into smaller subsystems, each subsystem can be designed independently speeding up the design process. There are two primary goals. one is to minimization the interconnections between the subsystems, and the other is that the sizes of decomposed subsystems are manageable. The most objectives of floorplanning and placement are to minimize area, determine shapes of flexible blocks and reduce netlength for critical nets. The routing is to assign the connection route for each net such that the total netlist is routable with performance satisfaction. Normally, routing is performed at two stages: global routing and detailed routing.

Here we only briefly introduce the fundamental characteristics of synthesis framework. The purpose of the framework is to make our following reviews on physical aware high level synthesis in

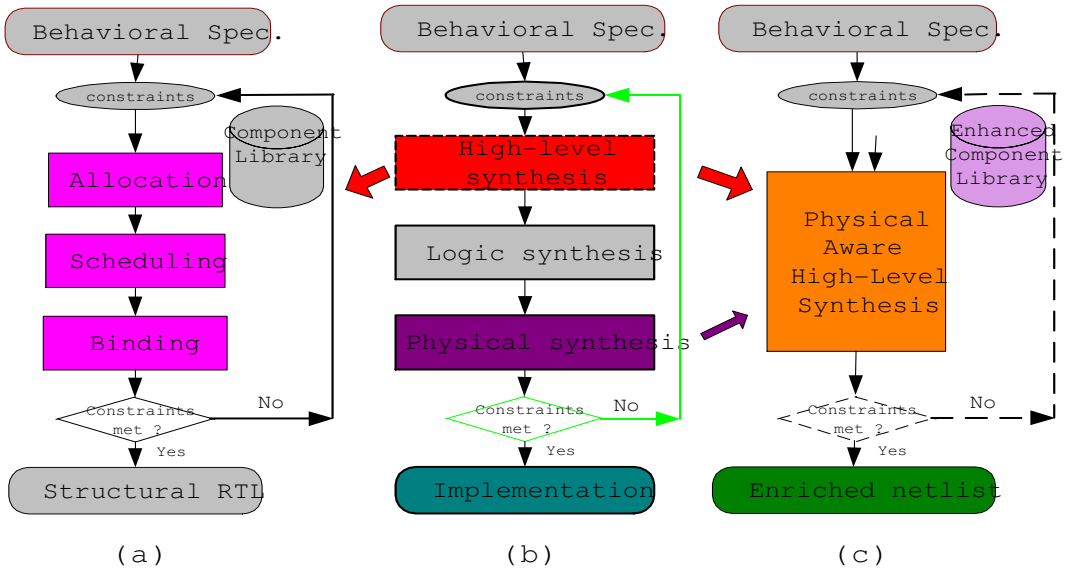


Figure 1.1: Physical aware high level synthesis

a more systematic way and hence it is colored to suit our discussion. More basic techniques and detailed algorithms of each subtask in the synthesis could be found in the literature [4, 5, 6, 7, 8]. Next, we look at the recent developments or research trends for the synthesis integration.

1.2 Recent Synthesis Efforts

During high level synthesis, the synthesizer takes the key decisions on algorithms (i.e., how to schedule the specified operations), and hardware architectures (i.e., what is the hardware resource support required for binding the functionality), but it does not deal with implementation details, which are left to the physical design phase.

One way to account for layout information in HLS is to actually go through a physical design procedure each time a candidate solution is generated, shown in the loop of Fig.1.1(b). The PEPPER [9] analysis tool performed an estimation of delay by explicitly performing tasks of placement and routing. Obviously, one drawback of such a design methodology is that it is an extremely time-consuming task. Further, designers may resist changing the source specifications because then they would have to resimulate. Once the design is simulated, the architect may expect to achieve area

and timing improvement through interacting with physical design tasks. Su et al. [10] have tried to use the post layout timing information to guide the resynthesis of soft macros with expectation of producing high area-efficient designs while satisfying the timing constraints. "It takes close to 1 full day to run one resynthesis iteration." Much effort should be invested to shorten the number of resynthesis iterations and thus speed up the entire design process.

1.2.1 New Design Representation

The input of HLS is behavioral description which may be specified in the format as C, VHDL or Verilog, etc. For the convenience of processing, that behavioral description is compiled into an internal representation as the beginning step of HLS. A control/data flow graph (CDFG) is a commonly used internal representation to capture the behavior of design. The control-flow graph (CFG) portion of the CDFG captures sequencing, conditional branching, and looping constructs in the behavioral description, and the data-flow graph (DFG) portion captures data-manipulation activity described by a set of assignment statements (operations).

In traditional HLS systems, the nodes of DFG represent operations and the directed edges represent data dependencies between operations, the synthesis is performed in terms of operations. A CDFG represents the specification of the design at a language level, rather than the final hardware level that it is trying to implement. Tarafdar et al. [11] formulate scheduling and binding problem based on data-transfer (DT) model. DT is a set of operations corresponding to the movement of a single instance of data. It contains the operation sourcing the data and all the operations using the data.

In the traditional DFG, an external time frame structure is required where the DFG can be manipulated to perform scheduling and binding simultaneously [12, 13]. Although Midas [11] emphasized the communication through the data-transfer graph, it did not allow operations in two or more synthesis domains to occur together. Bergamaschi's behavioral network graph (BNG) [14] introduced special nodes between operations representing potential state cuts. Deciding whether the node will become a true state cut relies on constant propagation from special input pins during logic synthe-

sis. This representation, consisting of a novel internal model for synthesis which spans the domains of high-level and logic synthesis, allows simultaneous scheduling and allocation. Dougherty and Thomas [15] extended that structure to BNG-style state cut nodes (SCNs). Each DFG operation has an associated set of shapes representing the dimensions and delays of potential hardware implementations, which incorporate strong ties to physical design early in synthesis process while yielding a high degree of simultaneous scheduling and allocation flexibility throughout.

Most of above works are targeting the data-flow dominated behavioral descriptions, while treating the control-flow synthesis separately. Nevertheless, handling control dependencies enables more precise scheduling and ultimately better quality solution [16]. Kountouris et al. [17] presented a unifying intermediate design representation appropriate for both subsets of behavioral descriptions. This hierarchical conditional dependency graph (HCDG) represents control and data dependencies at the same level of detail from a pure dataflow perspective, the well-established DFG techniques and algorithms are readily applicable on HCDGs. By elaborating control and data flows together in a unified framework, HCDGs can accommodate sub-systems targeting different implementation domains, such as software-hardware codesign. Doboli [18] presented another hybrid representation for hardware software co-design of low power embedded systems. The description hierarchical graph expresses data and control dependency among two kinds of nodes: the operation nodes for hardware implementation and cluster nodes for software execution.

To rapidly explore the design space, some systems are characterized in task graph format. Task graph model abstracts system functionality into a set of tasks represented as nodes in a graph, and represents functional dependencies among tasks with graph edges. The task graph emphasizes communication and concurrency between system tasks. Edge and node labeling are used to enrich the semantics of this model. Along with an architecture template, fast resource binding and task scheduling are feasible. Sblox (Serial Blocks with Black Box Operations) [19] belongs to this category. Task graph model, due to its simplified, abstract view of the system, focuses on only some facets of system. Each task node may be enhanced through subgraph representation which gives more detail implementation of that task. Applications can be found in [20, 21].

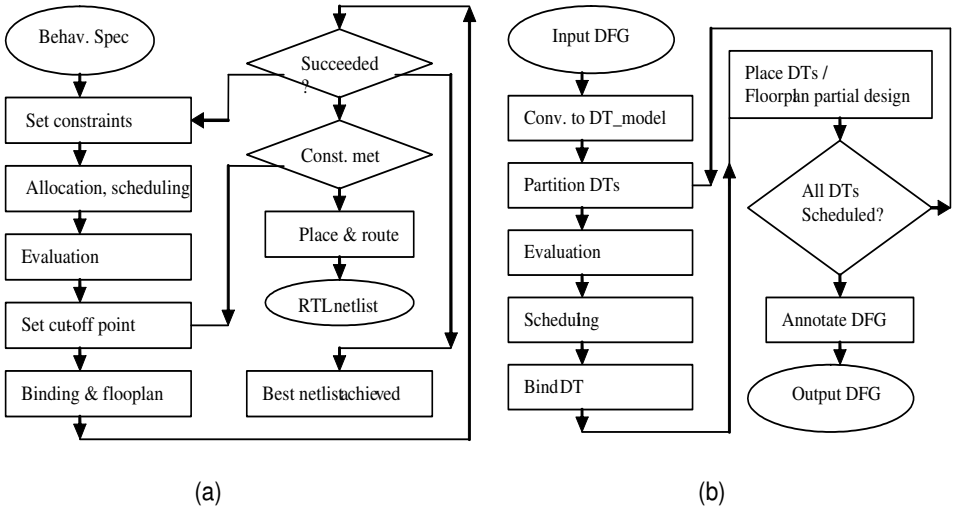


Figure 1.2: Two synthesis flows

1.2.2 Floorplanning Aware Synthesis

Floorplanning is the highest level of the physical design process with which we can evaluate different performance and cost measures of the design accurately. A few researchers have proposed methods to efficiently incorporate cost metrics for floorplanning into architectural synthesis. 3D scheduling [22] described an algorithm for simultaneous scheduling, binding and floorplanning. They performed the floorplanning of modules in critical paths, followed by an iterative improvement phase which performs rebinding to reduce the latency. Jang and Pangrle [23] used a grid-based connectivity binding approach and considered the minimization of interconnection lengths with the assumption of the bit-sliced stack architecture. Fang and Wong [24] described an integrated binding and floorplanning algorithm which performs a constructive binding for each move of a simulated annealing based floorplanning algorithm. In [25], a heuristic for area estimation was presented, which bases on performing module- and register-binding. This step is followed by an approximate floorplanning through taking into consideration the possible bindings of the operations to different resources in high-level synthesis.

Taking floorplanning into account during high level synthesis has also attracted a lot of attention. Fig.1.2(a)(b) show two synthesis flows corresponding to [2] and [11] respectively. Tarafdar et al. [11] used two floorplanners to provide the shapes and placement of components and buses

very early in the high-level synthesis flow. The global floorplanner generates the partial floorplan for the partial architecture. The incremental floorplanner modifies an existing floorplan based on the changes in the architecture caused by recent scheduling and binding actions. The result is a high level floorplan and an architecture in which data is stored close to where it is produced and used. Prabhakaran et al. [26] presented a simulated annealing algorithm which simultaneously optimizes the latency of a schedule and the overall area of the floorplan. The developed floorplanning algorithm takes into account the effect of interconnect delays on the overall cycle time of a given schedule. Further, in [27], the switching activity on CDFG edges was profiled. It was used with the floorplanner to optimize the power consumption of inter-module data transfers. Su et al. [10] presented a complete chip design method which incorporates a floorplanning-guided soft macro resynthesis method for area and timing improvement. During each design iteration, soft macros have been resynthesized with either a relaxed or a tightened timing constraint which is guided by the post-layout timing information. Choi and Levitan [28] integrated floorplanning tightly into the data-path allocation algorithm with consideration of the number of routing tracks as well as the number of functional units. An approach was also developed to estimate the number of routing tracks and the longest wire length accurately for a bit-sliced stack and random topology architectures.

1.2.3 Placement Aware Synthesis

All design automation systems, independent of the domain, must eventually produce a physical artifact with real physical characteristics. One common procedure of simultaneously performing floorplanning with high-level synthesis is: find a floorplan first; then evaluate a cost function that considers area, wire length, interconnection or the critical path delay; perform re-scheduling, re-binding or both. This procedure is performed iteratively until the best solution among many iterations of the above procedure is chosen. The iterative nature of those approaches is the timing expensive; further, the next place-and-route phase makes this situation worse.

Since behavioral decisions and transformations are represented by the acting on objects, (i.e., nodes

or vertices of a dataflow graph), in the behavioral model, the layout effects may be viewable by extending that objects placeable such that all the information is available in the graph throughout the process while the design is still in the behavioral format. CPR [29] developed an architecture specific macro library containing pre-placed and routed parameterized macros to replace a group of nodes in a dataflow graph. Moshnyaga et al. presented the combination of architectural synthesis with performance-driven placement in [30, 31]. They incorporated the wiring dedicated transformations onto the physical driven synthesis. These transformations are united with architectural transformations to provide space exploration mechanism. Xu and Kurdahi [2] used slicing trees to determine how operations were to be bound to a fixed number of functional units. With the help of a chip level area and performance estimator, the final result was evaluated without actually going through the time consuming phase of place and route.

Constructive placement is also adopted to manipulate the quality of the high level synthesis, which allows the behavioral and physical decisions to be made simultaneously. Because it is a constructive technique, design considerations in high level synthesis and physical design are being solved at once. No iterating between the behavioral and physical designs needs to occur. Dougherty et al. [15] attempted to unify physical design and high level synthesis. Each DFG operation has an associated set of shapes representing the dimensions and delays of potential hardware implementations. Using shape information, the physical design process operates directly on the behavioral DFG bypassing the need to create intermediate RTL and gate-level netlists. The placement process determines the locations and shapes producing the best physical design, and by doing so implicitly schedules, allocates, and maps the design at the same time. Kim et al. [32] used the metric of inter-cycle slack to be incorporated into architectural synthesis for distributed-register architecture. The placement algorithm used the concept of window, which combines timing and geometric constraints for performance optimization.

The execution time of above synthesis is greatly shortened. However, the effectiveness of this kind of constructive methodology is largely dependent on the evaluation of cost function. In terms of layout quality, this approach relies on the accurate and efficient prediction of design metrics, which

reduces the run-time (no iterative search) to evaluate the design solution. Hence, it is important to develop the prediction or estimation technique for the fast and accurate evaluation and we survey it in the next subsection.

1.2.4 Evaluation Techniques

Several estimation approaches have been proposed in the past. In the CADDY system [33], area/delay estimations were based on corresponding models in the module library, which has been completed before the real synthesis process starts. In [34], delay was based on the wire-length estimation through a combination consideration of analytical and constructive layout effects. Techniques to estimate storage requirements from a behavior were proposed in [35].

In order to produce an efficient RTL network, HLS has to estimate or compute the effect that a given high-level algorithmic decision will have on the final layout implementation. This effect is translated into costs based on the number of states and number of resources which are used in most HLS algorithms, such as scheduling, allocation and resource sharing. These cost metrics give a rough indication of the complexity and performance of the finite-state machine (FSM) and datapath area of the final design. Examples are the use of the number of literals [36] or gates [37, 38] to predict area in logic synthesis and the number of execution units to predict area in high level synthesis [39, 40].

Tiruvuri and Chung [41] estimated completion time of partially scheduled dataflow graphs in reducing the size of the search space when used in a branch and-bound scheduling algorithm. Ohm et al. [42] considered the dependencies among different types of resources. Their estimation was extended to the physical level which accounts for layout effect on both components (e.g. variations in area and delay with component shape and aspect ratio), and chip level designs (e.g. wiring effects, unused area) based on the previous work in [43]. Bazargan et al. [44] weighted the potential connections between any two resources based on the likelihood that such a connection is used in the final design, and did the floorplanning in a way that the nets with more probability end up hav-

ing smaller length. Gelosh and Steliff [45] modeled the layout tool, rather than the layout result, to capture the relationships between general design features and layout concepts through machine learning techniques.

Increasing complexity of deep-submicron is bound to make these objective functions more irrelevant. Also, the need for predicting more complex design metrics, such as power [46, 47] and testability, provide additional impetus to study objective functions during synthesis. Parulkar et al. [48] presented lower bounds on the number of test resources required to test a synthesized data path using built-in self-test (BIST). The estimations were performed on scheduled data flow graphs with a given module assignment and schedules such that the resulting synthesized data path requires a small number of BIST resources to test itself. Diguet et al. [49] dealt with the rapid prototyping of digital signal processing applications in which the cost/signal processing quality trade-offs can be achieved while changing the type of algorithm and the number of filter taps for a given algorithm specification through the cost estimation of processing units. So et al. [50] used the performance and area estimates from high level synthesis to guide the compiler which optimizes a design to increase parallelism through code transformations.

The optimization algorithm is proportional to the accuracy of the targeted objective functions. Most of evaluation techniques concentrate on the datapath synthesis by considering controller separately. During datapath synthesis, ignoring the controller aspects such as the size and delay of the control logic, multiplexers and registers makes it impossible for any scheduling, allocation or resource sharing algorithm to produce optimal results. Few researcher takes the influence of the controller on the overall area of a design into account. Katkoori et al. [51] and Menn et al. [52] were able to predict the area of a controller, based on information, which are available before the controller actually is generated.

1.2.5 Memory

Many behavioral descriptions for manipulating large amounts of data computations use array variables to represent data storages. Traditionally, variables are grouped into registers, and registers into register files (memory modules). Consequently, high level synthesis is required to allocate memory modules for implementing the array variables. In current systems-on-chip (SoCs), memory takes, more than half of the silicon real-estate. It is necessary to automate the process of finding the best memory configuration in terms of both total memory cost and design performance.

Instead of random-access registers, Carlos et al. [53] constructed a conflict graph that represents the relative overlap of value instances, performing a partial schedule that can be completed by a conventional scheduler without violating the storage file constraints. The library mapping problem for memories, which consists of mapping generic memories to specific memory modules present in a library, was addressed in [54]

Huang et al. [55] explored the partitioning of arrays (which were considered to be atomic generally) into smaller partitions and focused on joint computation and data partitioning to result in distributed logic-memory architectures. Memory binding techniques for control-flow intensive behaviors were presented in [56]. All the works described above are restricted to static arrays in behavioral descriptions. Memory optimizations for more complex abstract data types, and dynamically allocated memory, were described in [57, 58].

With the increasing design complexity and performance requirement, data arrays in behavioral specification are usually mapped to fast on-chip memories in high level synthesis. Seo et al. [59] described the non-uniform access speeds among the ports of memories during memory exploration. Memory-intensive behaviors often contain large arrays that are synthesized into off-chip memories. Since off-chip memory access is a relatively slow operation, Panda et al. [60, 61] incorporated off-chip memory interface protocols into high level synthesis by anticipating the improvement of schedule length for typical off-chip memories, such as dynamic random-access memory (DRAM).

With the fast growth of semiconductor technology and consequent increase of the level of integra-

tion, memory size (both for on-chip and off-chip memories) is no longer the main optimization issue. Memory performance and power consumption are now the key challenges in system design. Techniques for mapping multiple arrays to memories while reducing power consumption were described in [62, 63].

Physical partitioning of embedded memories has been analyzed by several authors. Benini et al. [64] described an application-driven partitioning of on-chip SRAMs based on a recursive formulation. The method explicitly accounts for the overhead induced by the partitioning. The partitioning engine was integrated inside a comprehensive framework that links the partitioning algorithm to the physical design phase.

1.2.6 FPGA or Reconfigurable Computing

The extreme flexibility and the growing capacity of Field Programmable Gate Arrays (FPGAs) has made them the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines.

Mapping computations to FPGA-based architectures is a lengthy and error prone process. So et al. [50] developed a compilation system that automatically maps high-level algorithms to application specific designs for FPGAs. Focusing on mapping a loop nest computation to a single FPGA with multiple external memories, they exploited instruction level and memory parallelism using loop unrolling. As a result, the logical operations and their corresponding operands in the loop body were replicated and exposed to high level synthesis optimizations.

For a scheduled dataflow graph, Xu et al. [2] constructed a fully connected netlist in which each function unit was connected to every register and each register was connected to every function unit. During binding operation, layout effects estimated for Xilinx XC4000 family FPGAs were incorporated to generate more predictable implementation.

To apply standard design and verification tools to the design of dynamically reconfigurable logic, Robinson and Lysaght [65] partitioned the design at abstract level and translated a dynamic designs

from VHDL into placed and routed circuits. Since the design was synthesized after partitioning, it should be readily extensible to other architectures and synthesis toolsets. Robertson et al. [66] reported an extension to support Xilinx Virtex FPGA series instead of XC6200 family FPGAs.

Advances in the FPGA technology, both in terms of device capacity and architecture, have resulted in introduction of reconfigurable computing machines, where the hardware adapts itself to the running application to gain speedup. To keep up with the ever-growing performance expectations of such systems, designers need new methodologies and tools for developing reconfigurable computing systems (RCS). As the FPGAs get larger and faster, both the number and complexity of the modules to load on them increase, hence better speedups can potentially be achieved by exploiting FPGAs in hardware systems.

Bazargan et al. [67] presented a high-level synthesis approach by compromising the clock frequency of the circuit to achieve speedups in the later placement phase. After the scheduling had been done for all the loops, a hierarchical two-stage placement algorithm was used to determine which loop blocks fit on the RFU and the location of those that fit. The placement method consists of the local placement step that determines the locations of RFUOPs (reconfigurable function unit operations) inside a loop block, and the global placement phase that determines the location of the loop blocks on the RFU.

1.2.7 Interconnect and Power

In traditional design flows, very often the design goes through several iterations of physical synthesis in order to meet the given cycle-time constraints through a tedious manual process of discovering critical paths in the design, fixing them in the RT-level design, and then re-synthesizing. Sivaraman and Aditya [68] avoided such iterations by producing an RT-level design that was expected to meet its cost and performance estimates through subsequent physical synthesis in one pass. The problem was tackled during architecture synthesis through using global information from the control and data flow graph prior to operation scheduling in order to identify what se-

quences of operations should be considered for chaining, and using local timing analysis during scheduling and hardware mapping to validate the candidate operator chains.

As process technology goes into deep submicron range, interconnect delay becomes dominant among overall system delay, occupying most of the system clock cycle time. Interconnect delay is now a crucial factor that needs to be considered even during high-level synthesis. We should no longer assume that interconnect delay between functional units is a part of one clock cycle and interconnect delay should be considered together with computation delay during architectural synthesis in order to achieve timing closure in deep submicrometer technology. Based on the distributed target architecture which separates interconnect delay for data transfer from component delay for computation, Jeon et al. [69] and Kim et al. [32] incorporated the concept of multi-cycle interconnect delay into scheduling and binding process, to reduce the critical path length and to minimize performance overhead due to interconnect delay, therefore the system latency.

For multi-gigahertz designs in nanometer technologies, data transfers on global interconnects take multiple clock cycles. In [70], a regular distributed register (RDR) micro-architecture was proposed for the synthesis of multi-cycle on-chip communication. A RDR architecture structurally consists of a two-dimensional array of islands, each of which contains a cluster of computational logic, local register files and finite state machine controller.

Most previous work tried to minimize the power consumed by the datapath while ignoring the power consumed by interconnects. Interconnects consume a significant fraction of total circuit power. Since wire delay is becoming more significant, wire buffer insertion has become popular. This in turn has increased the portion of circuit power consumed by interconnects. High-level synthesis for low power has attracted significant attentions [46, 37, 26, 47]. It took as its input a behavioral description in the form of a control data flow graph (CDFG) and output a power-optimized RTL circuit.

High-level synthesis can target either bus-based or multiplexer-based interconnections among functional units and registers. It has a significant impact on the switching activity and topology of interconnects in the resultant design. In [71], a technique was proposed to optimize bus power by

appropriately binding data transfers to busses.

Earlier work used floorplanning information in high-level synthesis to estimate the area and performance of the design more accurately. Only some recent works have tried to take interconnect power consumption into consideration. In [27], the switching activity on CDFG edges was profiled. It was used with the floorplanner to optimize the power consumption of inter-module data transfers. Zhong and Jha [72] used neighborhood and communication sensitive to guide the binding process to preserve/create locality in the physical implementation, and evaluated the power consumption in the steering logic and clock distribution network in addition to data transfer wires using early floorplanning information. To estimate interconnect power consumption accurately, wire coupling capacitance was taken into consideration.

Shin and Choi [20] described a power efficient scheduling method that exploits slack times. Dave et al. [21] proposed a low-power co-synthesis method that includes allocation, scheduling and performance estimation. Their methods were at the task level so that estimation was limited to average power. Since system functionality was traditionally described as a task graph with data dependencies, their synthesis did not address any conditional behavior expressed with control dependencies. Doboli [18] addressed control dependencies through performance models which are graphs that not only reflect data and control dependencies present in a system graph, but also capture the relationship between latency and power consumption and design decisions i.e. binding and scheduling.

In this section, we have attempted to cover a wide range of synthesis integration problems and their solutions. We have limited ourselves to high-level physical design issues, and have purposely neglected physical aware design techniques at the logic design level. We analyze design representation, evaluation technique, and integration issues separately, in most practical cases, design quality benefits from focusing on just a few facets of the design optimization problem at one time, while keeping the remaining issues in the background. In this manuscript, we try to integrate physical synthesis into high level synthesis where some physical design challenges are embedded, prioritized and solved in high level synthesis. We hope to obtain a well-balanced synthesis design

by adopting a design flow from the beginning of synthesis hierarchy, and get the acceptable result without synthesis iterations, if possible.

1.3 Overview of the Dissertation

Reconfigurable computing (RC) is going mainstream where FPGA plays an essential role. Synthesizing the application from concept and prototyping onto reconfigurable FPGAs has emerged as one of the main challenges in design automation area. A large number of new applications show the huge potentials of synthesis strategy and architecture development for FPGAs. High Level Synthesis (HLS) translates a behavioral-level specification into its corresponding register transfer level (RTL) structure, which is further synthesized into layout level representation which is physically implementable. This introduces the unpredictability and uncontrollability in HLS in term of final layout implementation. Incorporating physical information into high level synthesis is highly desirable.

The work presented in this dissertation deals with the synthesis and novel architecture of FPGAs. In particular, it tries to address physical aware high level synthesis (PAHLS) methodology to ensure the synthesis integrity for FPGAs. At the same time, motivated by the study of PAHLS, a hybrid interconnect structure is proposed to increase the performance and reconfigurability for FPGAs or FPGA-like reconfigurable platforms. The salient contributions of this dissertation are as follows:

1. Characterization of physical criticality at behavioral synthesis: Incorporating physical information into high level synthesis is highly desirable. Majority of the recent PAHLS methodologies that provide support physical synthesis during high level synthesis go through several iterations of physical synthesis in order to meet the physical restrictions of given constraints. Typically, most physical synthesis approaches (either floorplanning aware synthesis or placement aware synthesis) try to put some physical synthesis stage ahead into high level synthesis, and evaluate those potential solutions through estimation. The physical criticalness is only available after the estimation, and the violations are solved by re-synthesizing the intermediate RTL results or the behavioral

specifications. Further, no much attention is put on the routing. We first present a design technique which attempts to incorporate some critical physical information into high level synthesis. In the proposed approach, both critical operations and possible critical nets are considered when the macros are formulated. Since at the stage of high level synthesis, little or no routing information is available, we extract and weigh the physical criticality statistically.

2. Development of a hybrid interconnect structural model for FPGA: Since mesh interconnect scheme is employed in most commercial FPGAs, and Manhattan scheme is used to perform the interconnect routing. This strategy works well when the number of components or macros are small. In the above work, during the relational placement, the number of components to be placed around the referred component is limited due to its limited neighbors. This limitation would be relieved by introducing tree interconnect into mesh network. We propose a cluster-based hybrid interconnect structure which takes advantages of both mesh and tree interconnect topologies. The proposed architecture is developed with a combinatorial analysis which examines the number of switches needed. Our evaluation demonstrates that the presented model has less switch accrued effects due to the introduction of tree interconnects. By encouraging local routing and short implementations of long connections, significant reduction in the routing area and long path delay can be achieved.

3. Defragmentation by the support of multi-granular configuration: One of the important issues in run-time reconfiguration is the fragmentation of the device area as the reconfigurable blocks are allocated and released when tasks are placed, executed and deleted. Due to those scattered and unused resources, an incoming application may not be placeable or routable because there may have no enough contiguous free area, no enough routing resource, or both. To alleviate this difficulty, we extend the above proposed hybrid interconnect model to support multi-granular configuration. The studies show the efficiency of the extended model in overcoming the fragmentation problem with a penalty of modest increase in the number of switches for the construction of that model.

