Throughput Constrained and Area Optimized Dataflow Synthesis for FPGAs

Hua Sun
Brigham Young University - Provo

THROUGHPUT CONSTRAINED AND AREA OPTIMIZED DATAFLOW SYNTHESIS FOR FPGAS

by

Hua Sun

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

Brigham Young University

April 2008

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Hua Sun

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Michael J. Wirthlin, Chair
Date	Brent E. Nelson
Date	James K. Archibald
Date	Doran K. Wilde
Date	Clark N. Taylor

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Hua Sun in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Michael J. Wirthlin Chair, Graduate Committee

Accepted for the Department

Michael A. Jensen
Chair

Accepted for the College

Alan R. Parkinson
Dean, Ira A. Fulton College of
Engineering and Technology

ABSTRACT

THROUGHPUT CONSTRAINED AND AREA OPTIMIZED DATAFLOW SYNTHESIS FOR FPGAS

Hua Sun

Department of Electrical and Computer Engineering

Doctor of Philosophy

Although high-level synthesis has been researched for many years, synthesizing minimum hardware implementations under a throughput constraint for computationally intensive algorithms remains a challenge. In this thesis, three important techniques are studied carefully and applied in an integrated way to meet this challenging synthesis requirement. The first is pipeline scheduling, which generates a pipelined schedule that meets the throughput requirement. The second is module selection, which decides the most appropriate circuit module for each operation. The third is resource sharing, which reuses a circuit module by sharing it between multiple operations. This work shows that combining module selection and resource sharing while performing pipeline scheduling can significantly reduce the hardware area, by either using slower, more area-efficient circuit modules or by time-multiplexing faster, larger circuit modules, while meeting the throughput constraint. The results of this work show that the combined approach can generate on average 43% smaller hardware than possible when a single technique (resource sharing or module selection) is applied.

There are four major contributions of this work. First, given a fixed throughput constraint, it explores *all* feasible frequency and data introduction interval design points that meet this throughput constraint. This enlarged pipelining design space exploration results in superior hardware architectures than previous pipeline synthesis work because of the larger sapce. Second, the module selection algorithm in this work considers different module architectures, as well as different pipelining options for each architecture. This not only addresses the unique architecture of most FPGA circuit modules, it also performs retiming at the high-level synthesis level. Third, this work proposes a novel approach that integrates the three inter-related synthesis techniques of pipeline scheduling, module selection and resource sharing. To the author's best knowledge, this is the first attempt to do this. The integrated approach is able to identify more efficient hardware implementations than when only one or two of the three techniques are applied. Fourth, this work proposes and implements several algorithms that explore the combined pipeline scheduling, module selection and resource sharing design space, and identifies the most efficient hardware architecture under the synthesis constraint. These algorithms explore the combined design space in different ways which represents the trade off between algorithm execution time and the size of the explored design space.

ACKNOWLEDGMENTS

First of all, I want to express my thanks and great appreciation to my advisor Michael Wirthlin. His rare combination of strengths in both the theoretical and the practical has been a continuous inspiration to me, as well as his genuine enthusiasm for the field.

I am also deeply indebted to my committee members, Professors Brent Nelson, James Archibald, Doran Wilde and Clark Taylor. Their comments and welcoming discussions have been an enjoyable part of this work.

I would like to thank my readers and others whose discussions with me have had a large, direct impact on what is presented here. In particular, I want to thank Stephen Neuendorffer from Xilinx research labs, for sharing with me his wide knowledge of issues in design space exploration, and for his helpful comments.

I truely want to thank the many people who have contributed to this project. They have done so with caring sincerity, and their hard work deserves as much recognition as possible. These are Nathan Rollins, Bennion, Eric Johnson and all students who took the high level synthesis class during Fall 2005.

I am so very grateful for my family, and in particular for my parents, who have taught me and influenced me far more deeply than they realize. Their love and support has been continuously uplifting during many struggles.

For my wife, words cannot express my love, appreciation and respect for her. She has given her love and support freely and far beyond my hopes and dreams. Her mark is on every page of my work.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

FPGAs (Field Programmable Gate Arrays) are becoming an ever popular hardware platform for implementing computationally intensive algorithms. Over the years, a large number of such algorithms have been implemented on FPGAs for image and video processing, data encryption and decryption, digital communication systems, etc. For example, Porter et al. [3] implemented a maximal throughput neighborhood image processing algorithm on Stratix II [4] and Virtex-II [5] devices. Järvinen et al. [6] implemented a fully pipelined encryptor based on the Advanced Encryption Standard encryption algorithm with 128-bit input and key length (AES-128) on Virtex-E [7] and Virtex-II devices. Singaraju et al. [8] implemented an FPGA based signature match processor that can serve as the core of a hardware based network intrusion detection system.