4. Development of a simulation tool to evaluate run time effects for both placement and routing: In order to evaluate the extended architectural model, we develop a fast evaluation tool

to simulate on-line placement and routing effects on a run-time reconfigurable platform. There are some researches on the on-line placement. In order to simplify the problem, most on-line placement approaches model each task ONE rectangle with fixed height and width. This is not true in practical. A task may be made up of a set of macros. In our approach, we assume each task consists of a set of macros. A task can be configurable if and only if all the macros of that task can be placed at a location where both placement and routing constrains are satisfied. The evaluations of routing effects are performed simultaneously with those of placements.

5. On-line synthesis by ensuring placeability: Another important application of PAHLS is in dynamically and partially reconfigurable computing, where the tasks are synthesized and executed dynamically. The synthesis of those tasks should be aware of the physical resources which are available for allocation, especially, those available resources are determined on-line. On-line synthesis, on-line placement and on-line routing are the three essential steps in implementing an incoming task on the FPGA during run-time. Whereas there has been some research in on-line placement, on-line synthesis received relatively little attention. This proposal presents an automated framework to integrate physical placement information into high-level synthesis that is believed to be the first on-line synthesis methodology for partially reconfigurable FPGAs. In on-line synthesis, time for synthesis should be kept low while ensuring the placeability of the synthesized design in the available empty area on the FPGA and meeting the performance requirements. The proposed synthesizer allocates the FPGA resources adaptively and is incremental in nature. The algorithm is designed to be linear in terms of the number of operations to ensure its on-line usage.

6. Development of a hierarchical representation of specification: To cope with the increasingly larger complexity of electronic system, the design is often captured as intermediate representations that have specific properties, such as modularity and hierarchy. It is commonly agreed that the intermediate design representation is related to the quality of the synthesis results, and the internal representation strongly influences the optimization techniques. The structure of the specification are maintained by transforming it into block format, those intermediate blocks are further abstracted and constructed to a set of precedence graphs, which are DFGs that only contain the

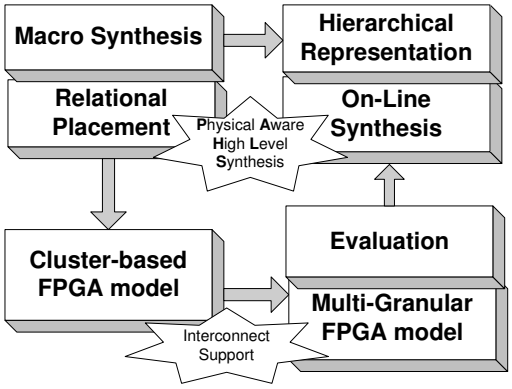


Figure 1.3: Relations of the works

precedence constraints for one iteration. This representation is suitable to be synthesized for sequential and parallel execution, and for the application of fast scheduling algorithm, such as list scheduling. This representation can be also used in an interactive manner and perform system level design space exploration which we will also discuss in the transformation synthesis methodology presented later.

The relations of the current works are shown in Fig. 1.3. The detailed description of each work is presented in the next several chapters sequentially.

1.4 Organization of the Dissertation

In Chapter 2 we present a macro generation and relational placement methodology during high level synthesis for FPGAs that illustrates that considerable improvement would be achieved even when only tiny physical information is considered during higher level formulation. Chapter 3 presents a framework for on-line synthesis from high-level specification to physical level mapping in partially reconfigurable FPGAs. We develop an incremental approach to make physical placement decisions just necessary for each operation being considered during high level synthesis. Then we propose a joint resource allocation process incorporating both logic and memory mappings at the same time. We formulate the problems in an integrated fashion by merging the resource mapping stages with high level synthesis, with the objective to optimize a design through the in-

crease parallelism and effective utilization of available resources. Since interconnect plays a key role in determining the performance of a design, Chapter 5 proposes a hybrid interconnect model for FPGAs. We present our analysis in both combinatoral and statistical ways. Chapter 6 presents an extension of that model to support the multiple granular configurations for the dynamically reconfigurable FPGA. We also develop a simulation tool to evaluate the features of dynamically reconfigurable platform. Finally, we summarize this dissertation and detail some future work.

Chapter 2

Forward-looking Macro Generation and Relational Placement During High Level Synthesis to FPGAs

As design cycle is shortening and design complexity is increasing, system designers respond by defining their designs at a higher level of abstraction, by moving to the more cost-effective block-based designs and by using high level synthesis (HLS) methodology. HLS is a translation optimization process from a behavioral specification into a structural description of the design. Design decisions made at this higher level of abstraction have a pronounced impact on the final outcome. However, the impact of those decisions cannot be found until later in the design process. Studies show that as much as 80% of the total die area on a typical commercial FPGA is devoted to interconnects and more than 50% of delay is due to the interconnects. Therefore, the earlier we can take the interconnect into account the more efficient will be the design at the end. Accurate delay information is not available before placement and routing stages. We present a novel approach to improve the performance of a design synthesized from a given behavioral application. We propose a macro formulation and relational placement methodology, which not only reduces the critical connection delay but also produces an approximate physical floorplan of the chip-level

implementation.

This chapter is organized as follows: Section 2.1 discusses work in related areas of research. In Section 2.3, we present our problem formulation and the overall synthesis flow. Next section describes our strategy for macro formulation and synthesis. Section 2.4 presents the experimental results.

2.1 Related Work

Much work has been done to study the interaction between logic synthesis and layout and interaction between HLS and layout [67, 15, 24, 32, 10, 11, 2]. Su et al. [10] incorporated a soft-macro resynthesis methodology in interaction with chip floorplanning to achieve area and timing improvements. One drawback of such a design methodology is that it is extremely time-consuming. Constructive nature of [15, 32] limits design space exploration. Bazargan et al. [67] addressed the need for fast compilation by compromising in the clock frequency while discarding all bit-width information and only dealing with the type of operations in a data flow graph (DFG). In contrast, we try to weigh the interconnections according to their criticality. This information becomes an input for the macro generation and the relational placement stages, which results in performance improvement. The designer uses this relative placement information to map the design to the target implementation.

2.2 Overall Synthesis Approach and Methodology

Consider the simple example shown in Figure 2.1. The critical path is labeled as darker arrows. The delays of addition and multiplication operations are assumed as 2ns and 4ns respectively. The conventional synthesizer will schedule that DFG into 4 cycles, resulting 16ns latency along critical path. Further, we now assume the short and long interconnect delays as 1 and 3 ns respectively. Without careful consideration, the physical synthesis may get the worst clock period 7ns, resulting

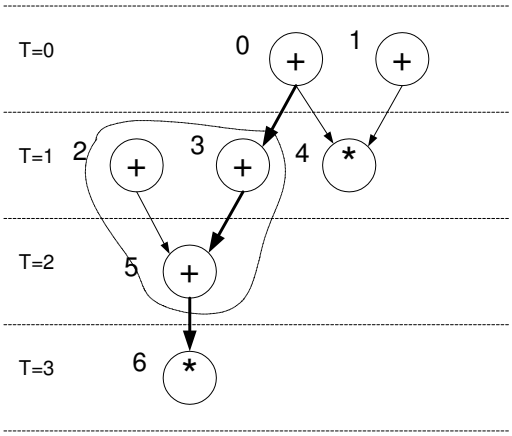


Figure 2.1: A Simple Example

28ns critical path latency. If the node 2,3, 5 are generated as a macro with the limitation to the largest delay of 4ns, the clock cycles will reduce to 3. Under the situation of proper placement, we may achieve the best possible clock period of 5ns and critical path latency of 15ns. In the paper, We present a quick synthesis to improve the performance by focusing on the critical path of given design for FPGAs. We first outline our synthesis approach briefly, then we detail our implementation steps in the following sections.

The overall design flow is shown in Figure 2.2. Synthesis process starts with an initial behavioral specification in high level description language, such as VHDL, C, et al. We schedule operations in CDFG and assign weights to all the interconnections. Then we perform library based resource binding. Critical path analysis is performed next. We define criticality of interconnection nets according to their weight and the interconnection topology. Macro generation and relational placement are done next. If the formulated macro results in performance improvements, it is added to the resource library and the updated nodes are scheduled again. The loop is for resource and timing satisfaction. Finally, the register transfer level (RTL) design is published in VHDL.

2.2.1 Design Representation

Behavioral level representation of the design is transformed into control-data flow graph (CDFG). The CDFG is represented in a behavior block intermediate format, which is organized as a list of

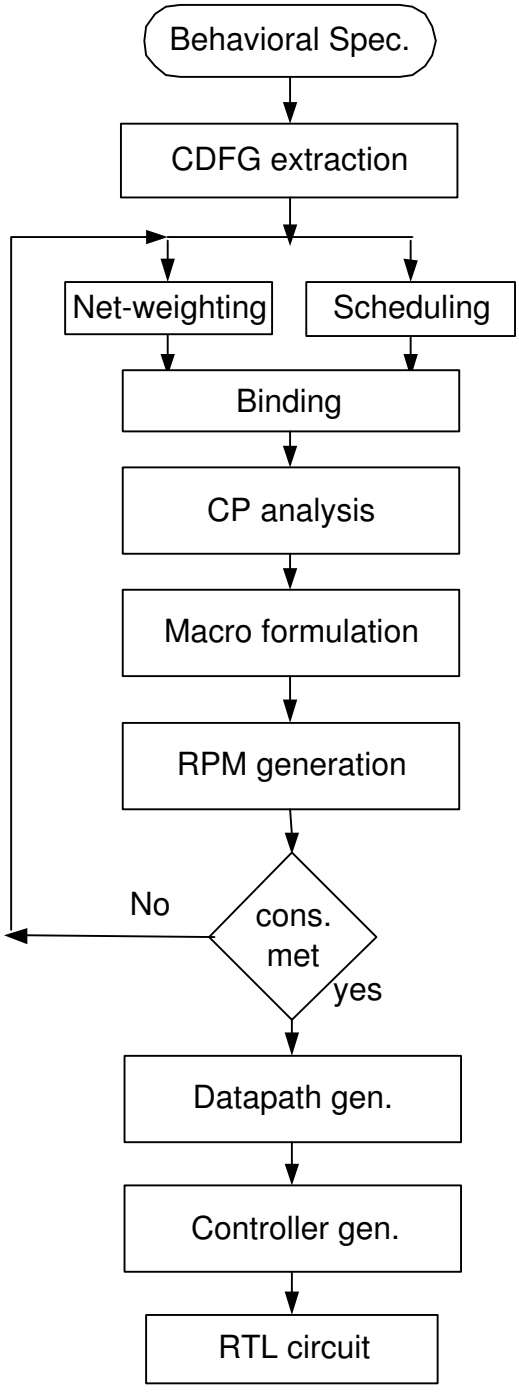


Figure 2.2: Synthesis flow

basic blocks, where the data flow and computations are captured within each behavior block, and the control flow is captured across the blocks. Thus each basic block represents a piece of straight line code in the behavioral specification. The behavioral design representation interacts with the environment through input and output ports that are visible across all behavioral blocks.

Each operation has an input node and an output node. This node represents the data input or output port of a operation node and may be implemented later as a register or wire. Control-dependency edges between behavior blocks ensure correct sequencing. Thus, the total design can be represented through a data and control dependency graph $G(V_{DFG}, E)$, where:

- $V_{DFG} = V_{op} \cup V_{io}$ is the set of vertexes, which consists of operations and their IO ports,
- $E = E_{data} \cup E_{seq}$ is the set of edges, which includes data value $E_{data} \subseteq V_{DFG} \times V_{DFG}$ and control sequence flow $E_{seq} \subseteq V_{io} \times V_{op}$. Note that E_{seq} describe the control dependencies of conditional node executions. It is necessary to consider both data and control flows in the following critical path analysis.

A part of the data and control dependency graph of an ALU design is illustrated in Figure 2.3. The operation nodes are indicated with a initial "V" and IO nodes are labeled through initial "I" or "O". The lighter arrows denote those are control flow edges, while darker arrows represent the data flow edges.

2.2.2 Assigning Weights to the Nets

Determining the criticality of interconnection at high level is a difficult task. In the proposed approach, we overcome this problem by capturing critical path information in the data flow graph using a net-weight model and by later super-imposing placement information onto this model for performance improvement. Since design's actual layout is not known at behavioral level, we use a simple model to weigh the criticality of a net through the available design parameters. We assign the weight of the nets based upon two criteria - the fanout load and the bit-width. We did not

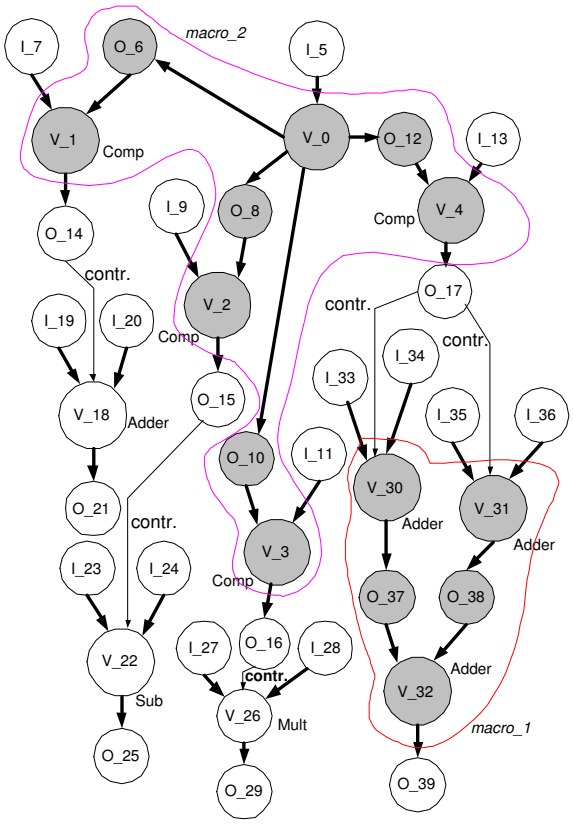


Figure 2.3: Example of data and control dependency graph

consider wire length or delay since these are either unavailable or inaccurate at this time in the design cycle. Net weight is defined by,

$$W_e = B_v \times W_v \times p \;\; where\, e \in E_{data}\, and\, v \in V_{DFG}$$

here e denotes the edge from node u to node v, $u, v \in V_{DFG}$, and B_v is the bit-width of v and W_v is the operation weight of node v. The penalty parameter p is introduced to balance the emphasis between connection and operation. In this paper, since we consider net weight and operation delay separately, p is set to 1.0. These operation weights were determined by performing a statistical analysis of a series of designs in Section 2.4 and are shown in Table 2.2. For each edge, we calculate the net weight for that edge based on the above weight equation. During the above

process, we have also labeled the top weighted nets which may be used at a later stage.

2.2.3 Binding

A component library contains a list of parameterized RTL components. Every operation in the CDFG should be supported by one or more components in the library. We assume that the resource set R needed for implementation of a specification is given. Thus, the number and type of hardware resources that can realize the V_{DFG} are known. Each element L_i in Library L is a 3-tuple (T_i, d_i, a_i). T_i is one of implementations that can perform a typical operation. Corresponding to T_i, d_i is the delay of that implementation, and a_i specifies area information.

The goal of the binding is to find a function resource for each operation node. A consequence of operation binding is that all attribute values (in our case, CPU cycles and area) of nodes become well defined. For example, the node execution time varies with resource type. After binding, execution time becomes known. We define the binding as two mapping steps:

$$V_i \leftarrow L_j \mid V_i \in V_{DFG} \; and \; L_j \in L$$

L_V is the subset of set L which is collection of all $L_j \; : \; L_j \in L$ mapped to a node $V_i : V_i \in V_{DFG}$. We also define function resource binding as

$$R_k \leftarrow L_j \mid R_k \subseteq R \; and \; L_j \in L_V$$

R_k represents the resource to which the node V_i is bound. Obviously, binding has to be done such that $\bigcup R_k \subseteq R$.

2.3 Macro Formulation

One of the performance measures for a digital design is latency. Designers and users need to know how fast a design can process the inputs and produce a usable output. The slowest path from input to output determines how fast the design can execute. That path is called the critical path. In digital synthesis, the overall delay of the critical path can be considered as a sum of all the interconnection and node delays on that path. The delay information for the nodes (whether they are operations or macros) is extracted from a library. Although the analysized path is a true path (that means it represents the true data transfer information and there are one or multiple physical paths corresponding to that path), it is still abstract and detail interconnect is unavailable because the path analysis is performed at behavioral level. We have already weighted the net importance, which would be used to guide our macro generation and relational placement later. We only consider the node delay at this moment. Determining the critical path requires an exhaustive depth-first search. The path with the worst delay is the critical path.

2.3.1 Refining Technique

The path information is refined based on critical operation detection. There are two criteria to choose the critical operations: connectivity and criticality. In the current stage, we use the degree of node to represent the connectivity of that bound operation and the worst delay component in one control step to calculate the criticality of that operation. After binding, the connectivity is modified as following:

$$C_k = \sum e_d \quad \forall e_d \in E_{data} \;\; and \;\; e_d \leftarrow v_i \times v_j$$

where $R_k \leftarrow v_i \; or \; v_j$ and $v_i \; or \; v_j \in V_{DFG}$. For the largest weight net or the node with largest connectivity/criticality in that path, we search for the maximum net-weighted cliques that cover that node. A clustering heuristic is used that attempts to merge nodes in the path based on the edge weights with a threshold on the logic depth as well as the size of the analysis regions corresponding to that cluster. The basic idea comes from the fact that the best timing solution would be achieved if

a macro were formulated absorbing some critical nets. The general strategy applied during macro generation is to move a fanin or fanout of a node or edge x into the macro that contains x. Each of these moves may lead to a smaller number of inter-cluster connections and hence a smaller amount routing outside the macros.

2.3.2 Macro Formulation Algorithm

The CDFG is treated as a forest of trees. Each output node is a root for one of these trees. The input nodes are the leaves. The analysis starts at an output node and travels down the tree along the critical path until it reaches a critical node or the largest weight edge. Depending on which one is first met, two procedures, *Node_ met_ first()* and *Net_ met_ first()*, are called recursively.

The procedure, shown in Figure 2.4, searches the entire critical path, and will eventually reach the input node. Function $update()$ labels all the covered edges and nodes in $G(V, E)$ and removes that nodes from $CP(V)$. There are several limitations of the macro formulation which are verified through $check()$ function. Each macro is checked to see if it contains any constraint violations. This behavior is controlled with a range-window that limits the number of operations to be localized within same neighborhood. The timing cost is used to ensure that critical logic elements are not moved into locations that would drastically increase the worst delay in one control step. This can be controlled by a specified logic depth. Also, it is difficult to formulate a macro across control constructs. The main challenge is to generate node schedules that respect the execution order defined by data and control dependencies. We maintain conditional node execution by not allowing E_{seq} covered during macro generation. Two formulated macros are illustrated as shaded parts in Figure 2.3.

The next step is to do the placement relationally along the critical path. The idea behind relational placement is that the components are placed nearby so that the interconnect delay between them is minimized and data transfers among such components is fast. This layout is not a replacement for a final absolute-coordinate layout. It only provides partial placement information on some

Table 2.1: Delay characteristics of set of operations (ns)

Bit	Adder		Sub		Mult		Mux		Comp		Tbuf	
Width	Logic	Conn.	Logic	Conn.	Logic	Conn.	Logic	Conn.	Logic	Conn.	Logic	Conn.
4	1.644	0.666	1.644	1.025	3.376	1.417	0.573	0.530	1.146	0.322	8.735	0.667
8	1.952	1.190	1.952	1.132	5.152	2.531	1.146	1.188	1.358	1.306	9.140	1.377
16	2.304	2.347	2.304	2.324	7.104	4.597	1.719	1.259	1.534	1.645	9.815	1.766
32	3.008	3.509	3.008	3.655	9.408	6.131	2.155	1.806	1.866	2.281	10.715	2.619
64	4.416	3.865	4.416	3.686	12.989	10.367	2.292	4.160	2.590	4.317	12.155	4.987

emphasized operations and forwards that information to the placement tool.

Normally, each critical net has two vertexes, which are either nodes or macros, hence our relational placement will place a list of vertex pairs. We do it in a straightforward manner. The input includes the ordered of set nets and architecture specification. The first step is to determine a reference function unit. It becomes the bound "seed" of resource R_k. The next is to determine which R_k is to be bound to that seed. It can be viewed as an ordering of vertexes based on their criticality. Actually, this has already been done since the process of macro formulation involves computing operation criticality. Therefore, we choose the vertex with the largest order and place it at the seed location. The second vertex from the first vertex pair is placed in relation to the seed location according to available resources. Once the seed vertex and its paired vertex are placed, the list is searched in descending order for other vertice connected to the seed vertex and these vertice are placed at the appropriate Manhattan distance away from the seed location.

2.4 Experimental Results

The first experiment was setup to validate the net weight equation which was put forward on Section 2.2.2 and acquire the appropriate coefficients for that equation. We used the set of operations listed in Table 2.1, showing each designs name, bit-width, the logic delay and the connection delay. Because we are interested in worst-case delay, we used the average delay on 10 worst connections of each design to determine the connection delay of that operation. We implemented one design at a time in Xilinx XC2V500-6cs456 to retrieve the delay information.

Table 2.2: Operation weight

V_{DFG}	Adder	Sub	Mult	Mux	Comp	Tbuf/IO
W_v	1.0	1.0	2.0	1.0	1.0	2.0

The operation and connection delay are plotted in Figure 2.5 and Figure 2.6 respectively. It is clear that delay increases as the bit-width increases. The logic delays of multiplier and the tri-state buffer and the connection delay of multiplier differ significantly from other operations. This prompts us to assign a larger weight for these operations. We noticed that most operations in our experimental designs are 8 to 32 bits wide, and rounded double weight (Table 2.2) is suitable to reflect the delay ratio in the above two figures. Our attentions should also be turned to the IO operation, which corresponds to the IO node introduced in the CDFG representation. A larger factor will be helpful to evaluate the covered weight during macro development.

Synthesis's experiments have been conducted over a number of synthesis benchmarks [73]. We implemented our design flow using C++ in a UNIX environment. We used the force-directed scheduling algorithm [74] for resource constraint scheduling. Although our synthesis flow is compatible with any architecture, for the experimental purpose, we chose the Virtex-II as our target architecture. The integrated multiplier in Virtex-II makes it easy to determine location of the seed operation which in most cases is the multiplier. The relational placement was implemented artificially through applying Xilinx placement constraints (user constraints file). From the point of view of the synthesis, it can be viewed simply as a component, with associated VHDL attributes defining relative placement information at RTL level. It is therefore necessary to use a synthesizer which is known to pass any required attributes, unmodified, directly through to a netlist. We selected Xilinx ISE 4.2 for the logic and physical synthesis of our published RTL VHDL.

Table 2.3 lists the final synthesis results. To show the effectiveness of the proposed flow, we compared it with the conventional synthesis approach without macro generation and relational placement. For each design, the first row shows the conventional synthesis results which performed high level, logic and physical synthesis consecutively. The second row is the results achieved by the proposed method. The 5th column lists the clock speed improvement, which is calculated from

results of each design in 4th column. Up to 26% improvement is achieved without any area cost (slices utilization in 3rd column, 1% less). We also reported the critical delay improvement in the last column, and average 12.7% is observed. The % gain is measured from two implementation results of each design indicated in 8th column. Since only partial placement is performed relationally on the selected path, we did not expect much improvement of place-and-route time cost listed in 7th column. Our aggressive macro formulation and placement method may cause non-critical connections to become critical due to the lack of global placement consideration. This is the reason for poor improvements in the FFT design.

Table 2.3: Comparisons of experimental results in terms of synthesis flow

Design	Device	Slices Util. (%)	Max. Freq. (Mhz)	Clock Impr.	Logic Syn. time(s)	P&R CPU times(s)	Max Delay (ns)	Perf Impr.
ALU	XC2V40	57	70.393		23	2+3	2.282	
	-6cs144	56	88.747	26%	21	2+3	2.037	10.7%
STATS	XC2V40	99	93.162		33	3+5	2.995	
	-6cs144	99	100.00	7.3%	32	3+5	2.435	18.7%
FFT	XC2V80	62	82.318		40	3+5	3.136	
	-6cs144	60	83.542	1.5%	39	3+5	3.001	4.3%
DCT	XC2V80	89	69.754		43	4+6	3.371	
	-6cs144	88	75.143	7.7%	39	3+6	2.924	13.3%
FIR	XC2V500	20	77.328		41	10+16	3.994	
	-6cs456	19	82.768	7.0%	40	8+15	3.334	16.5%

2.5 Summary

Considering the layout information at the earlier high level synthesis stage has a significant impact on the performance of the final implementation. We presented a forward-looking methodology for performance improvement. Experimental results demonstrate the benefits of incorporating layout considerations into behavioral synthesis.

In current method, we confined our placement on the critical path and limited the number of component to be placed. Although the proposed method has the ability to distinguish possible critical nets based on operation weights, it does not address the traditional wire length metric that could

be further explored at the initial floorplanning stage. One of the main focus of the future research will be to extend the current framework to integrate the macro formulation with incremental floorplanning/placement, leading to a more controllable and predictable design.

Inputs: $G(V, E)$ data and control dependency graph,
$CP(V)$ critical path in reverse order,
$CR(V, E)$ set of critical operations and nets
Outputs: $G(V, E)$: graph with macro node updated,
M : set of macro graph g $\{g \subseteq G\}$

Begin
Var $node\ v \leftarrow CP(V).first(),\ edge\ e \leftarrow \emptyset,\ g \leftarrow \emptyset;$
while $v \neq \emptyset$
if $v \in CR$
Node_met_first(G, CP, M, g, v)
elsif $e \leftarrow max\_net(all\_incident\_edges(v)) \in CR$
Net_met_first(G, CP, M, g, e, v);
end if
$v \leftarrow CP(V).next()$
end while
End
Node_met_first(G, CP, M, g, v)
Begin
if check$(g + \{v\})$ **then** $\{M \leftarrow M + \{g\};$
$g \leftarrow \emptyset;$ $\mathbf{update}(G, CP)$; **break**;$\}$
else $\{\ g \leftarrow g + \{v\};$
for $each\ edge \in all\_incident\_edges(v)\ \{$
$e \leftarrow Next\_max\_net(all\_incident\_edges(v));$
$\mathbf{Net\_met\_first}(G, CP, M, g, e, v);\ \}$
end for $\}$
end if
End
Net_met_first(G, CP, M, g, e, v)
Begin
if check$(g + \{e\})$ **then** $\{M \leftarrow M + \{g\};$
$g \leftarrow \emptyset;$ $\mathbf{update}(G, CP)$; **break**; $\}$
else $\{\ g \leftarrow g + \{e\};$
$v \leftarrow w | w \in e(w, v)$ *or* $e(v, w);$
$\mathbf{Node\_met\_first}(G, CP, M, g, v);\ \}$
end if
End

Figure 2.4: Macro Generation Algorithm

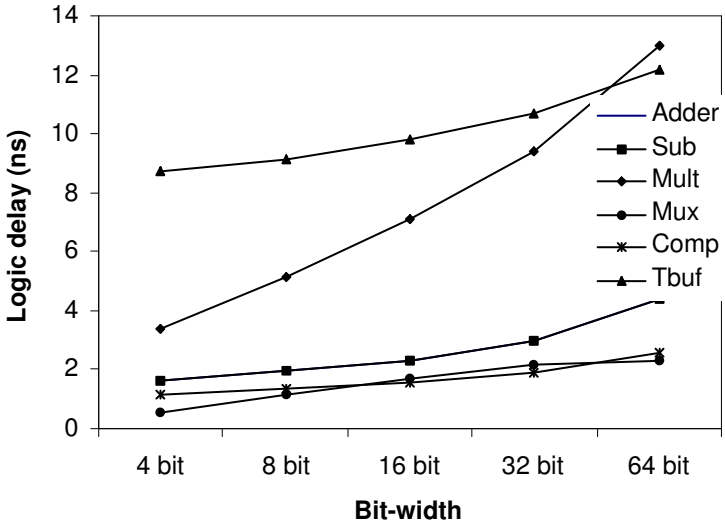


Figure 2.5: Logic delays

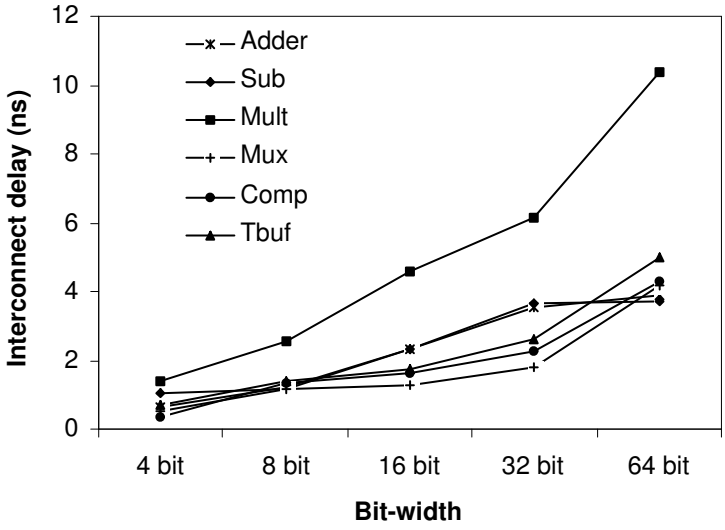


Figure 2.6: Interconnect delays

Chapter 3

On-Line Synthesis for Partially Reconfigurable FPGAs

State-of-the-art FPGAs offer capacities of nearly ten million gates [75]. It is now possible to run multiple independent application tasks on the same FPGA by using partial reconfiguration features during the run-time. Further, field-programmbale gate arrays (FPGAs) are being introduced into new application areas, for example, mobile computing, where dynamic and partial reconfiguration are necessary [76, 77, 78]. However, to fully realize this potential, suitable on-line CAD tools and operating systems are necessary [79, 80, 81]. Use of FPGAs in applications where dynamic task scheduling, allocation and excution are needed requires efficient methods for on-line synthesis, placement and routing. Whereas several researchers have begun investing on-line pacement issues [82, 83], to our knowledge, on-line synthesis has not received much attention. On-line synthesis, where the application tasks are synthesized on-the-fly during system operation, is an open problem.

Traditional FPGA design environments [84, 75], treat the high-level and physical synthesis problems in two separate steps. However, design decisions made during high-level synthesis have a strong impact on the physical design results. The final outcome may invalidate the high-level decisions due to the physical level design decisions. Integration of physical information with high-level synthsis is especially important in on-line FPGA synthesis since a synthesized design must

be placed in the available empty area on the FPGA.

In this chapter, we present, what we believe to be the first on-line high-level synthesis methodology, for partially reconfigurable FPGAs. Our approach has the following key features: (1) It includes physical placement information into high-level synthesis to ensure that synthesized design can be placed in the available area on the FPGA; (2) We use a hierarchical representation of the specification and perform incremental synthesis to ensure efficiency – our algorithm is linear in terms of the number of operations.

The rest of this chapter is organized as follows: The next section briefly discusses research in related areas. Section 3.2 formulates the on-line synthesis problem. In Section 3.3, we present our algorithm for on-line synthesis. In Section 3.4, we present the experimental results. The last section contains some concluding remarks.

3.1 Related Work

Consideration of layout effects in high-level synthesis has received much attention recently. The techniques reported in [22, 11, 15, 10] perform binding and floorplaning/placement either at the same time or iteratively. Their target designs are ASICs and their goals are achieved by incorporating two or more objectives into one optimization process simultaneously. This somewhat expensive in time and, in general, is not scalable to large design. Works reported in [85, 67] address the problem of high-level synthesis with physical information for FPGAs or FPGA-like architecture. Xu et al. [85] take into account the physical information which is characterized from estimation, and Bazargan et al. [67] use simulated annealing-based floorplanning for the local placement. Further, in their approaches the resources are statically allocated and scheduled.

In contrast, we focus on the partially reconfigurable FPGAs. We allocate the resources adaptively with the goal to minimize the effect of high-level synthesis decisions on the layout. Our approach uses a path-based algorithm. Similar scheduling schemes can be found in [86, 87, 88] where they try to achieve optimal or sub-optimal results with exponential or super-linear (greater than linear)

time complexities. Our approach schedules the given application task incrementally using the function resources allocated so far. Our algorithm has linear performance in terms of the number of operations.

3.2 Overview of the On-Line Synthesis Approach

3.2.1 Design Representation

The behavior level specification of the design is transformed into a control-data flow graph (CDFG). The CDFG is represented in a behavior block intermediate format, which is organized as a list of basic blocks, where the data flow within each behavior is captured block and the control flow is captured across the blocks. Thus each basic block represents a piece of straight line code in the behavioral specification. The behavioral design representation interacts with the environment through input and output ports that are visible across all behavioral blocks. Those behavioral blocks are further abstracted and constructed to a set of precedence graphs, which are DFGs that only contain the precedence constraints for one iteration. We use a hierarchical data structure to represent all this information.

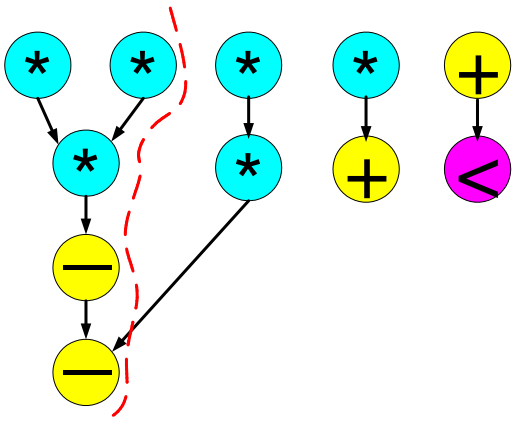


Figure 3.1: Precedence graphs

The behavioral specification is represented as tuple $\mathcal{G} = < \mathcal{P}, \mathcal{E} >$, where $\mathcal{P}$ consists of a set of precedence graphs [87], and $\mathcal{E}$ is the set of dependency edges between precedence graphs

which ensure correct sequencing or paralleling $\mathcal{E} \subseteq \mathcal{P} \times \mathcal{P}$. In this chapter, only parallel execution is concerned, the sequence would be supported by extension of the precedence graph. For each $p : p \in \mathcal{P}$ is a precedence graph, which can be denoted as tuple $p = < V, E >$. Here, $V = \{v_1, v_2, \ldots\}$ is the set of vertices, which corresponds to the operations in the behavioral specifications, and $E = \{e_1, e_2, \ldots\}$ is the set of edges, which indicates only data dependency between those operations: $E \subseteq V \times V$.

The precedence graphs of *diffeq* (the 2nd-order differential equation [89]) are shown in Fig.3.1. It consists of three precedence graphs and it is clear each precedence graph is acyclic. For easy description next, we define some useful notations for our run-time synthesizer in the following subsection.

3.2.2 Definitions and Problem Formulation

The FPGA is modeled as a two dimensional array of configurable units. We assume that the function resources (a set of configurable units which can be configured as RTL units such as adders, subtractors etc.), if allocated, can be only used to execute specific RT-level operations until they are re-allocated again. For consistency, we should associate each operation $v : v \in V$ with an attribute indicating the type of that operation. The available free resources on FPGA are partitioned into empty rectangles, and those rectangles are maintained as the set of all maximal empty rectangles (MER) [90]. Four MERs are shown in Fig.3.2. Work reported in [91] shows that MER could achieve 10% better placement quality than that is possible by using non-overlapping rectangles. $\mathcal{R}$ is used to denote those available MERs. For each $r : r \in \mathcal{R}$, w, h are the width and height of r respectively. Let $\mathcal{T}$ be the set of operation types of $\mathcal{G}$. Further, we use $\mathcal{A}$ and A denote the **currently** allocated function resources and the **newly** allocated resources respectively.

Definition 1 *A **procedural allocation** is a 2-tuple $< \mathcal{T}, \widetilde{A} >$, such that*

$$t \leftarrow \tilde{a} \mid t \in \mathcal{T} \text{ and } \tilde{a} \in \widetilde{A}$$

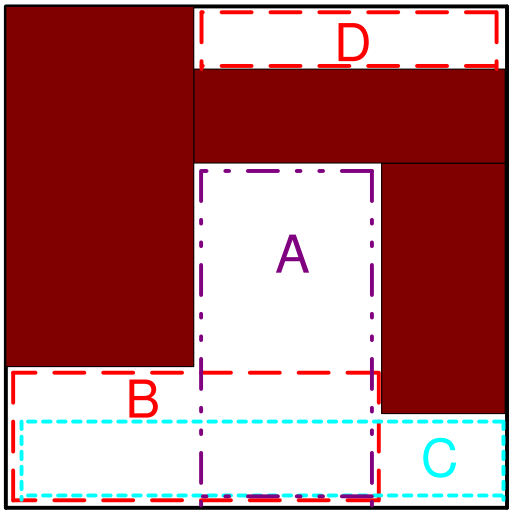


Figure 3.2: Maximal empty rectangles (MER)

satisfying the following:

1. $\widetilde{A} \subseteq (\mathcal{A} \cup A)$
2. $|\widetilde{A}| \leq |\mathcal{R}|$

$\widetilde{A}$ is the resources allocated so far. Each allocated resources $\tilde{a} : \tilde{a} \in \widetilde{A}$ is an instance of operation $v : v \in V$ with the type of $t : t \in \mathcal{T}$. This instantiation is extracted from component library. A component library contains a list of parameterized hard macros. Every operation in $\mathcal{G}$ is supported by one or more macros in the library. Each element L in Library $\mathcal{L}$ is a 3-tuple (L_t, L_d, L_a). L_t is one of implementations that can perform a typical operation. Corresponding to L_t, L_d is the latency of that implementation, and L_a specifies area information, which may be presented as the height and width of that hard macro. One important characteristic of component is that registers are embedded in each macro (Section 3.4). After each procedural allocation, the number and type of hardware resources that can realize $\mathcal{G}$ **at the present moment** are known.

The aim of scheduling is to assign each operation a time frame such that total order induced by such a assignment is consistent with the original execution order of $\mathcal{G}$. A partial schedule is one where scheduling decisions have been made for some of the operations considered so far.

Definition 2 *Let tuple* $\widetilde{S} =< V, \mathcal{F} >$ *be a partial scheduling, where* $\mathcal{F}$ *is the time frame. A* ***soft***

scheduling *is the tuple* $< \widetilde{S}, \widetilde{V}, \widetilde{F} >$*, such that*

$$\tilde{f}_{i1} \leftarrow v_{i1}\ , \tilde{f}_{i2} \leftarrow v_{j2} \mid v_{i1}, v_{i2} \in (V \cup \widetilde{V})\ \textit{and}\ \tilde{f}_{j1}, \tilde{f}_{j2} \in \widetilde{F}$$

which satisfy the following:

1. $V \cap \widetilde{V} = \emptyset$
2. $\mathcal{F} \subseteq \widetilde{F}$
3. $v_{i1} \prec v_{j2}$ *and* $\tilde{f}_{i1} < \tilde{f}_{j2}$

Except the last scheduling, each soft schedule is still a partial schedule. Here the "soft" means that both the allocated resources and the scheduled time frames are flexible. We employ an incremental algorithm to make the decisions necessary at each synthesis step which is discussed in details later. For each scheduled operation, we would allocate it the proper function resources, and also find the locations to place it as explained in the following definition.

Definition 3 *A* ***physical aware binding*** *is a tuple* $< \widetilde{V}, \mathcal{A}, \mathcal{R} >$*, which consists of two mapping steps:*

$$\tilde{v} \leftarrow \hat{a}_i \mid \hat{a}_i \in \widehat{A}\ and\ \tilde{v} \in \widetilde{V}$$

$\widetilde{V}$ *is the collection of all* $V \ : \ V \in \mathcal{P}$*, and* $\widehat{A} : \widehat{A} \subseteq \mathcal{A}$ *represents the resources to which the operation* $\tilde{v} \in \widetilde{V}$ *may be bound*

$$r_k \leftarrow \hat{a}_i \mid \hat{a}_i \in \widehat{A}\ and\ r_k \in R$$

where R *is the list of available rectangles constrained by the layout assessment (See Section 3.3), and we have* $R \subseteq \mathcal{R}$*.*

A consequence of the above mappings is that all attribute values (such as physical dimensions, clock cycles) of nodes are well defined. Therefore, the area and latency of design becomes known. For the on-line synthesis, we should take into account both resource and configuration constraints.

We should check if there is enough space on the FPGA for the currently scheduled nodes. Further, we consider the performance and layout constraints by examining a set of rules for the selecting of $r : r \in \mathcal{R}$. We say that the synthesis result is configurable if and only if it satisfies both resource and configuration constraints. Therefore, we formulate the problem as follows:

Given an application $\mathcal{G}$ which is hierarchically represented as a set of $\mathcal{P}$ and configurable operations $\widetilde{V}$ to be performed on partially reconfigurable FPGA $\mathcal{R}$, allocate the necessary functional resources $\mathcal{A}$, schedule them in time frames $\mathcal{F}$ and assign $\mathcal{A}$ to physical locations while satisfying both precedence and physical constraints.

3.3 On-Line Synthesis Algorithms

3.3.1 Adaptive Allocation

Let N_t be the size of $\mathcal{T}$. The lower bound for the allocation is that at least one resource is allocated for each operation type in $\mathcal{G}$. For example, at least three function resources are needed for the *multiplication*, *add/sub* and *comparison* in Fig.3.1 respectively. Therefore, the first step of our algorithm is to collect the function units and reserve those function resources to implement N_t operations.

We model the available empty space as a set of MERs. We have a set of hard macros from the component library for the instantiation of each corresponding operation. We should choose the component whose dimensions (w_l, h_l) are not greater than those of the candidate empty rectangles (w_r, h_r). Namely, $w_l \leq w_r, h_l \leq h_r$ should be always satisfied. Further, performance is alway the vital issue for the synthesis. We attempt to select fastest implementation under the physical dimension constraints.

After the fewest necessary resources are reserved, we calculate the allocability of each macro (the corresponding reserved resource). The allocability of a macro is defined as the number of resources which could be allocated for that macro. When calculating the allocability, if two or more macros

are competing for one rectangle, the priority is given to the macro with high potential gain of latency. This latency gain g_v for operation $v(t)$ is measured as:

$$g_v = N(t)_{curr} \times L_d$$

where $N(t)_{curr}$ is the number of operations $v(t)$ on the path which is currently considered. We also have the upper bound $N(t)_{max}$ of function resources to be allocated for operation $v(t) : v(t) \in \mathcal{T}$. $N(t)_{max}$ is determined as the minimum between two values, which are the maximal number of instances of that resource at the ASAP and ALAP time frames respectively. Therefore, we limit the resources which are assigned to certain kind of operation $v(t)$. The above gain equation does not only offer a resolution to the competition, but also narrow the allocation space for the currently synthesized operations.

```
1:  Algorithm for Adaptive Allocation
2:  V(t)_p: set of operation types on the path currently considered
3:  Inputs: R, V(t)_p
4:  Outputs: A, the newly allocated resources
5:  Begin
6:  A ← ∅
7:  CalculateAllocability(R, V(t)_p)
8:  for each v(t) : v(t) ∈ V(t)_p do
9:      if allocability(v(t)) ≥ 1 then
10:         v(t) ← r | r ∈ R ;
11:         v(t) ← a | a ≤ r;
12:         A ← A + a;
13:         UpdateAllocability(R − a, V(t)_p − v(t));
14:     end if
15: end for
16: End
```

Figure 3.3: Outline of procedural allocation

A look-up table is built to store the potential gains. The size of that table is relatively small, since only the operations on the selected path are needed to be maintained and their corresponding gains

are to be calculated after each new possible resource is allocated.

3.3.2 Path Based Scheduling

Based on Definition 2, only a subset of operations are considered during each synthesis synthesis. Similarly, only the partial order dependencies among the operations is needed to be maintained at each synthesis step. As mentioned earlier, the design is represented hierarchically. Correspondingly, an incremental synthesis algorithm (show in Fig.3.4) is developed to take advantages of that data structure.

We use path-based scheduling for the implementation of the soft scheduler. After the minimum functional resources are reserved, the initial time frames are scheduled according to the topological order of the longest path (the dotted line shown in Fig.3.1). As a result, we build the best schedule in terms of performance and expect to achieve the best acceptable result after each additional operation is scheduled.

For each precedence graph, there may exist several paths. As a criteria to determine their scheduling orders, we use the number of operations as the selection criteria. For the currently considered path, we also need to select the operation to be scheduled first. We call that operation the *reference operation*. The reference operation is the operation on the scheduled path which is the predecessor or successor of operation node on the current path. The other scenario is that when no predecessor or successor is available, for instance, when the first path of a new precedence graph is to be scheduled. Since this path can be executed in parallel, there is no data dependency among the already scheduled operations and the operations in current path. Therefore, any path-scheduling algorithm would be applicable. We have used the popular scheduling algorithm described in [92] in our implementation.

Obviously, the new schedule depends on the existing partial schedule. At any time frame, if no resource is available for certain operation, the next time frame is considered or the previous schedule is extended to include the operations on the current path.

Each time, after a new decision is made, the order of operations is constrained more tightly. After the new selected path has been scheduled, the operations at any time frame are fixed, and so, the maximum number of the various type resources is fixed. For each kind of resource already allocated, we check for the resource redundancy. If any redundancy is detected for each newly allocated resource, that allocation is rejected and the corresponding functional resources are released for the next allocation. Finally, we update the available resources, and add the new available resources into the $\mathcal{A}$ (shown in Fig.3.4).

```
1:  Scheduling Algorithm
2:  p: the precedence graph to be scheduled, p : p ∈ 𝒫
3:  P̃: the set of paths in p, sorted by criticality
4:  V_pc: set of operations on the path to be scheduled
5:  Inputs: 𝒫, ℱ, 𝒜, A
6:  Outputs: Ã, F̃
7:  Begin
8:  for each path p̃ : p̃ ∈ P̃ do
9:      A ← AllocationProcedure(p̃)
10:     Path_Scheduling(p̃, A + 𝒜, ℱ)
11:     LayoutEvaluation(A + 𝒜)
12:     for each a : a ∈ A do
13:         if No_Redundance(a) then
14:             𝒜 ← 𝒜 + a
15:         end if
16:     end for
17:     Ã ← 𝒜
18: end for
19: procedure Path_Scheduling(p̃, A + 𝒜, ℱ)
20: operation v_r ← ReferenceOperation(V_pc)
21: f_r ← v_r | f_r ∈ ℱ
22: for each operation v : v ∈ (V_pc − v_r) do
23:     timeframe f ← v
24:     if f ∉ ℱ then
25:         ℱ++; F̃ ← ℱ
26:         f̃ ← v | f̃ ∈ F̃
27:     end if
28: end for
29: end procedure
30: End
```

Figure 3.4: Incremental scheduling

3.3.3 Layout Effect Evaluation

The development of an on-line algorithm that finds a feasible and routable layout is not trivial. General purpose combinatorial algorithms such as simulated annealing or genetic algorithm are not suitable for the on-line synthesis because they are quite slow. We model free resources as a list of MERs. At the binding mapping stage, we choose one of MERs as per the placement and routing rule. The layout effects of configuration are evaluated using one of the following strategies:

Best fit (BF) BF chooses the free rectangle with smallest size that can accommodate the macro.

Worst fit (WF) WF chooses the free rectangle with the largest size that can accommodate the macro. This strategy leaves enough empty space around a macro to accommodate other macros.

Neighboring routable (NR) NR restricts to connections among rectangles which are neighbors.

Remote routable (RR) RR may choose the rectangles which are not the neighboring rectangles related to the current macro.

The first two rules are commonly used to search a location for placement. The routing rules are normalized using the following probabilistic model.

$$prob_i = \sum_j \frac{M_R - d(i,j)}{M_R}$$

where M_R is the Manhattan distance between any two rectangles, $d(i,j)$ is the Manhattan distance between two rectangles r_i and r_j , and rectangle r_j is the free rectangle which can cover the current macro when referred to rectangle r_i. Note that the Manhattan distance is the *least* distance between the two rectangles instead of the distance between the two center points (or any pair of corresponding points) of the rectangles. The intuition behind this model is that most FPGAs are employing the mesh interconnect structure, and the Manhattan routing scheme is used for most routing algorithms. By employing these routing rules, the rectangle r_j would be selected for configuration only if $prob_i$ meets the certain threshold value.

To apply the above rules, the first step is to determine a reference function unit. Again, the predecessor and successor are the good candidates. By doing this, we preserve the adjacent consistency between behavioral and physical neighbors. For the regular datapath, such as those in DSP, multimedia applications, this locality consistency has the benefits such as the minimization of interconnect delay.

Our algorithm is incremental. From the above description, whenever a decision is made for the operation, that bound is fixed. Further, each time only one operation is considered. Hence the presented algorithm has has linear time performance in terms of the number of opertions. Since we preserve the datapath structure of DFG, our algorithm results in a smaller problem size. Above all, we skip the logic synthesis and perform module (hard macro) mapping directly, which reduces considerable compile time. Therefore, our synthesizer is quite fast and caters to on-line synthesis.

3.4 Experimental Results

In this section, we discuss the experiments conducted to evaluate the proposed framework. We implemented our design flow using C++ in a UNIX environment. A number of DSP computation kernels are used for the experimental evaluation.

To our knowledge this is the first effort to investigate on-line synthesis for dynamic reconfiguration. In a dynamically reconfigurable system, the application tasks may come at arbitrary intervals, and those tasks need to be synthesized and configured on the reconfigurable platform at run time. There is no benchmark designs or synthesis methodologies available for comparison.

For experimental purposes, the size of reconfigurable platform is set to 128×128, and the largest size of any application is set to 64×64. The life time of each application task and the interval time between any two successive application tasks have been set to uniform distribution in [0, 1000] and [0, 50] respectively. We use Xilinx core generator 4.1i [75] for the hard macros. The option of output register is selected when generating each hard macro, that is each functional unit has an output register. (No separate register allocation is necessary.) The algorithm [91] is used to find

MERs.

We manage the on-line synthesis process as following. If a design cannot synthesized with the space available, it is denoted as rejected (shown in Fig.3.5) and attempted to be synthesized again later. Note that for each application task running on the reconfigurable platform, the exact finish time, as well as the occupied resources are known. Each task will be eventually synthesized and placed with a possible penalty of extra waiting time (shown in Fig.3.6) and increased number of synthesis attempts (shown in Fig.3.7).

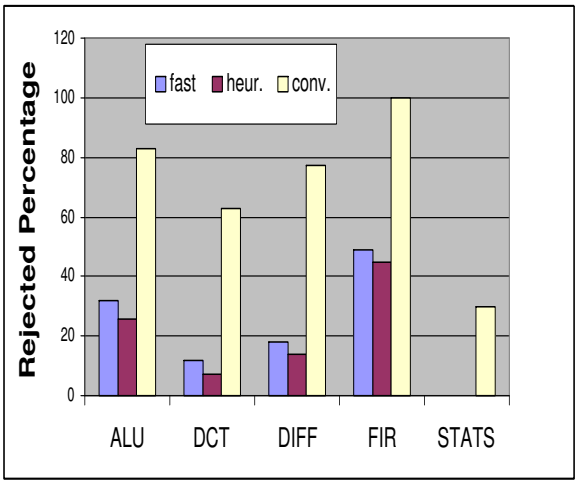


Figure 3.5: Rejection percentage of first synthesis attempting

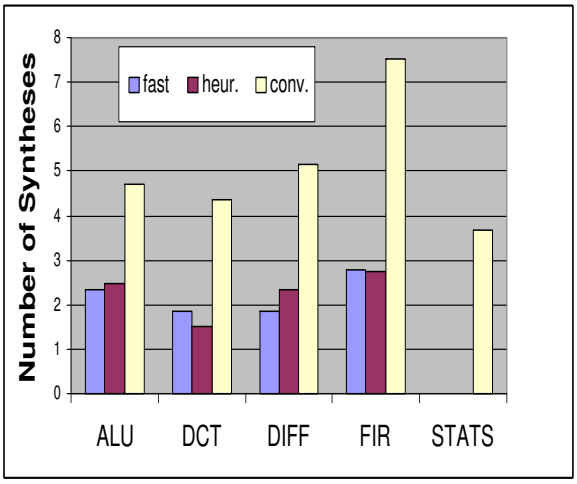


Figure 3.6: Number of syntheses per rejection

Since there is no on-line synthesis flow available for comparison. We set up three synthesis flows.

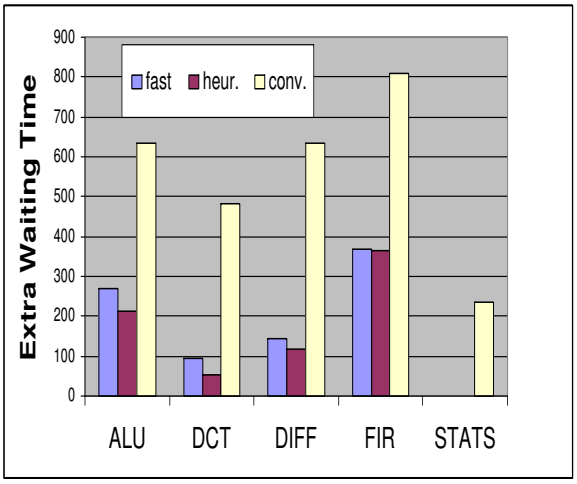


Figure 3.7: Extra waiting time per rejection

The first one is the proposed approach. The second one uses an algorithm [74] for timing constrained scheduling. To make it comparable, the time frame is set as the scheduled result of proposed scheduling algorithm. Therefore, the proposed scheduling algorithm (including adaptive allocation) is performed first at each beginning of synthesis. Further, we assume irregular layout is supported as that in the first flow. Since most on-line placement methodologies currently published support only regular layout, we simulate this by using [93] to compact the final synthesized macros from second synthesis method into one rectangle. This results in a new synthesis flow similar to the conventional synthesis approach, where no physical information is considered during high-level synthesis.

We denoted the above three synthesis flows as “fast”, “heur.” (heuristic) and “conv.” (conventional) respectively. We have conducted significant simulations for evaluations. We generated the reconfigured resources randomly and we ran the synthesis 100 times for each benchmark design to get average values of data.

We give some explanations about the results. **STATS** is a small design which calculates the mean and variance of eight 4-bit integers. It is always configurable at the first try for the first two synthesis flows. On the other end of extreme, due to its larger size, **FIR** is always declined at first attempting for the conventional synthesis flow. For “heur.” and “conv.” approaches, although the resources are not assumed to be allocated adaptively, force directed scheduling may reduce

the needed resources because the time frames are set to the same as that of "fast". From the above figures. We observe that the results of "fast" are very close to those of "heur". As to the conventional synthesis methodology, the phase coupling problem between the HLS and layout stages is obvious, the difficulty of irregular layout leads to long compile times and poor results. Therefore, it is not suitable for on-line synthesis.

As described in Section 3.3, the proposed approach has $O(N)$ time complexity (N is the number of operations). For the "heur." approach, it needs $O(NF)$ to evaluate forces in one time frame, $O(NF^2)$ for all time frames, and needs $O(N^2F^2)$ for all operations. In additional $O(N)$ to determine F. Both synthesis flows need $O(xn)$ for updating MERs [91], where x is the number of columns and n is the number of rows in the FPGA which lie immediately above top boundary of releasing resources. We ran our experiments on a SunBlade 100 with 256MB of main memory. The average CPU time over hundred runs for benchmark *diffeq* is listed in third column of Table 3.1.