Compared with other technologies, FPGAs offer several major advantages for computationally intensive algorithms. First, FPGAs have a performance advantage over sequential processors such as general purpose CPUs and DSPs (Digital Signal Processors). Although their performance cannot match fully customized circuits such as ASICs (Application Specific Integrated Circuits) and ASSPs (Application Specific Standard Product), they are capable of implementing most computationally intensive algorithms. Second, FPGAs have much lower up-front, non-recurring expenses (NREs). The typical NREs for taping-out a 90nm ASIC are tens of millions of US dollars. On the other hand, FPGAs are pre-manufactured and can be purchased off-the-shelf with tens to thousands of dollars, depending on the capacity of the chip. Third, FPGAs can be re-programmed after manufacturing which is necessary for algorithms implementation, since this allows late-stage design changes with little impact on the overall project schedule. This feature can also be used to fix the bugs in the design even after the system is deployed in the field. As Figure 1.1 shows, FPGAs offer a good balance between performance and programmability. Their performance is close to customized hardware such as ASICs and ASSPs, yet at the same time they can be re-programmed like general purpose CPUs and DSPs.

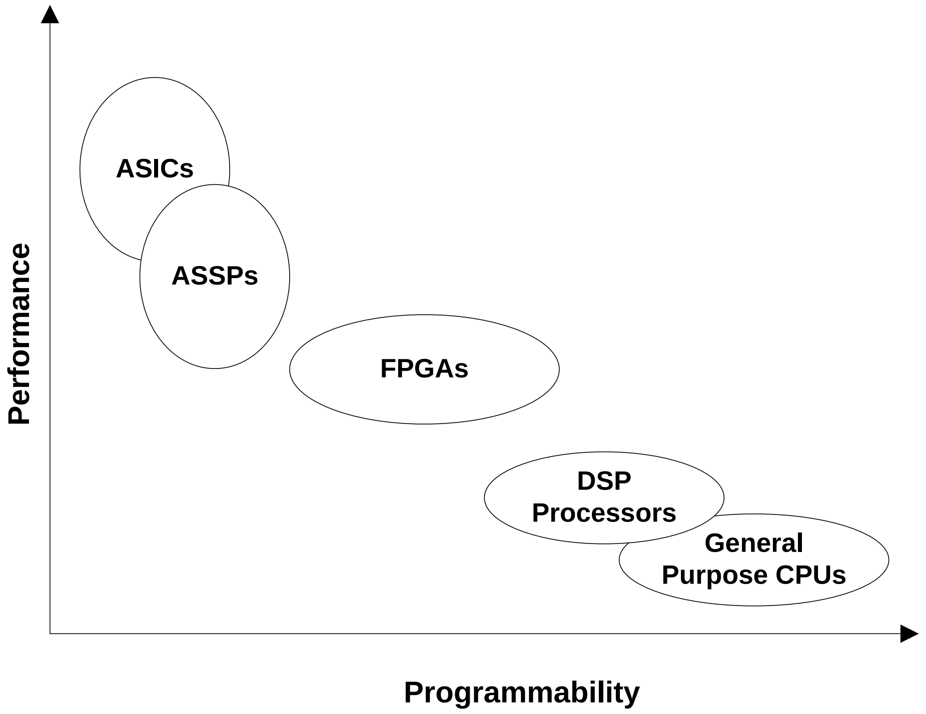


Figure 1.1: Performance and programmability comparison between FPGAs, ASICs, ASSPs, DSPs and general purpose CPUs [1]

There is one important disadvantage of using FPGAs for computationally intensive algorithms. Compared to microprocessors, FPGAs are significantly more difficult to program. Programing for microprocessors is *imperative*. The designers only need to specify a sequence of functions to perform on certain data objects. The compiler will automatically translate this specification into a list of instructions which can be executed by the targeted microprocessor. To program FPGAs, the designer has to create a customized circuit. All circuit details such as the hardware implementation of each operation, the behavior of the circuit at each clock cycle, and the interconnection and synchronization inside the circuit, must be specified by the designer.