Table 3.1: Running time comparisons

Method	time complexity	CPU time (s)
fast	$O(N + xn)$	0.182
heur.	$O(N + xn) + O(N^2F^2)$	0.462

Since both heuristics and irregular layout are supported in second synthesis flow, we may view "heur." as the optimal approach. Although we observe that the results of proposed methodology are slightly worse than those of second synthesis flow for most cases, they are still comparable in terms of optimality. Considering the time criticality, "fast" would be more suitable for on-line synthesis.

3.5 Summary

In this chapter, we presented a framework for on-line synthesis from high-level specification to physical level mapping in partially reconfigurable FPGAs. We developed an incremental approach

to make decisions just necessary for each operation being considered. We not only consider the allocation of functional resources adaptively, but also generate a placeable mapping from operations to physical locations.

We need to point out that there is no guarantee that overall result will be optimal. In the current method, we restricted our mapping to contiguous regions and also limited the number of resources to be mapped. Although the proposed method has the ability to generate a placeable configuration based on a set of hard macros, it does not address the effective utilization of resources that could be further explored with the help of soft macros. We have ignored the routability aspects in this chapter. Rotability analysis in conjuction with on-line synthesis remains a future work.

Chapter 4

Physical Aware FPGA Synthesis with Embedded Memory

According to the SIA (Semiconductor Industry Association), 52% of the area of a typical 2002 SoC design is occupied by embedded memories. This percentage is expected to increase to more than 71% after 2006. Although we see a few of researches have been performed in section 1.2.5, obviously, the increasing trend emphasizes the strategic role of on-chip memory motivates the more efforts in academia and industrial arena to improve the integrating the memory synthesis onto the same silicon substrate from beginning of high level synthesis.

Behavioral synthesis typically takes the design from high-level abstraction to a lower lever specification getting the design closer to a hardware implementation. The process usually consists of mapping operations of the application onto the hardware logic and mapping the data arrays onto physical memory banks of the hardware. Due to its great configurability, Field Programmable Gate Arrays (FPGAs) is getting more and more popular for fast hardware prototyping or substitutions of Application Specific Integrated Circuit (ASIC) implementations. Different from ASIC synthesis, where the mapping process is limited by the richness of the module libraries, FPGAs is intently designed for random logic implementation and there is more than one implementation for each computation. The only limit is the fixed hardware platform. In addition to the configurable

logic, a large number of memories are embedded within the latest FPGAs, and further, those embedded memories can be programmed to the different configurations, such as banks with different depth/width ratios, or even logic functions. Considering those factors, the design space to be explored is also very huge, and it is quite complicated to intelligently mapping the different abstract models to their counterparts in FPGAs.

Little work has been taking those features together into account in the synthesis process. In this chapter, we focus on the data-transfer intensive applications and their synthesis approaches for FPGAs. We propose a new synthesis approach to effectively utilize available resources for both memory and logic mappings by seeking the possible data and computation organization heuristically. The rest of this paper is organized as follows: Section 2 discusses some relevant work. Section 3 describes the problem to be formulated. Section 4 presents the synthesis algorithms. Section 5 shows some results obtained and a discussion of the approach. Finally, Section 6 concludes the paper by presenting ongoing work.

4.1 Related Work

Techniques have been developed to optimize memory accesses or the logic computations separately during high level synthesis, while several researchers tried to consider both problems on the particular architectures. Agarwal et al. [94] developed a technique to partition computations and data to balance load and maximize performance in distributed systems, where the high communication cost exists between processing elements. Huang et al. [55] proposed an enhanced high level synthesis flow that includes techniques to perform automatic data and computation partitioning, and demonstrated that exploiting the performance improvement potential of distributed logic memory architectures. Other researchers have addressed the issues memory operations scheduling in the context of application specific implementations. Schmit et al. [95, 96] used a centralized memory controller scheme where the scheduling of the operations is done in conjunction with the execution of the datapath computation. The techniques reported in [15, 10, 11] studied the inter-

action between high level synthesis and layout mapping either at the same time or iteratively. All target designs above are ASICs, and they decoupled memory operation mapping from the datapath execution. So et al. [50] developed a compilation system that automatically maps high-level algorithms to application-specific designs for FPGAs. Although memory bandwidth utilization is increased, the memory mapping mechanism is not clear. Ouaiss et al. [97] presented an automatic memory mapping methodology which takes the number of design data segments and physical memory banks into account by formulating an integer linear programming model from the available configurations. Their work did not take into account constraints posed by the data structures. Further, the architecture independent view of both datapath and memory mappings in above work is hardly extensible and scalable.

Different from the above references, in the work presented in this paper, the processes of logic allocation and of memory mapping are merged. The resultant spatial partitioning engine considers both logic and data structures at the same time. Both problems are tackled as part of the formulation or of the heuristic algorithm. It can either increase the throughputs by paralleling implementations or seeking the best achievable solutions. Our approach has the following key features: (1) We use a hierarchical representation of the specification, and we keep track of possible allocation by taking both advantages of coarse- and fine-level abstractions. (2) By identifying, exploiting the application specific information, and integrating with physical consideration, we not only reduce the design complexity, but also minimize the overall impacts at the beginning of synthesis in terms of final physical implementation. (3) To ensure the synthesis efficiency, the transformation processes are interacted with high level synthesis by aggressively eliminating unnecessary allocation in design space exploration.

4.2 Problem Formulation

The capacities of commercial FPGAs are far more than multi-million gates, on the same time, more and more on-chip memories are integrated into latest FPGAs. In this work we focus on

the data-transfer intensive designs such as multimedia processing applications. These applications require a very high-speed data access and large storage area. Since FPGAs are intently fabricated for the implementations of random logic, the configurable heterogeneities of FPGAs make the problem of mapping data structures and computations to them increasingly complex. Our problem is formulated as follow: given a fixed amount of resources on FPGA platform, minimizing the number of required resources might not be the best mapping solution; as long as the partitioning and mapping does not exceed the physical resource capacities, it should be allowed to use as many resources as it sees fit.

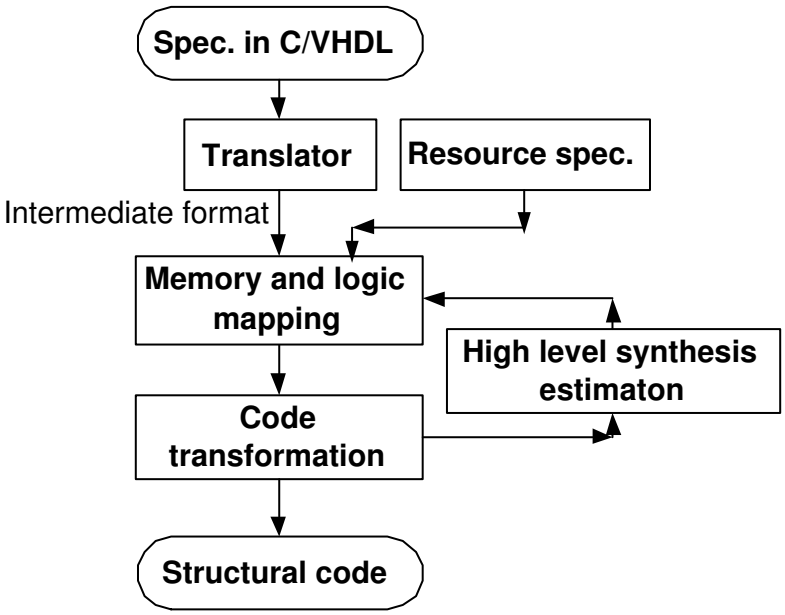


Figure 4.1: Design flow

We show the synthesis flow in Fig. 4.1 . First, the behavioral specification is translated into an intermediate format. This transformation characterizes the design as a hierarchical representation. During the transformation, data dependence analysis is performed, the accesses of logical operations and their corresponding operands are evaluated in light of task or block range. The memory mapping and logic allocation interacts with behavioral synthesis, the iteration evaluations are involved to find the best possible implementation. Finally, the description of structural codes is published.

We first describe our representation technique for the initial input. This intermediate interpret is

important for our synthesis since our transformation engine is based on this representation. The initial input is an algorithmic description that specifies the circuit's functionality without any detail implementation. This specification is translated into a behavior block intermediate format, which is organized as a list of basic blocks, and each basic block represents a piece of straight line code in the behavioral specification. Each block interacts with other behavioral blocks through input and output ports which would be projected into registers or grouped as memory banks later in our synthesis.

The first level of hierarchical representation is task level description shown in Fig. 4.2(a). A set of behavioral blocks are grouped to describe the flow of the data stream, for example, a computation of single or multi-dimensional loop is formulated as a task. The partitions of those tasks are based on the computation precedence. Therefore, the design can be described as the tuple $\mathcal{G} = \ < \mathcal{T}, \mathcal{E} >$, where $\mathcal{T}$ consists of a set of task graphs, and $\mathcal{E}$ is the set of dependency edges between task graphs which ensure correct sequencing or paralleling $\mathcal{E} \subseteq \mathcal{T} \times \mathcal{T}$.

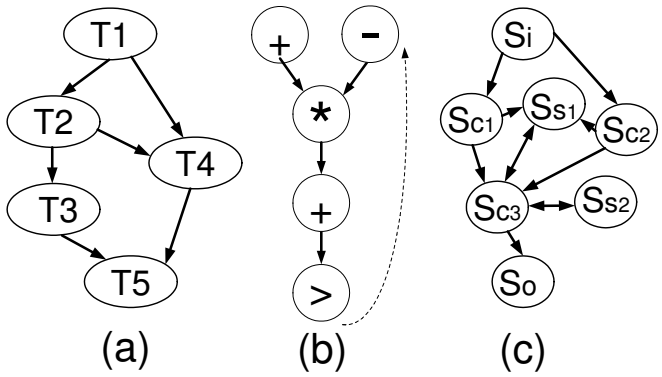


Figure 4.2: Two level representation

The behavioral blocks in each task are further abstracted and constructed to a set of precedence graphs, which are data flow graphs that contain the data dependencies for one iteration. For each $T : \ T \in \mathcal{T}$ is a precedence graph, which can be denoted as tuple $T = \ < V, E, S >$. Here, $V = \{v_1, v_2, \ldots\}$ is the set of vertices, which corresponds to the operations in the behavioral specifications. $E = \{e_1, e_2, \ldots\}$ is the set of edges, which indicates the data dependency between those operations: $E \subseteq V \times V$. One of task graphs of [98] is shown in Fig.4.2(b). Note that in that figure,

the implicit loop is illustrated as a dotted line. Different from V, E, $S = \{s_1, s_2, \ldots\}$, which consists of set of states (we will discuss in next section) describing the execution stages of a task in a more coarser way.

Although the input has been hierarchically represented in both coarse and detail formats, it has not been specified with any potential implementation solution. Next, with the help of behavioral synthesis, more detail information will be incorporated into that representation and the whole design would be elaborated toward hardware prototype in terms of final implementation.

4.3 Synthesis Methodology

We estimate the needed hardware resources by performing high level synthesis. Traditionally, high-level synthesis consists of three procedures: firstly, allocation chooses the type and number of functional resources; secondly, scheduling assigns time steps to each arithmetic or logical operation; finally, binding assigns the functional units to the corresponding operations. Each procedure affects or refines the final cost or performance by extracting the information from hardware models in the component library. A component library $\mathcal{L}$ containing a list of parameterized RTL components is assumed to be available. Every operation in the behavioral specification should be supported by one or more components in the library. Each element $L : L \in \mathcal{L}$ is a 3-tuple $t = < t_l, d_l, a_l >$. t_l is one of implementations that can perform a typical operation. Corresponding to t_l, d_l is the delay of that implementation, and a_l specifies area information.

The resources of FPGA are modeled as a tuple $\mathcal{F} = < \mathcal{R}, \mathcal{M} >$, where $R : R \in \mathcal{R}$ is the logic resources (a set of configurable units which can be configured as RTL units such as adders, as well as some internal functional units such as embedded multiplier.), and $\mathcal{M}$ is the set of on-chip memory banks. For each $m : m \in \mathcal{M}$, m_t, m_c are the memory type and capacity of m respectively.

4.3.1 Initial formulation

The data structures of each task $T : T \in \mathcal{T}$ are extracted at the beginning of synthesis. Let $\mathcal{D}$ be the set of data structures of T. For each $D : D \in \mathcal{D}$, w_d, h_d are the width (number of bits per word) and height (number of words) for the instance d of D. We assume each data structure has it corresponding memory bank. Note here we have not considered whether or not there are enough physical memory banks available in FPGA for the one-to-one mapping of those memory banks. When the feedbacks from behavior synthesis are available, we will identify the data access patterns within tasks and assign necessary memory banks to allow the maximum parallelism for the task execution.

Let $\mathcal{O}$ be the set of operation types of $\mathcal{G}$. Further, we use $\mathcal{A}$ denote the currently allocated function resources for $\mathcal{O}$ from $\mathcal{R}$.

$$O \leftarrow A \mid O \in \mathcal{O} \text{ and } A \in \mathcal{A} \text{ and } (\cup A) \subseteq \mathcal{R}$$

Each allocated resources $A : A \in \mathcal{A}$ is an instance of operation with the type of $O : O \in \mathcal{O}$. This instantiation is extracted from component library.

The aim of scheduling is to assign each operation a control step such that total order induced by such a assignment is consistent with the original execution order. Up to now, $T : T \in \mathcal{T}$ are still unscheduled data flow graphs, we use force-directed scheduling algorithm [74] to obtain the first schedule. Since force-directed scheduling achieves the minimum hardware resources under a certain latency constraint, it limits the concurrent operation executions. Those limitations are relaxed when more resources are available in the later improvement stages. After scheduled, we estimate the needed hardware resources by performing operation binding.

When a device is selected, the resource set tuple $\mathcal{F} = < \mathcal{R}, \mathcal{M} >$ needed for implementation of the application specification is given. Thus, the number and type of hardware resources that can realize $\mathcal{G}$ are known. The goal of the binding is to find a function resource for each operation node. A consequence of operation binding is that all attribute values (in our case, clock cycles and

the number of configurable units) of nodes become well defined. For example, the node execution time varies with resource type, after binding, execution time becomes known.

Finally, the information obtained so far is back-annotated to the state $S : S \in \mathcal{S}$. As shown in Fig. 4.2(c), those states specify the input S_i, output S_o, several computation stages S_c and intermediate states S_s for the storage requirements. Different from the traditional state transition graph, where each state corresponds to one node in the application graph, states $S : S \in \mathcal{S}$ are constructed on the data stability which several nodes may be grouped as a computation state $S : S \leftarrow (\cup v)$, where $v : v \in V$. Further, the intermediate storage states are built for those data which are scheduled for not immediately use or for the outer loop fetch in the multiple loop computation. Since we consider memory and logic resource mappings simultaneously, this kind of state specification is vital for the synthesis transformations. Later, we will see it also simplifies the parallel exploitation by replicating the corresponding states.

4.3.2 Performance driven allocation algorithm

One of the performance measures for a digital design is latency. Designers and users need to know how fast a design can process the inputs and produce a usable output. After back-annotation from high level synthesis, the overall area and delay can be approximated. We use the total clock cycles as the criterion of performance. We start with the memory configuration by seeking the most critical task which needs the longest clock cycles. We try to allocate more resources for it. The idea behind our algorithm is similar to the loop unrolling in the software while we focus on the hardware resource allocation. More memory banks and logic resources are employed to improve the performance. The algorithm is outlined in Fig. 4.3.

The first procedure is to find the initial memory assignment. From the feedback of behavioral synthesis, the life times of data structures have been analyzed when performing the scheduling execution. This life cycle analysis could further improve the memory mapping since segments that can overlap could be placed in the same storage area, namely, memory bank, thus decreasing the

```
1:  Procedure MemoryEval(M, T)
2:  T_c: critical task initially considered
3:  D_c, M_c: data structures and memory banks for T_c
4:  A_n, M_n: logic resources and memory banks newly allo-
    cated
5:  A, M: the resources and memory banks already allocated
6:  Begin
7:  M ← M_c
8:  for each T_i : T_i ∈ (T − T_c) do
9:     M_n ← ∅
10:    D_i ← T_i
11:    for each d_i : d_i ∈ D_i do
12:       if match(m, d_i) then
13:          m ← d_i | m ∈ M and d_i ∈ D_i
14:       else
15:          m_i ← d_i | m_i ∈ (M − M)
16:          M_n ← M_n + m_i;
17:       end if
18:    end for
19:    M ← M + M_n;
20: end for
21: End
22:
23: Procedure ParallExp(M, A, T)
24: T_n: task currently considered
25: D_n, S_n: data structures and state graph for T_n
26: Begin
27: if Available(R, A) then
28:    M_n ← ∅
29:    r ← v | r ∈ (R − A) and v ∈ S_n
30:    A_n ← ∪r
31:    while Available(M, M) do
32:       if AccessSatisfied(M + M_n, T_n) then
33:          A ← A + A_n
34:          M ← M + M_n
35:       else
36:          M_n ← m | m ∈ (M − M)
37:       end if
38:    end while
39: end if
40: End
```

Figure 4.3: Algorithm of joint allocations

total storage requirement. For this purpose, the mapper needs to know which data segments life cycles overlap. We begin the process from the critical task, and incrementally allocate the memory to all data structures corresponding to their task. For two data structures, if there is no precedence conflicting between their corresponding tasks, the data segments could be ideally mapped to the same physical bank. On the other end of the spectrum, if all conflicting pairs exist, the additional memory banks A_n should be allocated for the mapping of data structures which are currently considered. Let T_{i1}, T_{i2} be two tasks, where $T_i : T_i \in \mathcal{T}$. The data segments d_{i1}, d_{i2} can be assigned to the same memory bank m_i if we have

$$D_{i1} \leftarrow T_{i1},\ D_{i2} \leftarrow T_{i2},\ m_{i1} \leftarrow d_{i1},\ m_{i2} \leftarrow d_{i2}$$

where $m_{i1},\ m_{i2} \in \mathcal{M}$, $m_i \in M$ and $d_{i1} \in D_{i1},\ d_{i2} \in D_{i2}$, such that

$$m_i \leftarrow m_{i1},\ m_i \leftarrow m_{i2}$$

which satisfy the following:

1. $|m_{i1}| \subseteq |m_{i2}|$
2. $T_{i1} \prec T_{i2}$ and $d_{i1} < d_{i2}$

From the results obtained so far, we determine the enough logic resources and necessary memory banks for the implementation of design. Obviously, in most cases there may still exist memory banks and logic resources available for allocation. The second procedure is used to exploit the possible allocation by aggressively seeking paralleling execution and large chunk of data caching. For each newly allocation A_n, which is determined as follows. For the considered task, we recursively select the operation instantiation with the largest performance gain. The final A_n is the summation of all instantiations accepted.

$$A_n \leftarrow (\cup a_l) \mid a_l \leftarrow v, \text{ and } v : v \in S_c$$

After more resources are reserved, a rescheduling procedure is called again. Since only part of operations is affected, we use a simple list scheduling heuristic to accelerate this process and obtain new partial schedule for those affected operations.

The function *AccessSatified()* checks whether there are enough access patterns for the incurred executions. A footprint analysis of the memory accesses could tremendously help in guiding the mapping process: e.g. data segments that are extensively accessed should be assigned to faster and closer physical banks. We focus on the size of each data segment and the possible frequency of accesses to that segment. For each data structure in the design, the size of data segment correspond-

ing to that data structure depends on capacity of mapped memory banks and the number of words (depth). We first check the satisfaction of concurrent access for the currently execution units. We assume that all ports of the physical banks are assumed to be read/write ports. If a physical bank has multi ports, it will be served as multiple memory bank. Any data structure could be mapped to any port of a physical bank and each port can have its own configuration setting. Among the memory banks that lead to feasible memory configurations, we choose the one with the smallest memory banks. As mentioned earlier, we have already back-annotated the data access patterns S. After mapping, the data structures of different tasks may share the same physical banks, and some memory banks also support the concurrent accesses for the computation within one task. In order to satisfy the give access patterns, the memory banks should be increased until there are enough bandwidth. On the other hand, since a major advantage of on-chip memory is its short access time, the external memory accesses should be largely reduced as long as there exists the potential gain under that memory banks already allocated and there is no conflict between the memory accesses.

During any check point in the procedures, the following Capacity constraints should be met.

$$|\mathcal{M}| \geq M \text{ or } |\mathcal{M}| \geq (M + M_n) \text{ and } |\mathcal{M}| \leftarrow \cup m$$

$$|\mathcal{R}| \geq A \text{ or } |\mathcal{R}| \geq (A + A_n) \text{ and } |\mathcal{R}| \leftarrow \cup R$$

where $m : m \in \mathcal{M}$ and $R : R \in \mathcal{R}$ respectively.

It is necessary to point out that, in this paper, we focus on the date transfer and intentionally ignore the register optimization. Our targeted platform is the configurable architectures where the number of registers is not necessarily optimized. One important characteristic of component is that registers are embedded in each operation instantiation. This helps improving the accuracy of estimated execution cycles when they are back-annotated to the S.

4.3.3 Refining Technique

The basic computing units in T or S are the fundamental mathematical and logic operations. Some of those units can be grouped together to formulate a more complicated operation, which leads to both area and performance improvements. If we know the DFGs of those complicated computation models, we can search those model DFGs in the design DFGs. This is essentially a graph isomorphism problem. We use the exact network algorithm [99] to perform this kind of matching. Since our DFGs of tasks can be treated as a forest of trees, each output node is a root for one of these trees in the forest. The search starts at an output node and travels up the tree to find possible replacements. One of these matches is illustrated in Fig. 4.4(a), where a multiply-accumulator (MAC) is found in the bound box.

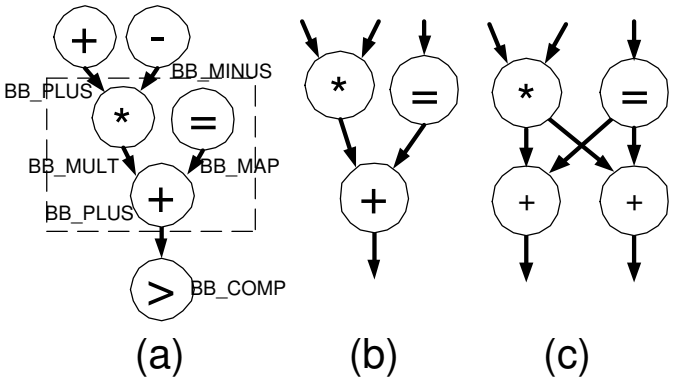


Figure 4.4: Simplized illustration of refinements

Another refining process occurs when budgeted resources are tight for allocation. For instance, for some certain operations, they need a lot of resources for their implementation and this may lead to the final synthesis impracticable for the selected device. Two possible solutions are offered. One is to comprise the performance by selecting the instantiation with less resource requirement from the component library. The other is to reuse the components by splitting the computation of that operation into two or more sub-computations which can be taken by the already available components. One of computation decomposition is illustrated in Fig. 4.4 (b) and Fig. 4.4(c).

4.4 Experiments

Experiments have been conducted over a number of synthesis benchmarks. We implement our design flow using C++ in a UNIX environment. First, we apply our algorithm to several DSP benchmark designs and generate register-transfer level (RTL) implementations. For experimental purpose, we assume that each design has a frame size of input data 320×240, which is encoded to 8 bit unsigned vectors. We select the Xilinx Virtex FPGAs as target devices. Xilinx Core Generator is used to generate the components in the component library. We compare our our synthesis approach with the conventional one, where Asserta [100] is used as high level synthesis tool while the memory banks are scalable to contain more data storage space.

Table 4.1: Comparisons of experimental results of DSP tasks

Design	Device	Slices Util. (%)	Impr. Ratio	RAM Util. (%)	Impr. Ratio	Total Cycles (M)	Impr. (%)
FFT	XC2V80	62		80		0.323	
	-6cs144	91	1.47	100	1.25	0.287	11.1
DCT	XC2V80	89		100		0.226	
	-6cs144	95	1.07	100	1.0	0.207	8.4
FIR	XC2V500	20		40		0.586	
	-6cs456	79	3.95	100	2.5	0.252	56.9

Table 4.1 lists the results obtained from synthesis of generated RTL codes with the help of existing Xilinx ISE synthesis tool. To show the effectiveness of the proposed flow, for each design the first row shows the conventional synthesis results and the second row is the results achieved by the proposed methodology. The 3rd, 5th columns list the final slice and block memory utilization respectively. The improvement ratios listed in 4th, 6th column are calculated from results of each design in 3rd, 5th column. We report the possible performance gain of each benchmark study in the last column which is measured from two implementation results for the whole execution of one frame data.

We give some explanations about the results. For the first two designs, although there are not much resources left after the conventional synthesis, the proposed approach can still achieve noticeable improvements. Due to its larger size, FIR can not be implemented on the same device as previous

one for both synthesis methodologies. We observe that the result is more promising when much more resources are available to be effective utilized. This property is further validated by the next experiment.

We then conduct experiments on the EigenFace algorithm (EFA) presented in [98] to evaluate the effects of real application. This algorithm is a typical data and computation intensive application for face recognition. The algorithm requires huge matrix operations which involve the calculations of the Face Vectors. For the input face image with mediate solution, the logic computation and memory requirements are pretty high. For consistency, We still assume that input face image is 320×240 of 8 bit encoded vectors. A careful examination will find that the intermediate eigenfaces should be encoded using 16-bit signed vectors, Face Vectors are 32-bit signed vectors, and large operations, such as 72-bit accumulator, are involved in the different computation stages. Two targeted devices (XCV300 and a much larger one, XCV800, we denote those two implementations as EFA1 and EFA2 respectively) are selected, and the results are reported in Table 4.2.

Table 4.2: Comparisons of Eigenface implementations

Device	Slices Util. (%)	Util. Ratio	RAM Util. (%)	Util. Ratio	Total Cycles (M)	Impr. (%)
XCV300	81		100		16.1	
-6pq240	88	1.08	100	1.0	11.9	26.1
XCV800	21		17		10.3	
-6hq240	88	4.19	70	40	2.90	71.8

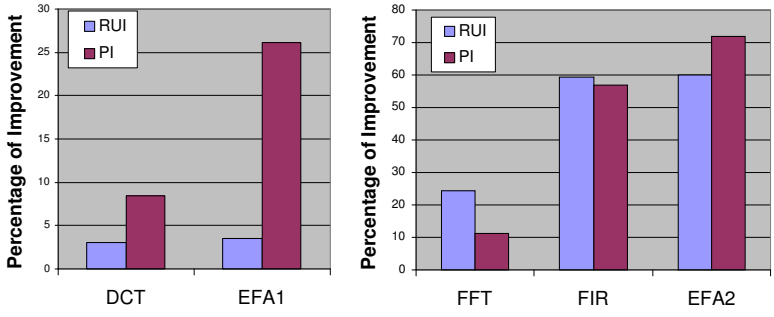


Figure 4.5: Resource and performance improvements

We illustrate the final improvements in Figure 4.5. We use RUI and PI to represent the resource

utilization improvement (or overhead) and final performance improvement respectively, where RUI is calculated as the average percentage of both increased memory and logic resources. It can be seen that, even for resource tight-budgeted devices, the performance improvements are still considerable. From Fig. 4.5(b), both RUI and PI are significant when there are much resource available. This is especially useful for today 's FPGA synthesis. Since both FPGAs and designs are becoming larger and larger, for a given device with fixed resource, it is necessary to increase the overall throughput by utilizing the hardware more efficiently. Also, from the results shown in Table 4.2, it is clear that the proposed approach is more attractive for the large design. In terms of performance improvement, our synthesis approach is able to obtain the gain of 26.1% for tight-budgeted device and achieve the significant gain (71.8%) for larger device by comparing with the results of conventional synthesis approach.

4.5 Conclusions

In this chapter, we present a transformation synthesis approach for FPGAs to solve two important problems: (1) a tight link from behavioral translation and system synthesis for FPGAs. (2) a joint resource allocation process of both logic and memory mappings at the same time. Both problems are tackled as part of the formulation or of the heuristic algorithm. We solved the problems in an integrated fashion by merging the resource mapping stages with high level synthesis. Our results reveal that, by taking into account different logic and memory resource assignments at the design space exploration phase, it is possible to optimize a design through the increase parallelism and effective utilization of available resources, and select the best design among a set of candidates. It is seen that presented technique could be used to prune the design space with the help of incorporating more physical information into behavioral specification. Such methodology, integrated with pre-placed or routed macros, restrict computations and communications to geographic proximity while reserve the quality of the final mapping to a large extent by limiting the search space. We believe that the methodology presented in this chapter could be easily extended to unify system synthesis and layout implementation, resulting in the efficient run-time mapping of behavioral applications

to reconfigurable hardware for the custom computation.

Chapter 5

Analysis and Evaluation of a Hybrid Interconnect Structure for FPGAs

Field-programmable gate array (FPGA)-based implementations map designs onto an array of configurable logic blocks that are connected by programmable interconnects. Compared to the custom designs, this design style requires fewer design iterations and has the advantage of shorter time-to-market. However, the system performance and logic density in FPGA-based implementation are not as high as those of custom designs due to the area and performance overhead of programmable logic and interconnect.

The programmable interconnect is a key design element in FPGAs. In some recent studies, the programmable routing structure occupies about $70 - 90\%$ of the overall chips silicon real estate, and $60-80\%$ of overall design delay are attributed to programmable interconnects [101, 102, 103]. The current trend in the FPGA industry is to pack more and more transistors on a chip (till ten million gates in state-of-art FPGAs [75]). This trend puts an ever-increasing burden on the routing structure.

Two-dimensional (2-D) array FPGAs have good routing wiring utilization and are easy to be synthesized for computer aided design (CAD) tools, but its Manhattan connection scheme makes the delays of long wires grow linearly. In order to improve the performance of FPGAs, commercial

designs provide fixed long interconnections. Works in [104, 105, 106, 107] describe hierarchical interconnects based on tree structures show that both density and performance can be further improved. However, their routing flexibility, are largely depended on the designs of connection topologies between any two consecutive levels in the tree network.

This chapter presents a cluster-based FPGA with a hybrid interconnect architecture, which takes both advantages of mesh and tree interconnects. In the proposed architecture, several CLBs are formed as a cluster, and tree networks are imposed over the mesh arrays to stitch parts of clusters together. The routing is performed through a hierarchical fashion: mesh and tree routings.

The organization of this chapter is as follows. We present our architecture model in the next section. In Section 5.2, we analyze and estimate the area and performance of proposed model in terms of switches needed. The experimental results based on the MCNC benchmarks are provided in Section 5.3. Finally, Section 5.4 summarizes this work.

5.1 Architecture Modeling

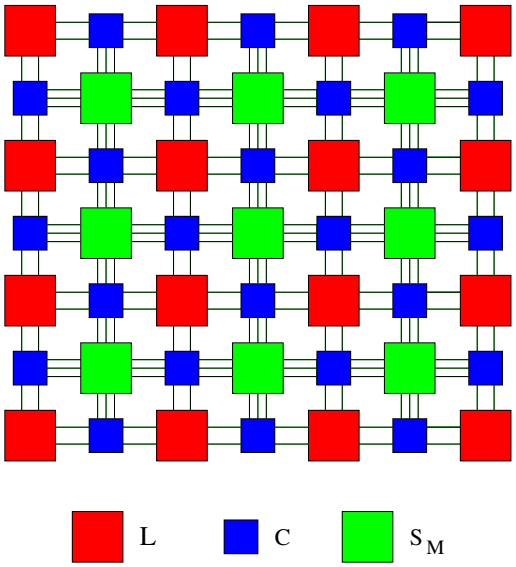


Figure 5.1: Array FPGA

A typical two-dimension island-style FPGA architecture [108, 101] is illustrated in Fig.5.1. It con-

sists of a 2-D array of configurable logic blocks (CLBs, labeled as *"L"*) and horizontal and vertical routing channels. The input and output pins of each CLB are connected to the connection blocks (labeled as *"C"*) in the adjacent channels. The connections between different connection blocks are performed through switch blocks (labeled as *"S_M"*) located in the each channel intersection.

Channel width, symbolized as *"W"*, is defined as the number of wires or tracks in a channel. Those tracks are the fixed connections between connection blocks and switch blocks. $W = 3$ in Fig.5.1. In order to make a connection flexible, there are a number of switches for each track to route that track to CLB through connection block or to the tracks of other sides of the corresponding switch block. For example, in the Xilinx 's XC4000 switch block, each programmable point has 6 switches. It is reasonable to consider that SW_{clb}, the total number of switches required for one CLB, is at least related to W as: $SW_{clb} = O(W)$ Note that the above is the lower bound of SW_{clb}. It would be approaching $SW_{clb} = O(W^2)$ when the routing flexibility of each track is increasing. If the array size is $N \times N$, the total switches of that FPGA are $SW_{total} = O(N^2W^2)$. Since N is fixed, W is a key role to measure the number of switches. Normally, *W* should be determined as the maximum number of needed tracks by routing a set of benchmark circuits completely within a given performance.

The inherent diameter of mesh structure is N, this means that the path will pass through $2N - 1$ switches in the worst case. The performance of an FPGA can be increased by reducing the number of passing switches along signal path. Tree structure [105, 109] decreases the diameter from N to $\log N$. Although this kind of hierarchical structure decreases the passing switches from $2N - 1$ to $2 \log N - 1$ in the longest path, great attentions are needed to put on the switch block design and a larger number of trees are needed to increase the routing flexibility. This can be illustrated by comparing the bisection width of their interconnect networks. The bisection width [1] of a network is the minimum number of wires that have to be removed in order to disconnect the network into two topologically identical halves. Consider the simplest case: the mesh network ($W = 1$ in Fig.5.1) and a binary tree network. The bisection width of mesh structure is N, while it is only 1 in the binary tree.

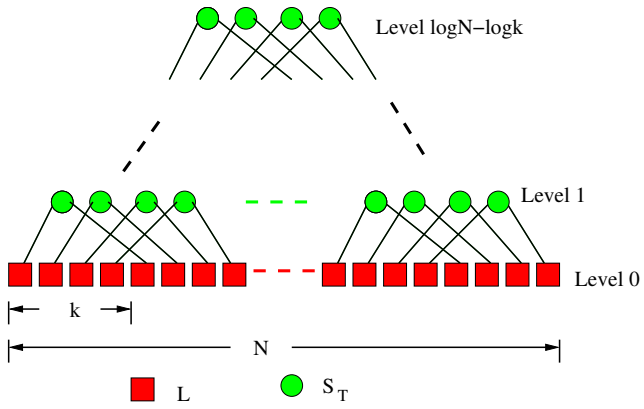


Figure 5.2: One dimensional connection hierarchy

The 2-D mesh interconnect structure works well for the shorter connections and is easy to be synthesized by the CAD tools. For the long wires, the tree interconnect is more attractive. How does a combination of mesh and tree work? This motivates us that a hybrid interconnect structure may take both advantages of mesh and tree-like interconnect networks. Since most routings in FPGAs are short connections, we may control the connections between consecutive nodes of trees by grouping several leaves together as a cluster and performs all short connections within in that cluster. That is to limit the local routings within clusters and complete part or all external routings between clusters through trees. Hence a hierarchical interconnect is constructed in Fig.5.2. For the clarity, we only show the structure in one dimension. The other dimension has similar structure due to the symmetric.

Let $k \times k$ denote the size of cluster, there are k leaves (from henceforth, we use the terms leaf and CLB interchangeably) for one dimension of cluster. Each leaf in a cluster has two trees connected to the corresponding leaves in the other clusters. Namely, in each dimension of an array FPGA, every two corresponding CLBs located on two clusters with distance k are formulated as two leaves of a binary tree. The same policy is applied to the root nodes from level 1 to level $\log N - \log k$ recursively. Therefore, instead of one, $k \times k$ mesh-of-trees (MoTs) are constructed and organized on a cluster-based array.

The topology of MoT is shown in Fig.5.3. The leaves of the tree are exactly $N \times N$ nodes of mesh. In each row and each column, wires and additional nodes are connected to form a binary tree. MoT

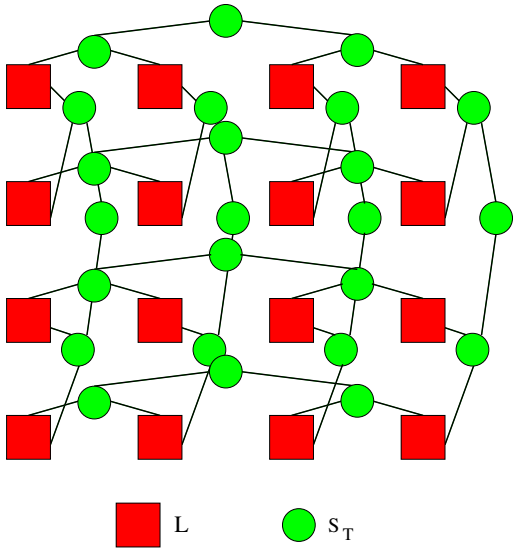


Figure 5.3: MoT [1]

has polylogarithmic $\Theta(\log N)$ or $\Theta(\log^2 N)$ running time on a wide range problems described in [1], which is dramatically faster than the typical running times of $\Theta(N)$ or $\Theta(N^2)$ for algorithms on meshes or trees. Further, works in [110, 107] show that mesh of tree permits area-efficient layout.

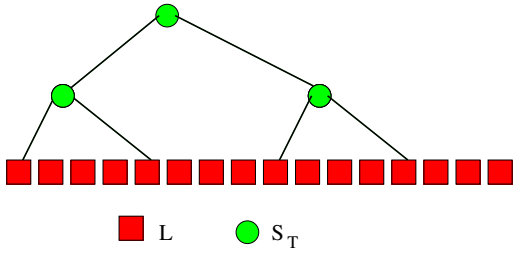


Figure 5.4: A tree with N/k leaves

After construction, N/k leaves are put together remotely through a tree network (shown in Fig.5.4, where $N = 16$ and $k = 4$). For the mesh array (level 0), we assume the same structure as the typical island-style FPGA. However, $k \times k$ CLBs are grouped as a cluster in which local routings are performed. The long connections between clusters are routed through tree networks if those tree routings are possible.

The switch blocks allow the needed inter- and intra-cluster connections to be realized. Since the same interconnect structures of island-style are applied at level 0, we assume that the typical connection topology Fig.5.5(a) such as disjoint or subset switch block is used. As to the tree switch

block *"S_T"*, a simple 3-side interconnect pattern is used (shown in Fig.5.5(b)). The number of parent pins is set to the summation of pins of two children. That is, every child pin can respectively be connected to a parent pin. For the connections between children, the flexibility of child pin is set to the number of pins of one CLB. By doing so, it provides the full connectivity at the beginning of each tree network. In practice, it is not necessary to guarantee that all connections would go through tree network since most routings are performed within clusters in a mesh fashion. To understand how many long connections would go through the tree network, a stochastic instead of combinatorial analysis may reveal more associations, and we leave it as the future investigation. As we will see in next section, even at worst case, this connection scheme will not significantly increase the number of switches needed in the proposed architecture as the sizes of trees are relatively small (N/k, instead of N^2).

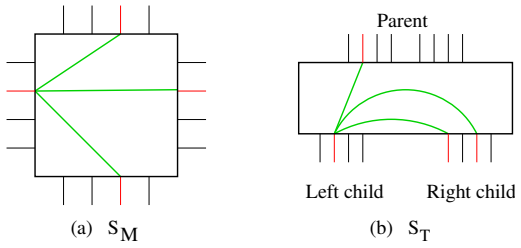


Figure 5.5: Connection patterns used in mesh and tree switch blocks

Normally, in a mesh array, each CLB has four neighbors (east, south, west and north). By applying the proposed hybrid network, the neighbors have been expanded from four to six (Fig.5.6). Two more neighbors are the siblings of that leaf in its two trees. Note that all neighbor connections could be realized by only one or two switches. With those expanded neighbors and imposed tree networks, it is possible to support more short connections, hence achieving performance improvements.

5.2 Model Analysis

Since switches dominate FPGA design, from FPGA architect's view, it is highly expectable to achieve high performance and high routabilty with the available switches in that FPGA. Routabil-

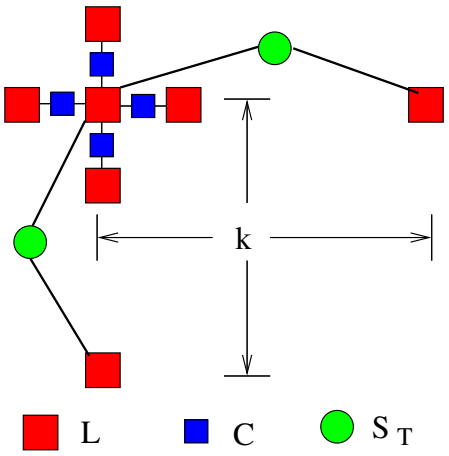


Figure 5.6: Neighbor expanding

ity describes the effectiveness in utilizing the programmable routing resources. Although there are lots of discussions about the issue of routing ability, unfortunately, there is no formal definition of routability. General understanding is that the routability of FPGAs depends strongly on the number of available connections. Bisection width is the vital factor which may measure the connections. The higher the bisection width, the larger the space of routing solutions and the easier the routing. We use the bisection width to evaluate the switch requirements between different architecture models.

5.2.1 Comparison of Switches Needed

Let 's consider two FPGA models: Type-A, the conventional array FPGA (Fig. 5.1), and Type-B, the proposed FPGA, which is a hierarchical combination of mesh and tree interconnects mentioned in the last section. For the further analysis, let m denote the number of pins of a CLB, and I is the number inputs and outputs from one side of CLB to the channel. We have $I = m/4$ for mesh architecture. For example, $m = 8$, $I = 2$ in Fig.5.1. The sizes of two FPGAs are set to $N \times N$. In order to be comparable, all bisection widths at highest level are set to be equal, which is $BW = pmN^2$. Here, mN^2 is the total number of possible signals. Since any signal may cross the middle line of chip several times or not pass it at all, we multiply it by a parameter p. Let W_a be the channel widths of Type-A model. According to Fig.5.1, there are N connection blocks and

$N-1$ switch blocks per row/column either along horizontal or vertical dimension. The bisection width can be counted as:

$$BW_A = N \times I + 2 \times (N-1) \times W_a \tag{5.1}$$

Let $BW_A = BW$ and substitute I with m, we have:

$$W_a = \frac{mN(4pN-1)}{8(N-1)} \tag{5.2}$$

Following the definitions in [108], if we assume the flexibility of connection block is W_a and the flexibility of switch block is F_s, then for the Type-A FPGA, the total number of switches is:

$$\begin{aligned} SW_A &= N^2 \times (2 \times 2 \times I \times W_a + \binom{4}{2} \times F_s \times W_a) \\ &= \frac{m(m+6F_s)(4pN-1)N^3}{8(N-1)} \end{aligned} \tag{5.3}$$

In the above calculation, we associate each logic block with one switch block and two connection blocks. Therefore, the total switches are the N^2 times of summation of switches in one switch and two connection blocks.

For the total number of switches of Type-B model, we approximate it as the summation of the mesh switches and tree switches. Let us estimate the switches used in the tree connections first. The total number of switches needed for the trees in the proposed interconnect is as following

$$\begin{aligned} SW_T &= 2kN \sum_{i=1}^{\log \frac{N}{k}} (2 \times 2^{i-1}m \times m + 2^i m) \times 2^{-i} \times \frac{N}{k} \\ &= 2m(m+1)N^2 \log \frac{N}{k} \end{aligned} \tag{5.4}$$

We give some explanations for SW_T. There are totally $2kN$ trees. In a tree network, at ith level, where $1 \leq i \leq \log \frac{N}{k}$, the switches for the connections between one child and the other child in one single block are $2^{i-1}m \times m$, the switches for the connections between parent and children in that block are $2^i \times m$, and the numbers of switch blocks are $2^{-i} \times \frac{N}{k}$.

Next, we look at the switches used in the mesh connections. Recall that the proposed FPGA is

cluster-based. This implies that there are two kinds of channels. One is the channel within a cluster, while the other is the channel between clusters. Two channels can be viewed as the results of different routings. The former is come from the local routing, while the latter is affected by the long routing. In order to keep the same bisection width at each partition level between those two models, obviously, the channel width within the cluster should be set to W_a. The channel width (denoted as W_b) between clusters is calculated as following.

The bisection width of proposed model is $NI + 2(N-1)W_b + mN^2/2$. Similar to BW_A, the first two items are the mesh bisection widths of proposed FPGA. The last item is the tree cut, which is calculated as follow. The cut will bisect k trees per row or column, and there are totally N rows or columns. Since each logic block has m pins, the total cut of tree will be $k \times m \times N \times 2^{\log \frac{N}{k}-1}$, which is $mN^2/2$.

$$W_b = \frac{mN(4pN - 2N - 1)}{8(N-1)} \tag{5.5}$$

Finally, the totally switches needed for the proposed model would be approximated as the summation of all switches used in the clusters, between clusters, and in the tree network, which result in the following equation.

$$\begin{aligned} SW_B &= \frac{N^2}{k^2}(mk(k-1) + 6F_s(k-1)^2)W_a \\ &\quad +2N(\frac{N}{k} - 1)(m + 6F_s)W_b + SW_T \end{aligned} \tag{5.6}$$

In the above equation, the first item is the number of switches needed for all clusters. It is calculated as following. Since there are totally $\frac{N^2}{k^2}$ clusters, for each cluster, the numbers of connection and switch blocks are counted as $k(k-1)$ and $(k-1)^2$ respectively. The second item is the total number of switches required for the connections between clusters. There are totally $\frac{N}{k} - 1$ channels in one direction and the number of switch and connection blocks are approximated as N for each channel. By substitution, we have

$$\begin{aligned} SW_B &= \frac{m(k-1)(mk + 6F_s(k-1))(4pN-1)N^3}{8k^2(N-1)} \\ &\quad +\frac{m(m + 6F_s)(4pN - 4k - 1)(N-k)N^2}{4k(N-1)} \end{aligned}$$

$$+2m(m+1)N^2 \log \frac{N}{k} \tag{5.7}$$

From Eq.5.3 and Eq.5.7, we may see that both models need $O(N^3)$ switches to achieve the same number of connections. We would like to see the effects of various parameters. In order to compare the switch requirements between those two models more clearly, we sketch the percentages of increased switches ($\frac{SW_B - SW_A}{SW_A} \times 100\%$) for different values of k with different m, F_s, p in Fig.5.7, Fig.5.8, and Fig.5.9 respectively. During each calculation, N is set to 128, since this number can cover the sizes of all commercial FPGAs currently available. For the non-changed parameters, we fix $F_s = 3$, $p = 0.5$, and $m = 40$ correspondingly. When compared with conventional architecture, the switch increments required in the proposed structure are largely depended on the size of cluster while kept in the same order.

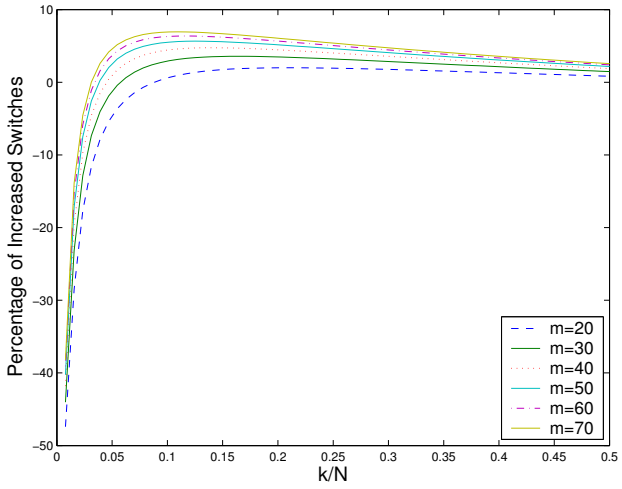


Figure 5.7: Switch overhead for various values of m

Obviously, in terms of switches overhead, the smallest k is desirable. For the lowest point which corresponding to $k = 1$, the switches required for each cluster would be zero from the Eq.5.7. This implies that the tree switches will dominate the total number of needed switches. From Eq.5.3 and Eq.5.4, we know that the switches required for the mesh and tree are $O(N^3)$ and $O(N^2 \log N)$ respectively. This is why the large reductions are observed in those figures when k is smaller. The increased switches in the tree network are balanced by the channel width reductions in the mesh array. In the later section, based on a series of benchmark studies, we will see this is exactly true.

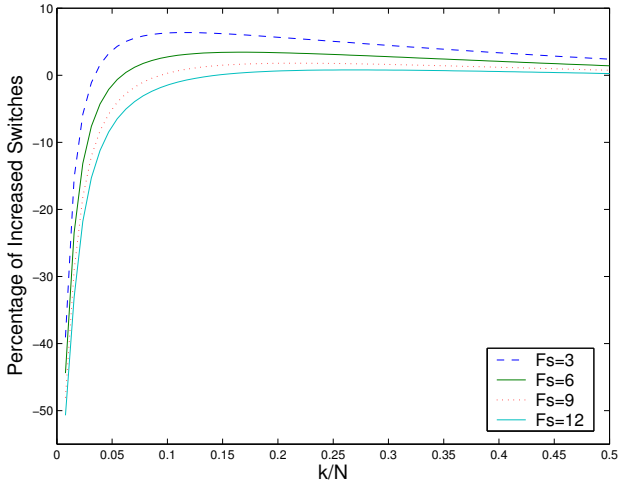


Figure 5.8: Switch overhead for various values of F_s

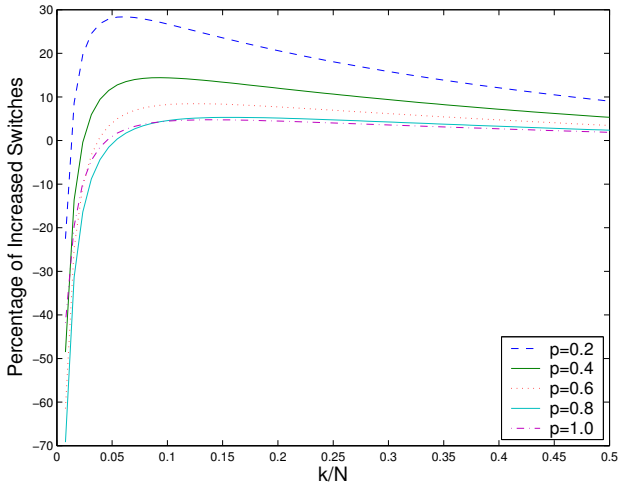


Figure 5.9: Switch overhead for various values of p

k can not be too small, otherwise the dominated tree connections would be easily suffered from congestion problem due to the lack of alternative mesh connections. Therefore, we only look at the switch overhead for $k > 1$. The mesh switches begin to dominate when k is slightly increased.

For the two architectural parameters m and F_s, the increased switches are under 10% even in the worst cases in the above figures. For parameter p, which describes the connection richness in the implementation of a design, we observe that the switch overhead is much high when p is small. From Eq.5.5 the channel width between clusters would be approaching to zero when p is decreasing to certain value. This means that the tree networks can provide much more connections than those

required for the design. Since the conventional array FPGAs need less channel tracks to implement that design (hence the less switches, see the beginning in Section 5.1) while the tree switches in the proposed model remain the same, the overhead will be high. When p is growing larger, the overheads are still comparable with other two parameters.

One important property is that the switch overheads are decreasing quickly when parameter p or F_s is increasing. This would benefit FPGA design a lot since today both FPGAs and connections in a design are becoming larger and larger. We also observe that the switch overhead is decreasing when the ratio $\frac{k}{N}$ is increasing. However, a larger k will undermine the performance improvement. Since most nets in FPGAs are short nets, the performances of those nets implemented in mesh routing are limited to k. Further, we will see next that the performance improvement of long connections due to tree networks is also weakened when k is larger.

5.2.2 Performance Gain

The proposed architecture encourages the short implementations of signal paths, which can be either realized through mesh routings for the short connections, or through tree routings for the long connections. Since the performance of an FPGA can be improved by reducing the number of passing switches along signal path, it is reasonable to relate the number of switches to the wire length. We are looking at the average case and use the methodology [111] to approximate the average connection length $\overline{l}$.

$$N(l) = q(l)D(l)[111] \tag{5.8}$$

Where $N(l)$ is the number of net of length l, $D(l)$ is the number of valid two-pin net placement sites and $q(l)$ is the probability that a placement site is occupied. Therefore, we have

$$\overline{l} = \frac{\sum_{l=1}^{2N-1} N(l)l}{\sum_{l=1}^{2N-1} N(l)} \tag{5.9}$$

Further, if we denote $\overline{l}(k)$ as the average length of all connections whose lengths are greater than k, we have

$$\overline{l}(k) = \frac{\sum_{l=k+1}^{2N-1} N(l)l}{\sum_{l=k+1}^{2N-1} N(l)} \tag{5.10}$$

For a pure mesh model, the switches used for above average length are approximated as:

$$\overline{l}_A = \sqrt{2}\overline{l}(k) + 1 \tag{5.11}$$

$\sqrt{2}$ is employed to measure the distance from Euclidean to Manhattan in the above equation. For the proposed model, although we expect all long connections are routed through tree networks, we use the normalized routing possibility to model the routing algorithm since those connections could be either realized through mesh or tree routings. Obviously, the performance of proposed FPGA is dependent on the size of cluster. When $k = 1$, most of nets are routed through the tree network and we may get the tree 's performance at the best case. On the other hand, when the size of cluster is large enough, such as $k = N$, nets are exactly routed through Manhattan scheme and the mesh 's performance is achieved.

$$\overline{l}_B = ((2\log(\sqrt{2}\overline{l}(k)/2) - 1)\frac{N-k}{N} + (\sqrt{2}\overline{l}(k) - 1)\frac{k}{N} \tag{5.12}$$

We sketch the above two equations and their ratios in Fig.5.10 for different lengths of long connections (namely, k). From that Figure, the best range of improvement is around $[4, 10]$. To see that improvement clearly, we redraw that gain in Fig.5.11 according to $\frac{\overline{l}_A - \overline{l}_B}{\overline{l}_A} \times 100\%$. The peak point is at $k = 9$ with achieving as much as 75% improvement in Fig.5.11. Notice that the above improvement is only for the implementations of long nets, while the switch requirements for short connections are assumed to be comparable since the short connections would be implemented through mesh routings for both models.

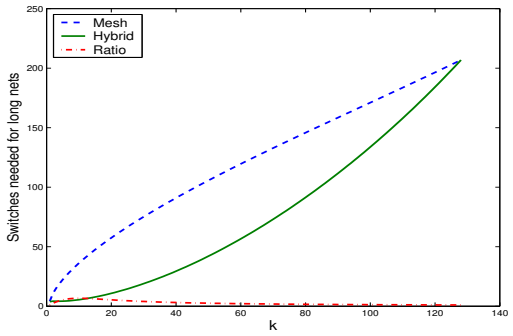


Figure 5.10: Switches required for the long nets

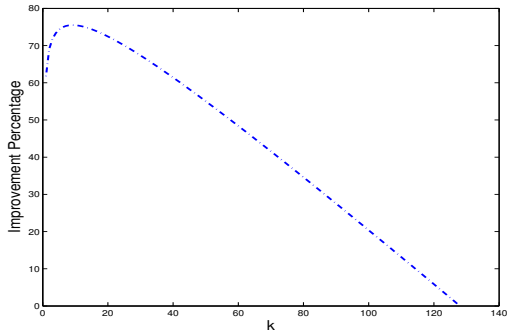


Figure 5.11: Performance improvement of the long nets

5.2.3 Further Investigations

In this section, we will present more investigations between the proposed structure and interconnects in the other FPGAs (including the commercial FPGAs and FPGA models from other researchers). In commercial FPGAs, such as Xilinx Virtex FPGAs, there are several kinds of routing resources: general purpose routing, global routing, I/O routing, and dedicated routing. The general purpose routing resources are located in horizontal and vertical routing channels associated with logic blocks, here we first look at this kind of routing since it is related to the topic we are talking about. For other routing resources, they will be investigated and compared with presented structure only when necessary. From the interconnect view, the general purpose routing resources can be divided into adjacent interconnect, direct interconnect, and long interconnect. The long interconnect can be further divided into double line, hex line, full-length long line, and so on according to the staggered patterns. Similar interconnect schemes have been adapted in Stratix FPGA from Altera and EC/ECP FPGA from Lattice Semiconductor. Table 5.1 lists the different strategies used in ma-

jor FPGAs which are current available in the market. The numbers in the first column are the logic blocks passed by the corresponding staggered patterns, and the dash line means no corresponding pattern is applicable for that FPGAs.