Currently the most mature method for FPGA based design is RTL (Register Transfer Level) synthesis. The input to RTL synthesis contains two parts. The first part is a data-path structure which contains functional units (e.g., adders, multipliers and shifters, etc.), storage units (e.g., registers and memory), and interconnection units (e.g., buses and multiplexers). The second part is a controller (finite state machine) which contains the detailed schedule (related to clock edge) of the data-path components. The output of RTL synthesis is a technology dependent implementation in terms of logic gates and their interconnections. The RTL synthesis output is then converted by FPGA vendor specific tools to program the target FPGAs.

RTL synthesis is limited due to its design complexity. RTL synthesis requires the designer to specify the detailed circuit micro-architecture as the design input. Unfortunately, defining such an architecture, creating, simulating, implementing and debugging the corresponding RTL code is very time consuming and error-prone, especially with the increasing complexity of the designs. Moreover, to optimize the hardware for area and timing purpose, the designer has to create and evaluate many different RTL designs. Manually changing from one micro-architecture to another one for design optimization can be prohibitively difficult due to the long design cycle.

The limitation of RTL synthesis creates a huge design productivity gap when implementing computationally intensive algorithms on FPGAs. Most computationally intensive algorithms are first written in software programming languages such as C or Fortran. These software programs are then *manually* translated into RTL descriptions for RTL synthesis. However, this manual translation process is tedious and error prone. The designers who translate must fully understand the software algorithm, determine the micro-architecture of the hardware implementation, verify the correctness of the micro-architecture, and synthesize the hardware that meets the area and timing requirements. If the micro-architecture requires modification due to changes in source algorithms or area and timing requirements, this manual translation process has to be repeated. With the growing complexity of these algorithms, the ever shortening time-to-market requirement and the constantly changing requirements, this manual translation process is creating a design productivity gap as shown in Figure 1.2. Thus, new synthesis techniques must be made available to program FPGAs for computationally intensive algorithms with significant improvements in productivity.

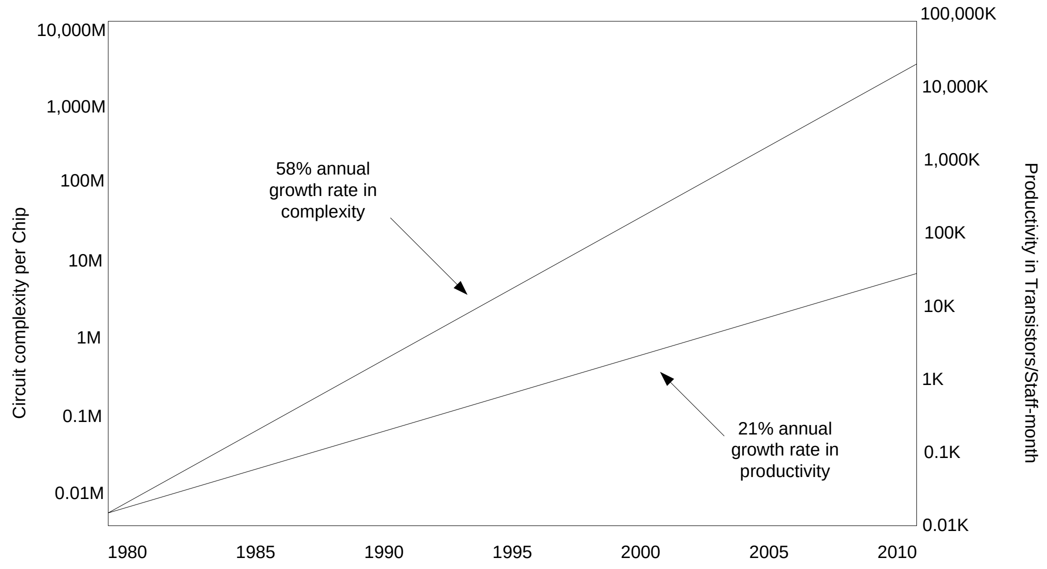


Figure 1.2: Current design methodologies and productivity improvements are failing to keep pace with the rapid and ongoing increase in circuit complexity [2]

1.1 High Level Synthesis for FPGAs

High-level synthesis (HLS) has been proposed to reduce this design productivity gap and address the limitations of RTL synthesis. HLS normally takes an *untimed* algorithm description such as a C program or a block diagram, and automatically generates different RTL descriptions based on different design requirements [9]. An untimed description is a more abstract representation of the algorithm than the RTL description because it is architecture independent and contains no implementation details, thus no timing specification or constraint. It is the responsibility of HLS tool to automatically explore and identify the best architecture and implementation for this algorithm based on various user constraints.