Table 5.1: Staggered Patterns in Commercial FPGAs

Company	Xilinx	Altera	Lattice Semiconductor
FPGA Device	Virtex	Stratix	ECP/EC
Logic block	CLB	LAB	PFU
2	double line	-	X1
3	hex line	-	X2
4	-	R4, C4	-
6	hex line	-	-
7	-	-	X6
8	-	R8, C8	-

From the table 5.1, we can easily find that the commercial FPGAs use the lines with different lengths for the implementations of long connections. Let's denote it as segment pattern strategy. Obviously, segment pattern strategy can benefit for high speed access across the given connection length. Another advantage from segment pattern is the predictable and repeatable performance for the software compiler and hardware migration. This kind of strategy is seemed to be used more intensively in the latest FPGAs (See Table 5.2).

Table 5.2: Comparison of Number of Segments

Segment	Virtex	Virtex II Pro X	Increment
single length	24	18	-8
double line	0	40	40
hex line	12	120	108
long line	12	24	12

Now you may ask why the segment pattern strategy becomes more and more popular, and what's advantage of the proposed hybrid interconnect over segment pattern strategy? Actually, the introducing of segment pattern strategy does not change the nature of mesh network. Although different segment lines can reduce the switches required for the long connections, extra switches are needed for the interconnects among IOs of logic blocks and the corresponding routing channels. This is why in practice no significant increase or reduction of switches is observed when segment strat-

egy is applied in mesh interconnect. Segment pattern strategy does improve the performance of FPGAs. Especially when both designs and FPGAs are relatively small, the performance of final implementation may be determined by the several critical paths, and the software can automatically place critical design paths on faster interconnects to improve design performance. The CAD software designer also loves the fixed segment patterns because that will simplify the development of heuristic algorithm by eliminating the re-optimization cycles in the routing process. Today both FPGAs and designs are becoming larger and larger, more longer segment patterns are needed to satisfy the performance requirements. This is one of reasons of the significant increments of longer lines observed in Table 5.1. On the other side, due to the lack of flexibility, CAD tool also benefits a lots from the increase of long segments for the arrangement of high priority routes. The proposed hybrid interconnect can satisfy all needs above. Further, it is more flexible by comparing with fixed segment pattern. Fixed segment patterns limit the accessible logic blocks and increase the channel width, therefore put much more restrictions on the physical synthesis stages, such as mapping, placement and routing processes. The proposed hybrid interconnect removes those restrictions, improves the resource utilization, and results in the dense and optimal realizations.

We also compare the presented model with two typical structures proposed by other researchers. In [104, 105], the fat-tree was used as as basic interconnect, and its goal was to build high speed FPGA by inserting the registers at proper locations of signal path. The insertion was performed by re-timing during mapping stage. In our model, we use the simple interconnect pattern since the tree levels is much less than those in [104, 105], and there is no re-timing issue during synthesis for performance for our model. Studies in [106, 107] dealt with the switch block design and layout area effects. It staggered the interconnection between two consecutive levels of tree to achieve the different Rent's parameters. MoT was introduced to maintain the property that each compute block contained a constant number of switches independent of the design size. It demonstrated that by given a sufficient multi metal levels the gate density remained constant across increasing gate counts. In our tree switch box, at the leave level connection, the parent pins is set to the total child pins, which may be a "stupid" interconnect at the first sight for that kind of interconnect would increase the total switches significantly. However, our interconnect structure is cluster-based, and

we also apply the MoT to reduce the tree levels. Further, the total number of MoT in presented structure is k^2 while only 1 in the above research work. If we ignore the switch block design and try to seek the similarity, and if applicable, the above structure is the special case of our interconnect (cluster size =1). Another major difference is that the neighbor computation blocks in the above researches are 2 or 4, while the number of neighbors is kept as 6 constantly in our model due to more MoTs have been used. We believe that the combination of tree network, cluster connection, and extra neighborhood provide the possible ways for the improvements both the architecture design and implementation performance of FPGAs since both short connections such as dedicated routing (carry chain, shift chain, and so on) and long connections such as global routing (distribute clock, high fanout signals, and so on) are tending to be essential in modern FPGAs.

Besides the above features, the regular structure in our proposed model makes it possible to adapt the available CAD tools, next section will partially demonstrate this advantage. Finally, the authors want to point out that it is never the intension that this hybrid interconnect overcomes all other structures and is a perfect model to the FPGAs. This would be supremely arrogant, and anyway impossible. In this manuscript, we study theoretically some promising abilities of proposed model while ignoring the fabrication issues. The further advocation, if any, would be come from the practice in real-world.

5.3 Evaluations

We present the experimental studies in this section. We use T-VPack and VPR (v 4.30) [112] to map and place the design. T-VPack is a logic block packing program, which packs LUTs and flip-flops into coarser grained logic blocks and converts a netlist to VPR 's format. VPR is a placement and routing tool for array-based FPGAs. We use sample architecture file 4_4-*lut_sanitized.arch* as our base mesh unit. Therefore, each CLB has 4 4-LUTs, 10 pins for input and 4 pins for outputs. We first use T-VPack to pack the netlist to support the sample architecture, and then use VPR to produce the placed design. By default, the VPR tries to place the circuit in the smallest rectangle

in which the circuit fits. After placement has been done, we stretch that rectangle into a large mesh structure. According to the cluster size, the size of stretched rectangle is the least 2 's power which is not less than original size. Since array sizes has been changed, we run VPR again to update the placement with the newly stretched sizes.

5.3.1 Routing Process

For the conventional model, VPR (running at default settings) is used as the routing tool. For the proposed FPGA model, since it allows routing to be done in both mesh and hierarchical tree interconnects, we have developed a custom routing tool to take advantages of those features. We constraint the short nets (all terminals of a net are guarded within the same cluster and the neighboring CLBs of that cluster) are routed through mesh routing. For a net which has terminals scattering at different clusters, the routing process is as following.

At the global routing step, the terminals are grouped according to their corresponding clusters. The routings for the terminals of each group are performed within that cluster. The connections between groups are the long nets which will go through the tree networks as much as possible. Hence, our routing process consists of two routing procedures: tree and mesh routings. The tree router is running on the top of mesh router. The tree routing algorithm is based on [113] while the algorithm sticks to the breadth first search in VPR for the mesh routing.

As to the tree routing, there are two main issues. One is the net ordering, and the other is the path search order. The ordering of all long nets is created to be fixed according to their lengths. The key of path order is to determine which gross tree (totally $2kN$) is chosen as the viable route. If there exists two terminals which belong to two leaves of a tree, the path crossing that tree is only route for choice. Otherwise, we search the trees with the intermediate leaves which are available to be connected to both two terminals with less distance than that of Manhattan scheme. Assuming that we have to route a net from location $(x1, y1)$ to location $(x2, y2)$, location $(x3, y3)$ and $(x4, y4)$ are the intermediate CLBs which are located in the same clusters of $(x1, y1)$ and $(x2, y2)$ respectively.

The routing path can be determined in one of the following ways. From $(x1, y1)$, through $(x1, y2)$ to $(x2, y2)$; from $(x1, y1)$, through $(x2, y1)$ to $(x2, y2)$; from $(x1, y1)$, through $(x3, y3)$, $(x3, y2)$ to $(x2, y2)$; from $(x1, y1)$, through $(x3, y3)$, $(x2, y3)$ to $(x2, y2)$; from $(x1, y1)$, through $(x4, y1)$, $(x4, y4)$ to $(x2, y2)$; from $(x1, y1)$, through $(x1, y4)$, $(x4, y4)$ to $(x2, y2)$; from $(x1, y1)$, through $(x3, y3)$, $(x3, y4)$, $(x4, y4)$ to $(x2, y2)$ and from $(x1, y1)$, through $(x3, y3)$, $(x4, y3)$, $(x4, y4)$ to $(x2, y2)$.

In case of all above paths unavailable, the net may be routed through mesh routings. Further, if there exists two terminals which belong to the different groups but are located in the two neighbor CLBs, that net will go through the local routing resource between those two groups. When there exists two possible routing directions which can be performed for the signal path between two clusters, horizontal first, then vertical routing next; or vertical first, then horizontal routing next, we select the routing direction randomly. In order to be comparable, only one length of segment is allowed in the routing procedures for both models. Further, in order to assure signal integrity and predictable delays, the wire switch of segments in the tree connections are assume to be buffered while it is not in the mesh network.

5.3.2 Results

We mapped, placed and routed the 20 largest MCNC benchmark circuits on our proposed FPGA. We report the minimum number of channels required for VPR and different cluster sizes in Table 5.3. The third column shows the array sizes for implementations of all designs. The minimum track counts for VPR are listed in the 4th column. As can be seen from that table, channel widths have been reduced largely. Even for a larger k, up to 40% reductions have still been achieved for the larger designs, such as *clma*. We also observe that the increase of channel width is slow when the cluster size increases, and for each cluster size, the deviations of channel widths are relatively small. This is especially useful, because more designs would be admitted for a given FPGA with a fixed channel width.

Table 5.3: Array sizes and channel widths

Design		Array		k			
ID	Circuit	nx, ny	VPR	2	4	8	16
1	alu4	32, 32	32	13	16	20	26
2	apex2	32, 32	37	13	17	25	31
3	apex4	32, 32	40	13	15	24	31
4	bigkey	32, 32	21	13	14	17	19
5	clma	64, 64	56	16	17	23	31
6	des	32, 32	19	13	15	17	17
7	diffeq	32, 32	25	12	15	20	23
8	dsip	32, 32	20	14	15	17	18
9	elliptic	32, 32	45	13	16	21	27
10	ex1010	64, 64	43	17	19	24	31
11	ex5p	32, 32	38	8	11	19	31
12	frisc	32, 32	47	13	17	25	32
13	misex3	32, 32	37	13	19	23	31
14	pdc	64, 64	67	15	19	28	42
15	s298	32, 32	30	12	15	17	19
16	s38417	64, 64	33	14	16	21	26
17	s38584.1	64, 64	32	13	15	19	23
18	seq	32, 32	39	13	17	23	29
19	spla	32, 32	61	17	21	27	37
20	tseng	32, 32	24	12	15	17	20

We also plot the routing area and critical path delay for VPR and different clusters in Fig. 5.12 and 5.13 respectively. The routing area is the summation of mesh routings and tree routings in terms of minimum-width transistor area [114]. The minimum routing area of each tree switch is calculated as the following. We assume each tree switch is implemented with a pass transistor, the number of minimum-width transistor required for that is 11.5. As mentioned early, the wire segment of tree connections is assumed to be buffered. We use the same buffer size as in Chapter 7 [114] and it occupies 19.7 minimum-width transistor areas. Therefore, the total area for each tree switch needed in the design is 31.2 minimum-width transistors. As to the delay, we use the same timing numbers in the sanitized architecture file. As explained in the VPR documentation, although they are not complying with the foundry process data, those numbers are still reasonable enough to allow CAD experimentation. From Fig. 5.12 and 5.13, it is clear that considerable improvements are achieved. For the routing area, we can see as much as 30% reduction is achieved. This is because

most of nets are routed through tree networks, resulting in a huge saving of switches needed. More significant achievements are observed for the performance improvements. Up to 50% reductions are possible for the large designs. Again, the improvements are due to our intentional introduction of the tree network. From those two figures, the larger the design, the more the improvement for both routing and performance. We notice that improvements increases when k decreases. While a small k is desirable, we also notice that the performances are degraded for the design ID 4, 7 and 14 when k is further reduced. This may be explained when k is too small, the routings will go through more mesh interconnects instead of tree connections due to the congestion problem. Since we enlarge the array size, much more tree networks are available for the routings of long nets and the congestion is largely relieved. This is one of limitations of our experiments. Therefore, considering those factors, a slightly large k is more implementable.

5.4 Summary

The routing area and performance in FPGAs is generally limited by the programmable interconnects. In this chapter, we presented a cluster-based FPGA with a hybrid interconnect structure which takes advantages of both mesh and tree interconnect networks. The presented architecture is developed with a combinatorial analysis which examines the number of switches needed. We evaluate the proposed architecture on a set of MCNC circuits. The benchmark studies show that our architectural model has less switch accrued effects due to the introduction of tree interconnects. By encouraging local routing and short implementations of long connections, significant reductions in the routing area and long path delay can be achieved. In additional, the CAD tools for the tradition island-style FPGAs, considering its immense popularity, can be easily adapted.

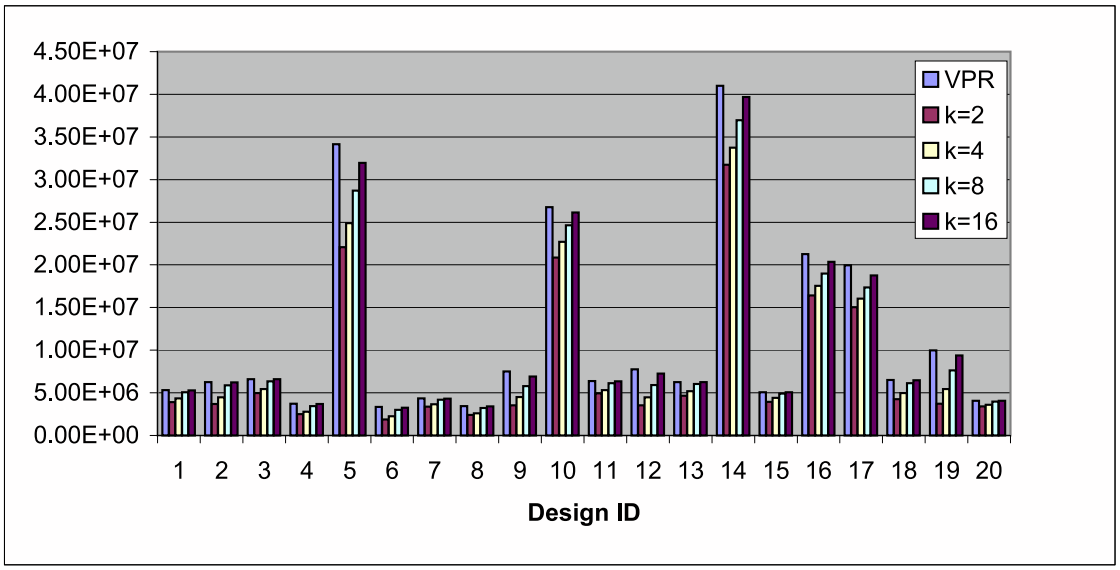


Figure 5.12: Routing area improvement

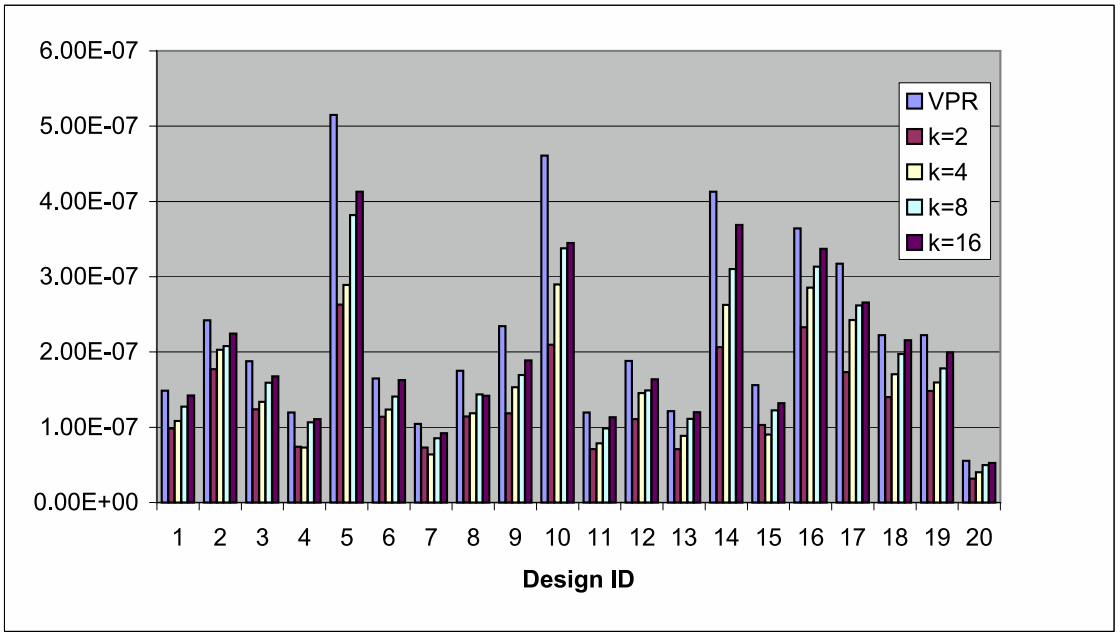


Figure 5.13: Performance improvement

Chapter 6

A Hybrid Interconnect Architecture for Dynamically Reconfigurable FPGAs

The performance and the capacity of FPGAs have increased manifold in the past few years. Since it is not practical to completely reconfigure such high capacity devices at one time, the support for dynamic or partial reconfiguration is a must. Latest generations of high-density and high-speed FPGAs, are dynamically reconfigurable, that means the reconfiguration affects only a part of the FPGA while other parts remain working. The run-time and partial reconfiguration capabilities of the state-of-the-art FPGAs [75] have shown potentials for a large number of novel applications of the FPGAs.

Rapidly increasing interests on Run Time Reconfigurable Computation incite both industries and academic are investing more and more efforts to emphasize on reconfigurable computing. Numerous research projects focus on new reconfigurable architectures definitions, either encouraging nearest neighbour links [115], embedding multiplier, even integrating with general-purpose CPU [75, 116, 117], or targeting computational datapath applications such as pipelining [118], multimedia [119], or tele-data communications [120]. The reconfigurable units in those studies are coarse-grained, which are column-based or word-level configurable. Major benefit of course-grained architectures is the drastic reduction of placement and routing complexity and the reduc-

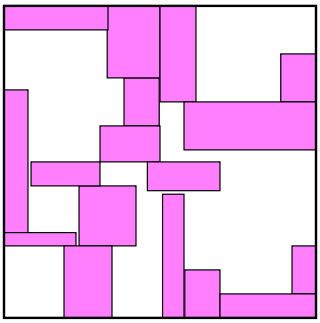


Figure 6.1: Lack of contiguous free resources for the task in Fig.6.2

tion of configuration time. However, coarse-grained architecture exhibits less flexibility than the corresponding fine-grained architectures, and is suffered from area fragmentation.

In this chapter, we present a cluster-based reconfigurable architecture which supports multi-granular (coarse- and fine-grain) reconfigurations. The developed architecture is an extension of the model which is presented in last chapter. To assess the proposed model, we develop an evaluation tool with ability of simulating run-time placement and routing effects on a given reconfigurable architecture.

6.1 Motivation

Capacity of the FPGAs has reached ten million gate counts. It is possible to run multiple independent tasks (e.g. the one shown in Fig.6.2) on the same FPGA using partial reconfiguration. However, executing multiple tasks on the same chip simultaneously can impose a lot of challenges. One main problem is fragmentation (more than 50% from our studies, Fig.6.10) of the FPGA area due to the dynamic addition and deletion of tasks. Lack of contiguous free area resources can prevent placement of subsequent tasks on the FPGA. Figure 6.1 shows example of fragmented FPGA area. Even if the task is placeable, it may not be routable due to limited routing resources. Most of the previous work do not answer these questions and assume that sufficient resources area available of task placement and routing.

Both course grained architectures and the fine-grained architectures have their advantages and

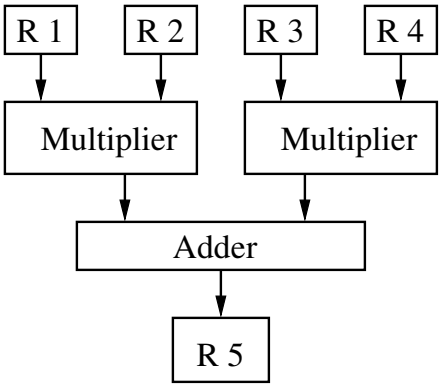


Figure 6.2: A common computational task used in DSP applications

disadvantages. Most the recently proposed architectures are domain-specific course grain architectures. Those customer-base architectures are limited to the domain they belong to. Also, these course-grained architectures have less flexibility. In comparison, the fine grain architectures exhibit better area utilization and more flexibility. Reconfiguration time of fine grain architectures is comparatively more.

We envisage that the future reconfigurable architectures will support multi-granular reconfiguration. They will have a coarse-grained block-based logic and a fine grain interconnect network that will have enough flexibility to satisfy different communication needs of various tasks. In this chapter, we propose a cluster-based reconfigurable FPGA architecture which is the extension of FPGA model proposed in last chapter. As we will show in this chapter, the extended model has advantages over both coarse- and fine-grain model in terms of the area utilization. The contributions of this chapter are:

- Proposed a reconfigurable FPGA model by imposing more tree interconnect into mesh networks to get over fragmentation problem and support multi-granular reconfigurations.
- Developed an evaluation tool to assess the run time place and routing effects on a dynamically reconfigurable platform.

6.2 Model Extension

The two-dimension mesh interconnect works well for the short connections and tree networks is more attractive to routing the non-neighbor connections. In the above section, due to the lack of routing resource, we see that a task which consists of a set of macros may not be configurable even there are enough free space available in the reconfigurable platform. In the last chapter, we proposed a FPGA model which integrates tree networks into mesh interconnect, and observed improvements on both routing area and delay. In this chapter, we still assume the same architecture which consists of a mesh array of reconfigurable units (RUs) with a hybrid interconnect, and those RUs have CLB-like internal structures. However, we extend that model so the mesh-of-tree networks are imposed on the mesh array while the mesh interconnect remains the same as that of array FPGAs. Namely, we still keep the same number of tracks in the channels and the tree networks are the imposed extra connections when compared with the conventional FPGAs.

6.2.1 Connection Hierarchy

The reconfigurable units (RUs) are arranged as a mesh array and we assume the same interconnection structure as the typical array FPGAs. Same as the description in last chapter, the RUs are grouped as a cluster. Since our extended model has much more extra connections, both short and long connections between clusters or with cluster can be routed through either mesh or tree networks if those networks are available and the routing through that interconnect is necessary. Further, grouping RUs into clusters have two obvious benefits from configuration point of view. Since most of the applications need reconfiguration of more than one RU, RU clustering proves to be more efficient. Moreover, the cluster organization would fast locate target area and reduce the size of hardware decoder. Similarly, the neighbors of each RU have been increased from four to six. Note that although we have not drawn those neighbors, all neighbor connections can be implemented by only one (through tree) or two (through mesh) switches. Increased neighbors can help in realizing multi-granular configurations.

6.2.2 Switch Overheads

Following the same procedure in last chapter, the needed switches for mesh and tree networks are calculated and listed as next two equations:

$$\begin{aligned} SW_A &= N^2 \times (2 \times 2 \times I \times W_a + \binom{4}{2} \times F_s \times W_a) \\ &= \frac{m(m+6F_s)(4pN-1)N^3}{8(N-1)} \end{aligned} \tag{6.1}$$

$$\begin{aligned} SW_T &= 2kN \sum_{i=1}^{\log \frac{N}{k}} (2 \times 2^{i-1} m \times m + 2^i m) \times 2^{-i} \times \frac{N}{k} \\ &= 2m(m+1)N^2 \log \frac{N}{k} \end{aligned} \tag{6.2}$$

We sketch the percentages of increased switches for different values of k with different m, F_s, p in Fig.6.3, Fig.6.4, and Fig.6.5 respectively.

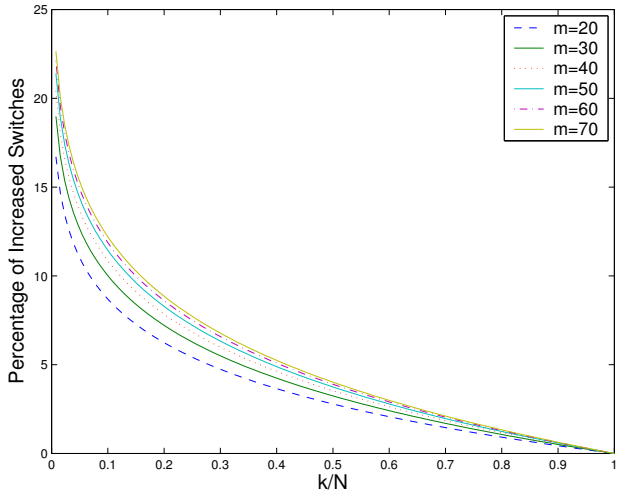


Figure 6.3: Percentage of increased switches for different m's

From above figures, it is clear that all switch overheads are decreasing quickly when the ratio

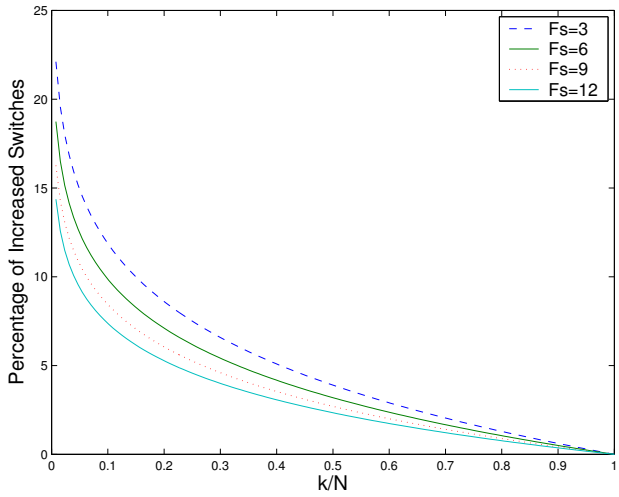


Figure 6.4: Percentage of increased switches for different F_s 's

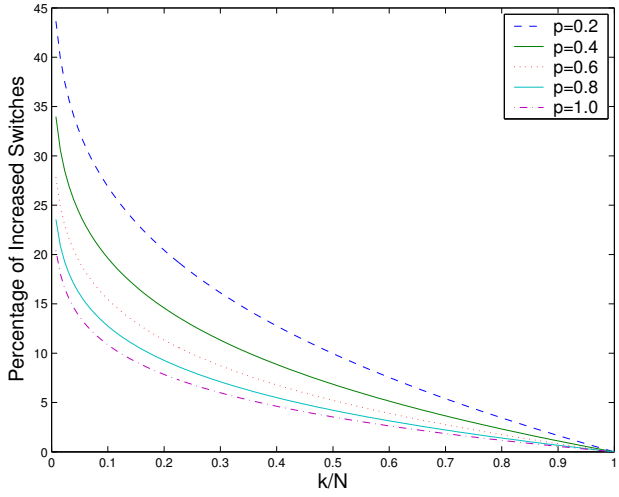


Figure 6.5: Percentage of increased switches for different p 's

$\frac{k}{N}$ is increasing linearly. From Eq.6.1 and Eq.6.2, although we know that the switches required for the mesh and tree are $O(N^3)$ and $O(N^2)\log\frac{N}{k}$ respectively, the switch overhead is a little high when k is relative small. However, a small k would not benefit the design a lot, since most (re)configurations in FPGAs are involved more than one RU, small k will decrease the efficiency of (re)configuration. On the other hand, k can not be too small, since if k is small enough, either mesh or tree interconnect can offer necessary connections for an application to be (re)configurable. Therefore, there are a lot of waste connections. Apparently, k can not be too larger, the benefits from the extra connections introduced by the tree networks would be weakened when k is large

enough.

Another observation is that the switch overhead is insensitive to the values of parameter m and F_s. When m and F_s are increased, both overheads are only slightly (1 or 2 percent) increased. The parameter p, which describes the connection richness in a design, is more meaningful than those of architecture parameter m and F_s. When p is growing larger, the overheads are still comparable with other two parameters. This would benefit reconfigurable computing a lot, since today reconfigurable platform can contain more configurations which are running simultaneously and more connections are involved among those (re)configurations, therefore, it is possible for more applications to be reconfigurable onto the proposed architecture.

For the typical settings ($m = 30, p = 0.6, F_s = 3$, and $\frac{k}{N} = 0.1$), the overheads are under 10%. There are two reasons. First, as we have already seen, the switches required for the conventional mesh array is $O(N^3)$ while it is $O(N^2) \log \frac{N}{k}$ for the tree. This implies that the mesh switches are dominating the total number of needed switches. Second, the sizes of trees are relatively small, which is (N/k, instead of N^2).

6.2.3 Reconfiguration Mechanism

In order to make it dynamically reconfigurable, each cluster is associated with a cluster memory block and that cluster memory block can be used to store reconfiguration instantiation for fast switching.

The basic unit of reconfiguration in our model is a cluster. Just like Xilinx Virtex device [121], the configuration memory for each cluster is arranged into frames, and the frame is the smallest unit of configuration. Each frame is corresponding to a RU within that cluster. This denotes that the smallest piece of memory that can be read or written to is frame size. It is desirable that the targeting reconfigurable platform is used under the control of a host processor. Since there are different applications, and each application consists of a set of precompiled macros (see section 4), a macro can be placed in any locations where it is feasible. The host controller must put some placement

information into configuration instantiation. When switching, only affected configuration frames are updated. The operations of other RUs are not disturbed during the intermediate configuration instantiations.

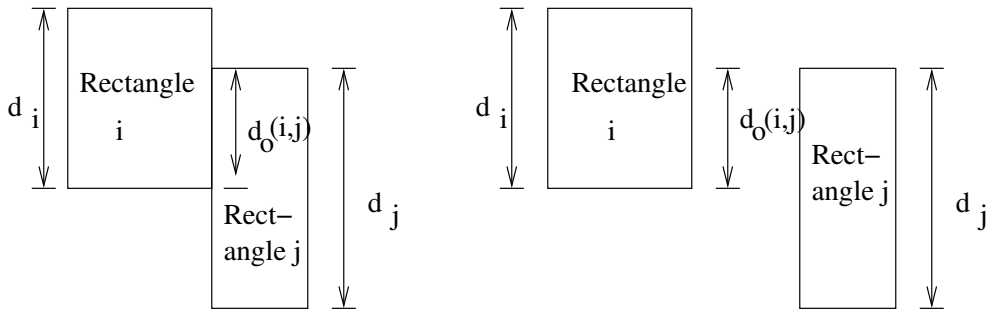
In additional to the cluster configuration, the proposed model also encourages row- or column-based configuration due to good regularity of tree connections in horizontal and vertical directions. However, there is a cost. When only one row or one column RUs in a cluster are to be reconfigured, the entire cluster is affected. Since the configuration of switches requires more configuration bits than the RUs, the above overhead may not introduce a significant increase of the reconfiguration time. Further, if both configurations are performed in parallel, the introduced overhead will not increase any reconfiguration time at all.

Here, instead of detail implementation, we focus on discussion of a cluster-based interconnect structure which is a key aspect in the definition of the reconfigurable FPGAs. Next, we developed an evaluation tool which can simulate run time place and routing effects for a reconfigurable platform.

6.3 Evaluation Methodology

An efficient evaluation methodology can be used to compare the properties of different architectures. We develop an evaluation methodology for comparing our hybrid routing architecture with other state-of-the-art architectures.

In a dynamically reconfigurable system, the applications (tasks) may come at arbitrary arrivals, and those tasks need to be placed on the reconfigurable platform at run time. Fragmentation of the area resources and routability of the tasks are two main challenges faced by such systems. Reduction of the fragmentation comes at the cost of increase in complexity of the online placement algorithm or the hardware architecture. Even if fragmentation is minimized, the resulting circuit may not be routable because of limited resources.



(a) General case (b) Equivalent to (a) in our model

Figure 6.6: Rule NR illustration and equivalence case in proposed model

6.3.1 Run Time System Model

Most online placement approaches [83, 91] model each task as a rectangle with fixed height and width. Tasks area assumed to be internally routed. This may not be true in practice. A task may be made up of a set of macros. Packing all of these macros in a given rectangle may cause more area fragmentation. In our model, the FPGA is modeled as a two dimensional array and the empty space on the FPGA is managed as a list of overlapping maximal empty rectangles [91]. Each task consists of a set of hard macros. A macro can be placed at a location where both placement and routing constraints are satisfied. The routing evaluations are doing by online placement of a set of tasks on the FPGA.

6.3.2 Place and Route Rules

The development of an online placement algorithm that finds a feasible and routable location for a task is not trivial. General greedy algorithms such as simulated annealing or genetic algorithm are not suitable for the run time placement because they are inherently very slow. Our online placement algorithm maintains empty area as a list of maximal empty rectangles. At the run-time, we choose one of these rectangles as per the bin packing rule. Two commonly used bin packing rules are best fit and the worst fit. With combination of following routing rules, online placement and routing effects are simulated.

BF: BF (best fit) chooses the free rectangle with smallest size that can accommodate the macro.

WF: WF (worst fit) chooses the free rectangle with the biggest size that can accommodate the macro.

This strategy leaves enough empty space around a macro to accommodate other macros.

NR: NR (neighboring routable) restricts to connections among rectangles which are neighbors.

We also draw a equivalent case (illustrated in Fig.6.6) of the proposed architecture where two rectangles are not adjacent to each other while still viewed as neighbors due to mesh of tree interconnect.

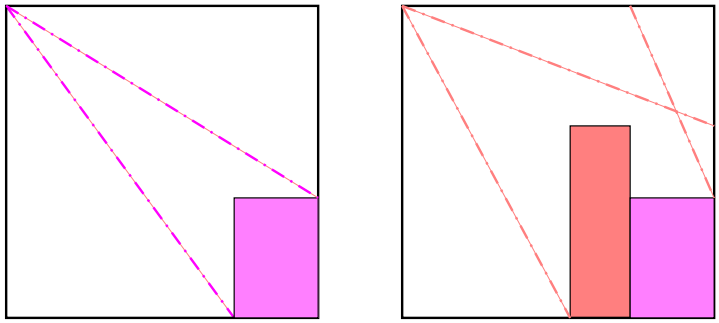
RR: RR (remote routable) uses the following rule to model the routing feasibility P_{RR} of rectangle when all neighboring rectangles are not placeable for the current macro.

$$P_{RR} = \sum_{j} (\frac{d(i,j)}{M_R} + \frac{\log d(i,j)}{\log M_R}) \times \frac{overlap(R_i, R_j)}{Min(R_i, R_j)}$$

where M_R is the maximal Manhattan distance between any two rectangles, and $d(i,j)$ is Manhattan distance between two rectangles R_i and R_j, and rectangle R_j is the free rectangle which can cover the current macro while there are other rectangle(s) between rectangle R_i and R_j. $overlap(R_i, R_j)$ is the overlap length along one dimension between those two rectangles shown in Fig.6.6(a), and $Min(R_i, R_j)$ are the smaller dimensional sizes of those two rectangles along that direction respectively. We select a rectangle for placement only if the P_{RR} is more than 1.

6.3.3 Evaluation Algorithm

We perform online placement of tasks at the run-time. Our goal is to make best use of the reconfiguration resources. Worst Fit rule is used to place first macro of every task. This leaves more free space at the local neighbors of that macro. The macros in a task are sorted from task input to output



(a) Before configuration (b) After configuration

Figure 6.7: The updating after a rectangle is placed

port based on their connectivity order. Each macro has a member element indicating whether there is a connected macro to be placed or not. If that attribute of current macro is true, we select WF rule again, otherwise, BF rule is chosen. After each macro has been placed, we need to update the list of maximal rectangles. This includes two steps. The first one is to generate new maximal rectangles for all rectangles overlapped with that macro. The process is shown in Fig.6.7. Before placement, there are two maximal rectangles labeled by a diagonal dotted line from right-bottom point to left-top point respectively. After placement, three new maximal rectangles are generated. The second step is to find the list L_N of all rectangles which are the neighbors of current macro. For the next macro placement, only after we fail to match from , we will search left rectangles according to the rule RR. At any point in function PR_Evaluation() (Fig.6.8), if a macro can not be placed due to either lack of suitable rectangle to cover it or limited routing resource, the routine would return the flag of a failed status. We denote it as task_rejected (section 6.4). Although the proposed algorithm would not guarantee to get the optimal solution, it caters for the run time features of dynamically reconfigurable environment. Further, compared with other heuristics algorithms such as simulated annealing, it may be more acceptable when the macro number of a task is relatively small.

```
Input:    L_R list of maximal rectangles
          L_M list of macros
Output:   L_R, L_M updated
Function: PR_Evaluation(list L_R, list L_M )
 macro m = L_M.first( );
 do {
        if ( Is_fisrt_macro(m) )
            if ( not worst_fit(m) )
                return task_rejected;
        else
        {
            if  ( L_N(m) )        /* L_N is the list of neighboring rectangles */
            else if ( L_R_N(m) )  /* L_R_N is the list of remote rectangles  */
            else
                return task_rejected;
        }
        m = L_M.next( );
     } while ( m != L_M.last( ) )
 return task_accepted;
```

Figure 6.8: Evaluation algorithm

6.4 Simulation Results

We have developed a framework to evaluate three kinds of reconfigurable models. Model A is Xilinx Virtex-like FPGA, which is partially reconfigurable in vertical, chip-spanning columns. Model B is fine-grain reconfigurable architecture; we still assume the same structure as model A but assume that it is reconfigurable in each RU (CLB). The last one, Model C, is the model proposed in section 3. We assume all three models have the same number 128 128 of RUs

We have a set of policies for selection of a maximal empty rectangle for placement of macro in different models during evaluating process. For model A, the priority of selecting which rectangle to be targeted is tuned to the vertical direction and , while to either horizontal or vertical direction for Model B and C. We choose a MER with minimum width that can accommodate a macro in model A. Further, when a rectangle is chosen, the right bottom point of macro is exactly placed to the right bottom point of that rectangle for Model A and B. For Model C, the right bottom point of macro is matched to the right-est and bottom-est point of the cluster in that rectangle. Another policy is that after a task is configured, all left area in vertical direction within configured chip

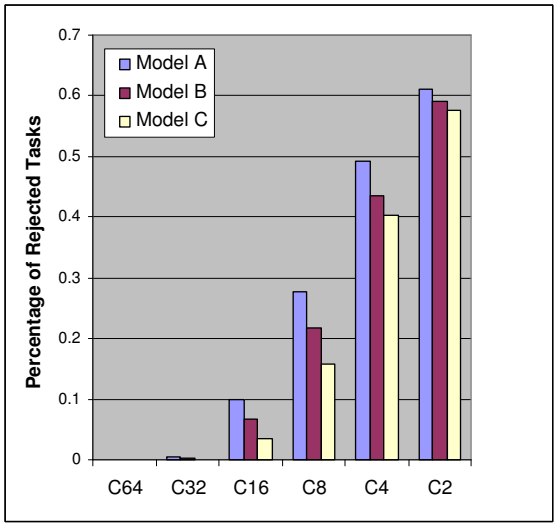


Figure 6.9: Percentage of rejected tasks

can not be further reconfigured until the configured task in that column is removed in Model A; for Model C, the left area in the rectangle where macro is placed cannot be mapped to any macro which has sizes greater than the uncovered region in that rectangle, but can be placed for the small macros with the sizes less than it; while in Model B, all unoccupied area can be further allocated without any constraint except the place and route rules.

We run the intensive simulations using 6 different classes of tasks. The maximal size of task in each class is set to $\frac{1}{64}, \frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}$ of the chip size, and denoted as $C64, C32, C16, C8, C4, C2$ respectively. In each class, we randomly generated the tasks with uniform distribution size in [1, Max_Size_of_that_Class], then the set of macros for each task are randomly generated again with uniform distribution size in [1, Max_Size_of_that_Class] until the total area of all generated macros reaches the area of that task. During macro generation for each task, the connectivity among those macros is also randomly assigned. For the life time of each task, we assume it is uniform distributed in [1, 1000], and the interval time between any two arrivals of tasks has been opted to the uniform distribution in [0, 50] [91]. We have conducted significant simulations for evaluations. We generate 100 tasks for each task class. For each task class, we ran it 100 times to get average values of data. We need to point out that all tasks are assumed to be data independent on other tasks.

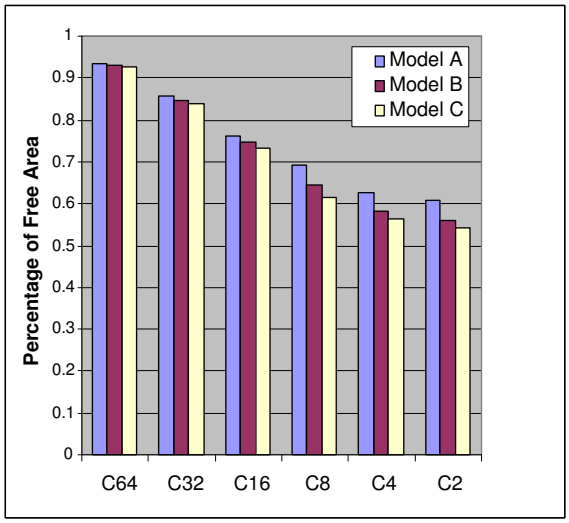


Figure 6.10: Percentage of free area after each task placed

We use two different scheduling methodologies. In first approach, if a task cannot be placed due to placement or routing constraints, that task is considered rejected and no further attempt is made to place it. We denote that task as rejected_task (Fig.6.9). After each task has been configured, we calculate the free area left (Shown in Fig.6.11). In the second scheduling methodology, if a task cannot be placed at its arrival time, it is put into a queue and attempt is made to place it again after deletion of a task from the FPGA. All tasks are executed on the FPGA at the expense of delay in their execution(Shown in Fig.6.13).

In Fig.6.10, we can see that the empty area after each configuration is more than 50% of the chip area. This implies all three models are suffering fragmentation significantly. Compared with other two architectures, our model shows the efficiency on alleviating that problem. Note that the reconfigurable area is pretty larger and some percentage improvements are still significant. This can be further confirmed from Fig.6.9 where the rejected ratio is reduced by 10% for the mid-size task classes. In order to see the difference clearly, we also draw the empty area after each configuration for class C8 in Fig. 6.11.

It is more difficult to place tasks in Model A as compared to the other two models because of more routing restrictions in model A. Similarly, placement is easier in model C than the model B. Although we see the improvements of area utilization for most task classes from Fig.6.10, the larger

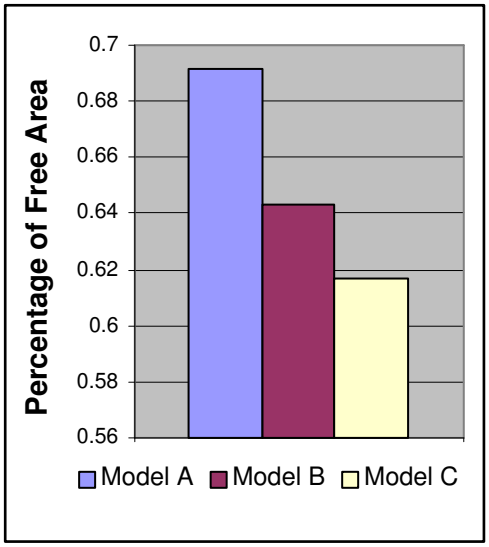


Figure 6.11: Free area per configuration for C8

values makes that figure obscure. In order to see more clearly and measure more precisely, we give result of the average area per unit time in Fig.6.12. For the mid-size classes, 10% improvement is achieved.

Generally, the tasks can be placed without difficulty in all three models for small sizes. For classes C64 and C32, there is almost no task rejected (Fig.6.9) and enough free area left (more than 90% shown in Fig.6.10). When the size of the tasks become larger (such as C2), the rejected ratios of all three models are pretty high and all of them are coming close to the same percentage (60% in Fig.6.9).

We also show the average waiting time per rejected task for a task class in Fig.6.13. It is calculated from the extra time which is the difference between the actual finish time needed to configure all tasks in that class and the ideal accomplishing time of those tasks. The overhead is considerable when the task size is large. While relatively stable for mid and small size, since we sketch the data in that metric for easy comparison in one figure, considering the multiplication with rejected tasks, the improvement should be huge.

Neither smallest size nor largest size class could offer enough messages at those sizes. Therefore, we concentrate on the results of mid-size classes. We draw the ratios of rejected tasks, left area

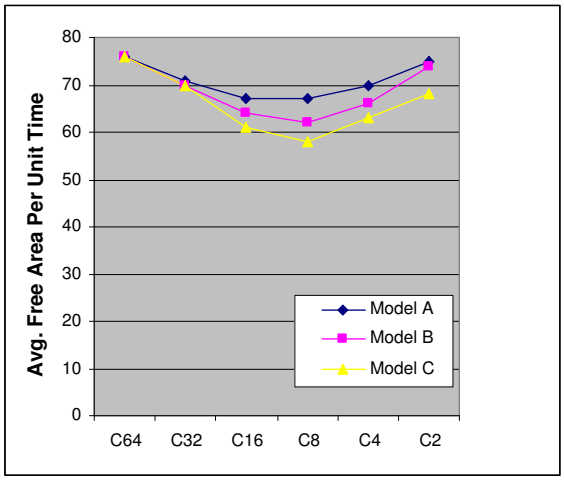


Figure 6.12: Avg. free area per unit time

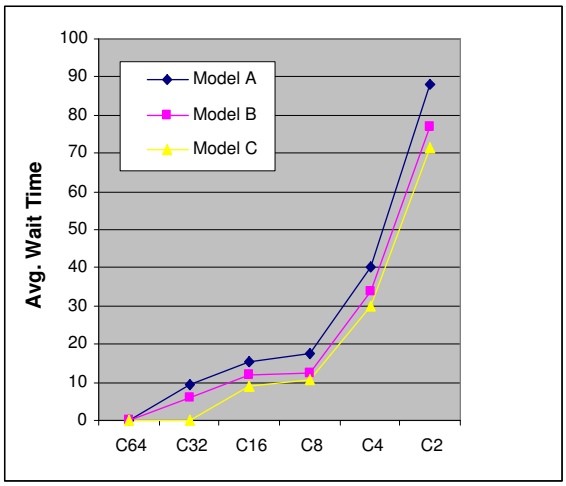


Figure 6.13: Avg. waiting time per rejected task

per unit time and waiting time per task class of three models for the class C8 in Fig.6.14. It is clear that out model has significant improvements in terms of task rejection and the waiting time as compared to the other models. The improvement in area utilization is also noticeable. Although it is not considerable, it is implied that, at any time point, a small improvement on area utilization contributes a lot to the configuration flexibility. This is largely due to its good connectivity of interconnect structure. Since it has enlarged neighbors and offers more routing options between two configuration logics, it suffers little routing limitations. Therefore, it is less vulnerable from fragmentation.

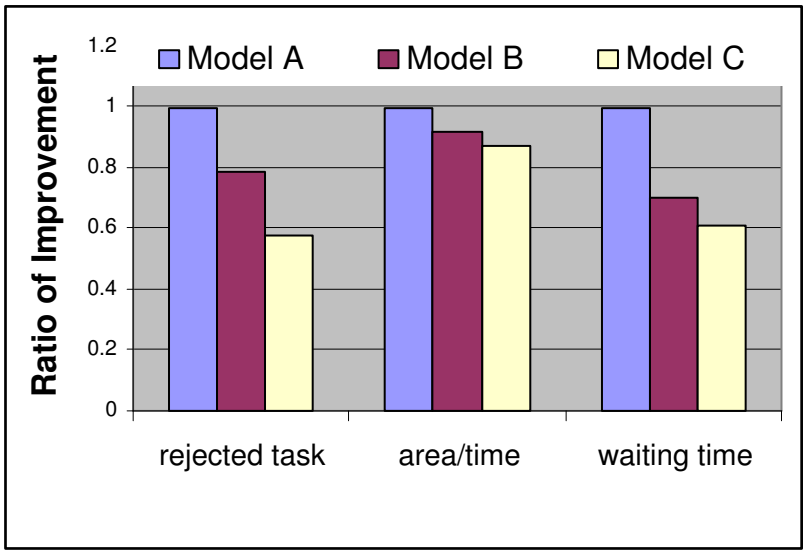


Figure 6.14: Ratios of improvement

6.5 Conclusions

Dynamically reconfigurable devices allow run-time reconfiguration to permit execution of incoming tasks or task fragments. One of the important issues in run-time reconfiguration is the fragmentation of the device area as the reconfigurable blocks are allocated and released when tasks are placed, executed and deleted. Due to those scattered, unused resources, an incoming application may not be placeable or routable. In this chapter, a cluster-based reconfigurable FPGA architecture is proposed to alleviate this difficulty. We present an assessment of the proposed architecture. We develop a fast evaluation tool to simulate on-line placement and routing effects on a run-time reconfigurable platform. The simulation results show the efficiency of the proposed architecture in relieving the fragmentation problem at the price of a modest increase in the number of switches.

Chapter 7

Summaries and Conclusions

Reconfigurable computing (RC) is going mainstream where FPGA plays an essential role. Synthesizing the application from concept and prototyping onto reconfigurable FPGAs has emerged as one of the main challenges in design automation area. A large number of new applications show the huge potentials of synthesis strategy and architecture development for FPGAs.

The work presented in this dissertation deals with the novel synthesis methodology for FPGAs. In particular, it tries to address physical aware high level synthesis (PAHLS) methodology to ensure the synthesis integrity for FPGAs. Due to its fixed hardware resources, considering the layout information at the earlier high level synthesis stage has a significant impact on the final performance of FPGA implementation. Incorporating physical information into high level synthesis for FPGAs is highly desirable. We try to attack that problem to increase the performance and reconfigurability for FPGAs or FPGA-like reconfigurable platforms. Our synthesis engine is considering both design space exploration and layout effects at the same time. Both problems were tackled as part of the formulation or of the heuristic algorithm. At the same time, the processes of high level synthesis and parts of layout synthesis are merged. We expect that this work would provide the possible techniques to extend the synthesis frontier to a more advanced level that can take both synthesis advantages of higher level specification and lower level implementation in a synergistic manner.

7.1 Our Work and Accomplishments

The first vital step in our work is to represent the design. Traditionally, a control/data flow graph (CDFG) is a commonly used internal representation to capture the input behavior of design. CDFG represents the specification of the design at a language level, rather than the final hardware level that it is trying to implement. To rapidly explore the design space, we hierarchically characterize the input design in task graph format. This task model abstracts system functionality into a set of tasks represented as nodes in a graph, and represents functional dependencies among tasks with graph edges which emphasize communication and concurrency between tasks. Edge and node labeling are used to enrich the semantics of this model. Along with an architecture template, fast resource binding and task scheduling are feasible. Further, each task node is enhanced through subgraph representation which has associated set of potential hardware implementations of that task. This incorporation technique makes strong ties to physical design early in synthesis process. Detailed descriptions of our representation techniques can be found in [122, 123, 124].

Bringing the behavioral and physical domains together allows the effects of transformations to be immediately viewable while still in the behavioral format, where the most design flexibility exists. We have developed a performance-driven PAHLS [122] where relational placement is combined with the macro generation strategy during high level synthesis. The basic idea behind relational placement is to place the components or macros that are critical at behavioral level in close physical proximity. In the proposed approach, both critical operations and possible critical nets are considered when the macros are formulated. Based on a set of benchmark designs, significant improvements in the execution frequencies and critical path delay reductions were observed [122].

An important research area for FPGAs is to exploit run-time reconfiguration. On-line synthesis is the first essential step in implementing an incoming task on FPGA during run-time configuration. However, on-line synthesis, which requires much attention on the feasibility of physical implementation during early synthesis process, received relatively little attention. In [123], we present an automated framework to integrate physical placement information into high-level synthesis that is believed to be the first on-line synthesis methodology for partially reconfigurable FPGAs. In

on-line synthesis, time for synthesis should be kept low while ensuring the placeability of the synthesized design in the available empty area on the FPGA and meeting the performance requirements. The proposed synthesizer [123] allocates the FPGA resources adaptively and is incremental in nature. The algorithm is designed to be linear in terms of the number of operations to ensure its on-line usage.

Without the adequate awareness of trade-off between different resources, it is extremely difficult for system synthesis tools to achieve high performance solutions when mapping the applications to FPGA-based computing engines. We present a transformation mechanism to extend the synthesis frontier to heterogeneous configurable architectures such that, by efficiently using the available resources, the large parallelism from data accesses and computations are extracted in a synergistic manner. We develop an automatic synthesis methodology which attacks both memory and logic assignments by interacting with behavioral synthesis. The problem is formulated as part of the heuristic algorithm by exploiting application specific information and organizing possible data and computations. We have evaluated the proposed framework on a set of DSP benchmarks and a real multimedia application by generating register-transfer level (RTL) implementations. The results show that circuits using our proposed technique achieve significant (upto, 71.8% average of 34.8%) performance improvements over the conventional design method.

Generally, regular structure facilitates predictability and simplifies the synthesis process. Further, prior to the really synthesis, the target architecture should be also characterized into an internal representation at the beginning to capture the features of possible hardware implementations. To facilitate synthesis process, we propose a cluster-based FPGA model with hybrid interconnect structure which takes advantages of both mesh and tree interconnect topologies [125]. A common observation is that most recent synthesis efforts use pre-built modules to characterize the physical traits during behavioral synthesis. By combining with this proposed regular structure and introducing module synthesis into high level synthesis, it is possible to perform physical aware high level synthesis, and to design the system more efficiently and effectively from the beginning. The evaluation results [125] demonstrate that the presented model has less switch accrued effects. Our

experimental studies show that up to 30% improvement in routing area and 50% improvement in critical path delay were achieved respectively for the large designs [125].

With the help of the elaborated representation, we then develop a fast evaluation tool [126] to simulate run-time management mechanism for on-line synthesis and layout synthesis on the reconfigurable platform. Our assessments show that, the possible physical influence, such as placed and routed effects, previously only available after layout synthesis, can now advise within the synthesis process how to manipulate the behavior to produce a high quality layout.

7.2 Contribution

In above section, we attempted to address the problem of PAHLS and presented our investigations for PAHLS by focusing on FPGA applications and their solutions. We made the following contributions.

1. Developed a representation technique which is a tight link from behavioral translation and system synthesis [122, 123, 124].

2. Developed an automatic mechanism to extract the application specific information, organize possible structure computations, and exploit the corresponding mappings by evaluating their layout implementations [123].

3. Developed a module synthesis approach by adopting a design flow where layout challenges are prioritized and attacked in advance [122].

4. Developed an synthesis approach to efficiently use both memory and logic resources by interacting with behavioral synthesis, and by extracting the large parallelism of data accesses and computations [124]

5. Validated the proposed frameworks for a set of benchmarks on the real FPGAs [122, 123, 124] or on our developed structural FPGA model [125, 126].

7.3 Conclusion

In this dissertation, we have presented our research efforts toward the integration of physical synthesis with high level synthesis. By incorporating physical considerations into high level specification, we restrict computations and communications to geographic proximities while reserve the quality of the final result to a large extent within limited resources of FPGAs. We believe that the proposed methodology provides possible directions for synthesis unification of high level abstraction and lower level implementation, and is on the right track towards achieving a well-balanced (or even a globally optimum mapping, this is the long-run objective of PAHLS) synthesis result.

Chapter 8

Directions for Future Research

High level synthesis consists of mapping a set of abstract functional models onto architecture. Efficient system design must be supported by a design flow that takes physical design into account in all its steps. In this manuscript, we study several strategies which combine the physical decisions with high level synthesis for FPGAs or FPGA-like reconfigurable architectures. The simulations demonstrate that incorporating physical design into earlier design stages is generally an effective synthesis approach since most key decisions on system organization have already been taken at that moment.

Although we have seen that it is high desirable to expect the progresses of PAHLS and efforts have been done in both industry and academia. Unfortunately, the state of the art has not yet converged to a complete methodology, and many techniques in this manuscript (both the approaches in chapter 1 and our proposed strategies) are applicable only at a single synthesis stage in the design flow and focus on only one or two of the design metrics. On the other hand, synthesis efficiency ultimately depends on detailed implementation issues, such as run-time reconfiguration. We believe that the progresses of physical aware high level synthesis will be helpful to develop the system synthesis tools for run-time reconfigurable platforms (shown in Fig.8.1). The next several sections will highlight some specific areas which may attract more research efforts.

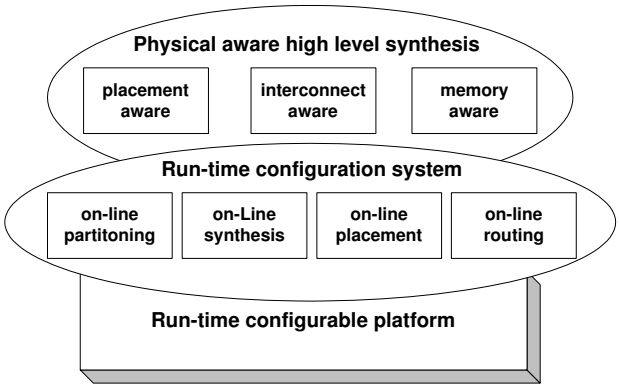


Figure 8.1: Physical aware high level synthesis and run-time synthesis

8.1 Extension of Design Representation

Bringing the behavioral and physical domains together allows the effects of transformations in either to be immediately viewable while still in the behavioral format, where the most design flexibility exists. We have developed a hierarchical representation to cater for our on-line synthesis. Such representations have been more or less countered by the development of optimization techniques which focus on the scheduling and placement algorithms. We strongly believe that in the future this representation should be associated with more elaborate layout characteristics which are indispensable for substantially improving synthesis efficiency. At the same time such representations will provide a unified framework for both HLS and physical synthesis. We hope that, with the help of this more elaborated representation, the possible placed and routed results, previously only available after layout synthesis, can now advise within the synthesis process how to manipulate the behavior to produce a high quality layout.

8.2 Interconnect Aware Synthesis

Global consideration taken at the beginning of synthesis may quickly identify a feasible implementation. Interconnect must be taken into account as early in the design process as possible. However, taking interconnect into high level synthesis is far from easy. Prior to the really synthesis, the target

architecture should be specified. The irregular structure may complex the problem and cause inaccuracy and difficulty for interconnect modeling. To the best of our knowledge, addressing routing during high level synthesis is a novel research topic, there is no report explicitly discussing on this problem.

The regular structures of our proposed FPGA model may be employed to avoid this problem and facilitate synthesis process. Remind that the developed FPGA model is cluster-based and a common observation is that most recent PAHLS efforts use pre-built modules to characterize the physical traits during behavioral synthesis. By combining with the regular interconnect structure and introducing module synthesis into high level synthesis, it is possible to perform interconnect aware high level synthesis, and to design the system more efficiently and effectively from the beginning.

8.3 On-Line Routing for Partially Reconfigurable FPGAs

As mentioned earlier, on-line synthesis, on-line placement and on-line routing are the three essential steps in implementing an incoming task on the FPGA during run-time. There are some researches on on-line placement while little work on on-line routing is reported. We have proposed an approach for the on-line synthesis. Although in our approach, we model the routing possibility by limiting the placement in the neighbor free area, we have ignored the detail implementations of routing.

The mesh interconnect employed in current commercial FPGAs has the advantage of wiring utilization, and performs well for the short range interconnections. We also see that tree networks are more attractive for the long-range interconnect. Since our proposed interconnect structure has more neighbors and encourages either row- or column-based interconnect, one possible future research is to perform on-line routing based on the proposed FPGA model. We shall see that locality information available in high level specification should be used to take advantage of model features such as neighbor direct connection or intra-cluster routing. The long connections between blocks (or tasks) may make use of tree routing. Although we know that, compared with mesh interconnect,

tree networks need more buffers to assure signal integrity and predictable delays, the additional hardware should be added into the tree networks in order to support on-line interconnection.

8.4 Incremental Synthesis

Physical design aware high level synthesis provides a quick link between high level synthesis and layout level effect. Since little changes are often taken to correct local adjustments or to make local improvements either in high level synthesis or physical accommodation, incremental algorithms for synthesis are needed to comply with those local or incremental changes. A lot of incremental algorithms independently at the high-level, logic-level, and layout level are proposed recently, and we also developed an incremental algorithm in on-line synthesis. With the rapid increase in the integration level, we strongly believe that focused participation in research and development in the area of incrementally physical aware HLS is greatly needed to make it possible to perform these tasks concurrently, and would help us manage the complexity of today s VLSI systems.

8.5 Summary

We envisage several possible directions to be exploited for PAHLS: design representation, interconnect aware synthesis, on-line routing, memory synthesis and incremental synthesis. Within this literature, we have attempted to address the problem of PAHLS and interconnect for FPGAs. Several synthesis methodologies and a hybrid interconnect FPGA model are proposed. In our studies, we formulate the problems which are limited to high-level physical design issues, focusing on the FPGA applications and their solutions. Although physical information is considered during high level synthesis, the synthesis flow does not cover the whole physical issues. For example, routing is worthy of a huge effort for investigation, and we only estimate their possibilities during the placement. Due to the limitation of available physical information in behavioral specification, integrating interconnect routing into high level synthesis is still an open problem.

In this research, we have investigate several approaches to integrate physical synthesis into high level synthesis where some physical design challenges are prioritized and solved in advance. In this chapter, we speculate the several possible research directions to be further studied separately, it should be clear that it is impossible to design well-balanced systems by focusing on one or two of these areas only, and completely disregarding the others. However, since most sub-problems such as allocation, scheduling at high level synthesis, placement and routing in physical synthesis are NP-complete or NP-hard problems, combining all those problems together and to solve it simultaneously is very difficult, or impossible under the current technologies. Ideally, we would like to tackle all design issues at the same time, and achieve a globally optimum design. This is the long-run objective of PAHLS and remains the future study.

Bibliography

[1] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1992.

[2] Min Xu and Fadi J. Kurdahi. Layout-driven rtl binding techniques for high-level synthesis using accurate estimators. *ACM Transactions on Design Automation of Electronic systems*, 2(4):312–343, 1997.

[3] Youn-Long LIN. Recent developments in high-level synthesis. *ACM Trans. On design automation of electronic syn.*, 2(1):2–21, January 1997.

[4] D.D. Gajski, N.D. Dutt, A.C. Wu, and S. Y. Lin. *High-Level VLSI Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1991.

[5] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[6] P. Michel, U. Lauther, and P. Duzy. *The synthesis Approach to Digital System Design*. Kluwer Academic, Norwell, MA, 1992.

[7] S. Devadas, A. Ghosh, and Kurt Keutzer. *Logic Synthesis*. McGraw-Hill, 1994.

[8] Naveed A. Sherwani. *Algorithms for VLSI Physical Design Automation, Third Edition*. Kluwer Academic Publishers, 1999.

[9] D. LaPotin and Y. Chen. Early matching of system requirements and package capabilities. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 394–397, November 1989.

[10] Hsiao-Pin Su, Allen C.-H. Wu, and Y. L. lin. A timing-driven soft-macro resynthesis method in interaction with chip floorplanning. In *36th Design Automation Conference Proceedings*, pages 262–267, 1999.

[11] S. Tarafdar, M. Leeser, and Zixin Yin. Integrating floorplanning in data-transfer based high-level synthesis. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 412–416, Nov. 1998.

[12] Richard J. Cloutier and Donald E. Thomas. The combination of scheduling, allocation, and mapping in a single algorithm. In *proceedings on 27th ACM/IEEE design automation conference*, pages 71 – 76, Jan. 1991.

[13] P. Kollig and B. Al-Hashimi. Simultaneous scheduling, allocation and binding in high-level synthesis. *Electronics Letters*, 33(18), August 1997.

[14] Reinaldo A. Bergamaschi. Behavioral network graph: unifying the domains of high-level and logic synthesis. In *Proceedings of conference on Design automation conference*, pages 213 – 218, Jun 1999.

[15] W. E. Dougherty and D. E. Thomas. Unifying behavioral synthesis and physical design. In *Proceedings of the 37th conference on Design automation*, pages 756 – 761, 2000.

[16] Petru Eles, Zebo Peng, Paul Pop, and Alex Doboli. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):472–491, October 2000.

[17] Apostolos A. Kountouris and Christophe Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(3):380 – 412, Jul 2002.

[18] A. Doboli. Integrated hardware-software co-synthesis for design of embedded systems under power and latency constraints. In *Proceedings of the conference on Design, automation and test in Europe*, pages 612 –619, March 2001.

[19] Ranga Vemuri and R. Radhakrishan. Sblox: A language for digital system synthesis. In *Tchnical Report No. 258/05/01/ECECS, University of Cincinnati*, 2000.

[20] Youngsoo Shin and Kiyoung Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of conference on Design automation*, pages 134 – 139, Jun 1999.

[21] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. Cosyn: hardware-software co-synthesis of embedded systems. In *Proceedings of the 34th annual conference on Design automation*, pages 703 – 708, June 1997.

[22] J.-P. Weng and A. C. Parker. 3d scheduling: High-level synthesis with floorplanning. In *Proc. Design Automation Conf*, pages 668–673, Jun 1992.

[23] H. Jang and B. Pangrle. A grid-based approach for connectivity binding with genmetric costs. In *Proceedings of ICCAD*, pages 94–99, Nov 1993.

[24] Y.-M. Fang and D. F. Wong. Simultaneous functional-unit binding and floorplanning. In *Proc. Int. Conf. Computer-Aided Design*, pages 317–321, November 1994.

[25] M. Rim, A. Mujumdar, R. Jain, and R. De Leone. Optimal and heuristic algorithms for solving the binding problem. *IEEE Transactions on VLSI Systems*, 2:211–225, June 1994.

[26] Pradeep Prabhakaran, Prithviraj Banerjee, James Crenshaw, and Majid Sarrafzadeh. Simultaneous scheduling, binding and floorplanning for interconnect power optimization. In *Proceedings of the Eleventh International Conference on VLSI Design*, pages 428–434, January 1998.

[27] Pradeep Prabhakaran and Prithviraj Banerjee. Simultaneous scheduling, binding and floorplanning in high-level synthesis. In *Proceedings of the 12th International Conference on VLSI Design*, pages 423 –427, January 1999.

[28] K. Choi and S. P. Levitan. A flexible datapath allocation method for architectural synthesis. *ACM Trans. Design Automation Electronic Systems*, 4(4):376–404, 1999.

[29] S. Cadambi and S. C. Goldstein. Cpr: A configuration profiling tool. In *Proceedings of FPGAs for Custom Computing Machines(FCCM)*, pages 104 –113, 1999.

[30] V. G. Moshnyaga, H. Onodera, and K. Tamaru. A performance-driven macro-block placer for architectural evaluation of asic designs. In *Proc. 8th Annual IEEE International ASIC Conference and Exhibit*, pages 233–236, 1995.

[31] V. G. Moshnyaga and K. Tamaru. A placement driven methodology for high-level synthesis of sub-micron asic's. *Proc. Int'l Symposium on Circuits and Systems*, 4:572–575, 1996.

[32] Daehong Kim, Jinyong Jung, sunghyun lee, jinhwan jeon, and kiyoung choi. Behavior-to-placed rtl synthesis with performance-driven placement. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 320 – 325, 2001.

[33] H. Kramer, M. Neher, G. Rietsche, and W. Rosenstiel. Data path and control synthesis in the caddy system. In *Proc. of the Intern. Workshop on Logic and Architecture Synthesis for Silicon Compilers*, May 1988.

[34] Champaka Ramachandran and Fadi J. Kurdahi. Combined topological and functionality based delay estimation using a layout-driven approach for high level applications. In *Proceedings of the conference on European Design Automation*, pages 72 – 78, November 1992.

[35] Ying Zhao and Sharad Malik. Exact memory size estimation for array computations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(5):517 – 521, Oct 2000.

[36] K.-T. Cheng and V. Agrawal. An entropy measure for the complexity of multi-output boolean functions. In *DAC*, pages 302 –305, 1990.

[37] M. Nemani and F. Najm. High-level power estimation and the area complexity of boolean function. In *Proceedings of international symposium on Low power electronics and design*, pages 329 –334, 1996.

[38] Kavel M. Buyuksahin and Farid N. Najm. High-level area estimation. In *Proceedings of international symposium on Low power electronics and design*, pages 271 – 274, Aug 2002.

[39] Alok Sharma and Rajiv Jain. Estimating architectural resources and performance for high-level synthesis applications. In *Proceedings of the 30th international on Design automation conference*, pages 355 – 360, Jul 1993.

[40] Seong Y. Ohm, Fadi J. Kurdahi, and Nikil Dutt. Comprehensive lower bound estimation from behavioral descriptions. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 182–187, Nov. 1994.

[41] Giri Tiruvuri and Moon Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(2):162 – 180, April 1998.

[42] Seong Y. Ohm, Fadi J. Kurdahi, Nikil Dutt, and Min Xu. A comprehensive estimation technique for high-level synthesis. In *Proceedings of the 8th international symposium on System synthesis*, pages 122–127, Sept. 1995.

[43] C. Ramachandran, F. J. Kurdahi, D. D. Gajski, A. C.-H. Wu, and V. Chaiyakul. Accurate layout area and delay modeling for system level design. In *Proceedings of international conference on Computer-aided design*, pages 355 – 361, Nov 1992.

[44] Kiarash Bazargan, Abhishek Ranjan, and Majid Sarrafzadeh. Fast and accurate estimation of floorplans in logic/high-level synthesis. In *Proceedings of the 10th Great Lakes Symposium on VLSI*, pages 95 – 100, March 2000.

[45] J D. S. Gelosh and D. E. Steliff. Towards modeling layout tools to derive forward estimates of area and delay at the rtl level. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):451–491, 2000.

[46] Paul Landman. High-level power estimation. In *Proceedings of international symposium on Low power electronics and design*, volume volume 8, pages 29 – 35, Aug 1996.

[47] Deming Chen, Jason Cong, and Yiping Fan. Low-power high-level synthesis for fpga architectures. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 134 – 139, August 2003.

[48] Shwar Parulkar, Sandeep K. Gupta, and Melvin A. Breuer. Estimation of bist resources during high-level synthesis. *Journal of Electronic Testing: Theory and Applications*, 13(3):221 – 237, Dec 1998.

[49] J. Ph. Diguet, D. Chillet, and O. Sentieys. A framework for high level estimations of signal processing vlsi implementations. *Journal of VLSI Signal Processing Systems*, 25(3):261 – 284, July 2000.

[50] Byoungro So, Pedro C. Diniz, and Mary W. Hall. Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In *Proceedings of the 40th conference on Design automation*, pages 514 – 519, June 2003.

[51] Srinivas Katkoori and Ranga Vemuri. Accurate resource estimation algorithms for behavioral synthesis. In *Proceedings of the Ninth Great Lakes Symposium on VLSI*, pages 338 – 339, March 1999.

[52] Carsten Menn, Oliver Bringmann, and Wolfgang Rosenstiel. Controller estimation for fpga target architectures during high-level synthesis. In *Proceedings of the 15th international symposium on System Synthesis*, pages 56 – 61, Oct. 2002.

[53] Carlos Alba-Pinto, Bart Mesman, and Jochen Jess. Constraint satisfaction for relative location assignment and scheduling. In *Proceedings of IEEE/ACM international conference on Computer-aided design*, pages 384 – 390, Nov. 2001.

[54] Pradip K. Jha and Nikil D. Dutt. High-level library mapping for memories. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):566–603, Jul 2000.

[55] Chao Huang, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. High-level synthesis of distributed logic-memory architectures. In *Proceedings of international conference on Computer-aided design*, pages 564–571, Nov 2002.

[56] K. S. Khouri, G. Lakshminarayana, and N. K. Jha. Memory binding for performance optimization of control-flow intensive behaviors. In *Proceedings of international conference on Computer-aided design*, pages 482 – 488, Nov 1999.

[57] J. L. da Silva, F. Catthoor, D. Verkest, and H. De Man. Power exploration for dynamic data types through virtual memory management refinement. In *Proceedings of international symposium on Low power electronics and design*, pages 311 – 316, August 1998.

[58] L. Semeria, K. Sato, and G. De Micheli. Synthesis of hardware models in c with pointers and complex data structures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):743 – 756, December 2001.

[59] Jaewon Seo, Taewhan Kim, and Preeti R. Panda. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *Proceedings of the 39th conference on Design automation*, pages 608 – 611, Jun 2002.

[60] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting off-chip memory access modes in high-level synthesis. In *Proceedings of international conference on Computer-aided design*, pages 333 – 340, Nov 1997.

[61] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Incorporating dram access modes into high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(2):96 –109, Feb 1998.

[62] Preeti Ranjan Panda and Nikil D. Dutt. Behavioral array mapping into multiport memories targeting low power. In *Proceedings of the Tenth International Conference on VLSI Design*, pages 268–272, Jan 1997.

[63] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, July 2000.

[64] L. Benini, A. Macii, and M. Poncino. *Memory Design Techniques for low-Energy Embedded Systems*. Kluwer, Dordrecht, 2002.

[65] D. Robinson and P. Lysaght. Methods of exploiting simulation technology for simulating the timing of dynamically reconfigurable logic. *IEEE Proceedings Computers and Digital Techniques*, 147(3):175–180, May 2000.

[66] Ian Robertson, James Irvine, Patrick Lysaght, and David Robinson. Timing verification of dynamically reconfigurable logic for the xilinx virtex fpga series. In *Proceedings of ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 127 – 135, Feb. 2002.

[67] Kia Bazargan, Seda Ogrenci, and Majid Sarrafzadeh. Integrating scheduling and physical design into a coherent compilation cycle for reconfigurable computing architectures. In *Proceedings of the 38th conference on Design automation*, pages 635–640, Jun 2001.

[68] Mukund Sivaraman and Shail Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pages 35–42, Oct. 2002.

[69] Jinhwan Jeon, Daehong Kim, Dongwan Shin, and Kiyoung Choi. High-level synthesis under multi-cycle interconnect delay. In *Proceedings of the conference on Asia South Pacific Design Automation Conference*, pages 662–667, Nov 2001.

[70] Jason Cong, Yiping Fan, Xun Yang, and Zhiru Zhang. Architecture and synthesis for multi-cycle communication. In *Proceedings of international symposium on Physical design*, pages 190 – 196, April 2003.

[71] Sungpack Hong and Taewhan Kim. Bus optimization for low-power data path synthesis based on network flow method. In *Proceedings of international conference on Computer-aided design*, pages 312 – 317, Nov 2000.

[72] Lin Zhong and Niraj K. Jha. Interconnect-aware high-level synthesis for low power. In *Proceedings of IEEE/ACM international conference on Computer-aided design*, pages 110–117, Nov. 2002.

[73] S. Sundararaman, S. Govindarajan, and R. Vemuri. Application specific macro based synthesis. In *Fourteenth International Conference on VLSI Design*, pages 317–324, 2001.

[74] P.G. Paulin and J.P. Knight. Force directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on CAD*, volume 8:661–679, Jun 1989.

[75] Xilinx Inc. *virtex-II Pro Platform FPGA Handbook and ISE Manual*.

[76] S. Guccione, D. Verkest, and I. Bolsens. Design technology for networked reconfigurable fpga platforms. In *Conf. of Design, Automation and Test in Europe (DATE)*, pages 994–997, Mar. 2002.

[77] Paul J.M. Havinga, Lodewijk T. Smit, Gerard J.M. Smit, Martinus Bos, and Paul M. Heysters. Energy management for dynamically reconfigurable heterogeneous mobile systems. In *10th Heterogeneous Computing Workshop*, April 2001.

[78] Gerard J.M. Smit, Paul J.M. Havinga, Lodewijk T. Smit, Paul M. Heysters, and Michel A.J. Rosien. Dynamic reconfiguration in mobile systems. In *International conference of field programmable logic*, pages 171–181, sept. 2002.

[79] Gordon Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Proceedings of the 6th International Workshop on Field Programmable Logic and Applications*, pages 327–336, 1996.

[80] G. Wigley and D. Kearney. The management of applications for reconfigurable computing using an operating system. In *Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, volume 6, pages 73–81, 2002.

[81] Grant Wigley and David Kearney. The first real operating system for reconfigurable computers. In *Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 130–137. IEEE Computer Society, 2001.

[82] Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems . *IEEE Design and Test - Special Issue on Reconfigurable Computing*, 17(1):68–83, Jan.-Mar. 2000.

[83] Herbert Walder, Christoph Steiger, and Marco Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing . In *International Parallel and Distributed Processing Symposium (IPDPS'03)* , page 178, April 2003.

[84] *Forge, http://www.xilinx.com/xlnx/xebiz/designResources/.*

[85] Min Xu and Fadi J. Kurdahi. Layout-driven high level synthesis for fpga based architectures. *Proceedings of the conference on Design, automation and test in Europe*, pages 446–450, 1998.

[86] R. Camposano. Path-based scheduling for synthesis. *IEEE Transactions on Computer-Aided Design*, 10(1):85 – 93, 1991.

[87] Jianwen Zhu and Daniel D. Gajski. Soft scheduling in high level synthesis. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 219 – 224, June 1999.

[88] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh. A super-scheduler for embedded reconfigurable systems. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 391 – 394, Nov 2001.

[89] N. Dutt and C. Ramachandran. Benchmarks for the 1992 high level synthesis workshop. *Tech. Rep. 92-107, Dep. Inform. Comput. Sci., Univ. California, Irvin*, 1992.

[90] K. Bazargan and M. Sarrafzadeh. Fast online placement for reconfigurable computing systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 300–302, April 1999.

[91] Manish Handa and Ranga Vemuri. An efficient algorithm for finding empty space for online fpga placement. In *ACM/IEEE 41st DAC*, pages 960–965, June 2004.

[92] J. Nestor and D.E Thomas. Behavioral synthesis with interfaces. In *Proceedings of the IEEE Conference on Computer Aided Design*, Nov 1986.

[93] Saurabh N. Adya and Igor L. Markov. Fixed-outline floorplanning through better local search. In *IEEE Int. Conf. on Computer Design*, pages 328–334, September 2001.

[94] A. Agarwal, D. A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel and Distributed Systems*, 6(9):943962, 1995.

[95] H. Schmit and D. Thomas. Synthesis of applications specific memory designs. *IEEE Transactions on VLSI Systems*, 5(1):101–111, 1997.

[96] Joonseok Park and Pedro C. Diniz. Synthesis of pipelined memory access controllers for streamed data applications on fpga-based computing engines. In *the 14th international symposium on Systems synthesis*, pages 221 – 226, Oct. 2001.

[97] I. Ouaiss and R. Vemuri. Hierarchical memory mapping during synthesis in fpga-based reconfigurable computers. In *the conference on Design, automation and test in Europe*, pages 650 – 657, March 2001.

[98] M. A. Turk and A. P. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71 – 96, 1991.

[99] Bruno T. Messmer. *Efficient Graph Matching Algorithms for PreProcessed Model Graphs*. PhD thesis, University of Berne, Nov 1995.

[100] N. Narasimhan. *Formal-Asssertions Based Verification in a High-Level Synthesis System*. PhD thesis, University of Cincinnati, 1998.

[101] Stephen M. Trimberger. *Field-Programmable Gate Array Technology*. Kluwer Academic Publishers, Norwell, MA, 1994.

[102] A. DeHon. *Reconfigurable Architecture for General Purpose Computing*. PhD thesis, MIT, 1996.

[103] Mike Sheng and Jonathan Rose. Mixing buffers and pass transistors in fpga routing architectures. In *International Symposium on Field Programmable Gate Arrays*, pages 75 – 84, Feb. 2001.

[104] A. Agarwal and D. Lewis. Routing architectures for hierarchical field programmable gate arrays. In *International Conference on Computer Design*, pages 475–478, 1994.

[105] Yen-Tai Lai and Ping-Tsung Wang. Hierarchical interconnection structures for field programmable gate arrays. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(2):186–196, 1997.

[106] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. Hsra: High-speed, hierarchical synchronous reconfigurable array. In *International Symposium on Field Programmable Gate Arrays*, pages 125–134, Feb. 1999.

[107] Raphael Rubin and Andr DeHon. Design of fpga interconnect for multilevel metallization. In *11th international symposium on Field programmable gate arrays*, pages 154 – 163, 2003.

[108] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field-programmable gate arrays*. Kluwer Academic Publishers, Norwell, MA, 1992.

[109] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100 In *seventh international symposium on Field programmable gate arrays*, pages 69–78, 1999.

[110] F. T. Leighton. New lower bound techniques for vlsi. In *IEEE 22-nd Annual Symposium on ther foundations of Computer Science*, 1981.

[111] P. Christie and D. Stroobandt. The interpretation and applicaiton of rent's rule. *IEEE trans. on VLSI*, 8(6).

[112] *http://www.eecg.toronto.edu/ ṽaughn/vpr/vpr.html*, 2003.

[113] L. McMurchie and C. Ebling. Pathfinder: A negotiation based performance-driven router for fpgas. In *International Symposium on Field Programmable Gate Arrays*, pages 111–117, Feb. 1995.

[114] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, 1999.

[115] Reiner Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array. In *conference on Asia South Pacific design automation*, pages 163–168, Jan 2000.

[116] Takashi Miyamori and Kunle Olukotun. Remarc: Reconfigurable multimedia array coprocessor. In *ACM FPGA*, page 261, Feb 1998.

[117] Hartej Singh, Ming-Hau Lee, G. Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. Chaves Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 49(5):465–481, 2000.

[118] C. Ebeling, D. Cronquist, and P. Franklin. Rapid: Reconfigurable pipelined datapath. In *FPL, LNCS 1142, Springer Verlag*, pages 126–135, Sept 1996.

[119] A. Marshall, T. Stansfield, I. Kostanrnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *ACM/SIGDA FPGA*, pages 135–143, Feb 1999.

[120] X.Tang, M. Aalsma, and R. Jou. A compiler directed approach to hiding configuration loading latency in chameleon reconfigurable chips. In *FPL, LNCS, Springer-Verlag*, pages 29–38, Aug 2000.

[121] Xilinx, XAPP 151(v1.5). *Virtex Series Configuration Architecture User Guide*, sept 2000.

[122] Renqiu Huang and Ranga Vemuri. Forward-looking macro generation and relational placement during high level synthesis to fpgas. In *IEEE International Parallel and Distributed Processing Symposium*, April 2004.

[123] Renqiu Huang and Ranga Vemuri. On-line synthesis for partially reconfigurable fpgas. In *IEEE 18th international conference on VLSI Design*, pages 663–338, Jan 2005.

[124] Renqiu Huang and Ranga Vemuri. Transformation synthesis for data intensive applications to fpgas. In *ACM Great Lakes Symposium on VLSI, accepted to appear*, April 2006.

[125] Renqiu Huang and Ranga Vemuri. Analysis and evaluation of a hybrid interconnect structure for fpgas. In *IEEE/ACM International Conference on Computer Aided Design*, pages 595–601, Nov 2004.

[126] Renqiu Huang, Manish Handa, and Ranga Vemuri. Analysis of a hybrid interconnect architecture for dynamically reconfigurable fpgas. In *International conference on Field-Programmable Logic and its Applications*, pages 900–905, Aug 2004.