High-level synthesis normally performs scheduling, binding and control synthesis to translate a high-level description into an RTL description [10]. The scheduling problem is to determine the time step or clock cycle in which each operation in the design executes. The purpose of binding is to determine the number of resources that need to be allocated to synthesize the hardware circuit, as well as the mapping from the operations, variables, and data (and control) transfers in the design to the allocated resources. Control synthesis generates a control unit that implements the schedule. This control unit generates control signals that control the flow of data through the data path.

High-level synthesis offers three major benefits compared to RTL synthesis. First, modeling at higher levels of abstraction makes the designs smaller in code size and less complex than equivalent RTL descriptions. The designs are easier to write, understand and debug because they are closer to the algorithms being developed. Second, HLS provides an automatic and much faster way to implement untimed software algorithms in hardware. This greatly improves the design team's productivity and helps to close the design productivity gap. Third and most importantly, an HLS design methodology shifts the focus from the detailed architecture to the behavior of the design. A high-level description defines the algorithm or behavior to be performed with few or no architectural details. With HLS tools, the designer can direct the synthesis tool to generate alternate architectures by modifying constraints (such as clock speed, hardware resource constraint, and latency, etc.). The output of HLS tools is RTL code that implements the behavior of the high-level description with the best hardware architecture. Thus, designers can explore more architectural alternatives than possible with a RTL synthesis methodology.

The unique "reconfigurable" feature of FPGAs make FPGA-specific HLS even more attractive. The first benefit is accelerated verification. One big limitation of HLS tools for ASICs is the long verification period of the synthesized design. Verification for ASIC design can only be done through software-based simulation, because

ASICs cannot be manufactured until the design is fully verified. However, verification for FPGA design can be performed directly on the hardware itself. Hardware-based verification is often orders of magnitude faster than software-based verification. The second benefit is faster design space exploration. High-level synthesis can generate multiple implementations from a single design input varying in clock speed, area, latency and power consumption, etc. Validating these design metrics estimated by HLS tools can be very difficult and time consuming for ASIC designs. However, FPGA-based design can validate these estimations by downloading the synthesized implementation to the actual hardware instantly. This fast prototyping greatly improves the design space exploration speed and quality of high-level synthesis tools.

Despite the enormous previous research in high-level synthesis [11, 12, 13, 14, 15], there are still some major limitations of current HLS tools. The first limitation is that few HLS research are throughput constrained and area optimized. Most of them are area constrained and optimized for clock frequency or latency and others are frequency or latency constrained and optimized for area. However, throughput constrained synthesis is crucial for synthesizing streaming algorithms. The second limitation is that the synthesis quality from HLS is still not satisfactory. Although numerous researchers on high-level synthesis have proposed various ways for the automatic translation task, few of them focused on the synthesized hardware quality. The main reason for unsatisfactory synthesis quality is the small design space previous HLS work explored.

The third limitation of the previous HLS work is that most of them are focused on ASIC technology only and do not address the unique features of FPGAs. FPGAs are normally 3 to 4 times slower than ASICs due to their slow logic and interconnections. On the other hand, FPGAs have a plethora of registers which makes pipelining a very common technique for FPGA based designs. Thus, FPGA-specific HLS must support automatically generating pipelined implementations. Although there are some HLS work which perform pipeline synthesis, their scope is very limited. Another unique feature of FPGAs is their significantly higher silicon cost related-to ASICs, and the area costs of some FPGA components are quite different from ASIC technology. For example, the multiplexer in FPGAs is as expensive as an adder while it is much cheaper in ASIC technology. Thus FPGA-specific high-level synthesis tools must minimize the area cost of the generated implementation, and address the unique component area cost.

1.2 Proposed FPGA HLS Methodology

To address the limitations of current HLS tools, this work proposes an FPGA-specific HLS methodology which synthesizes throughput constrained, minimum hardware implementations from untimed computationally intensive algorithms. Most computationally intensive streaming algorithms require a unique synthesis constraint called throughput. The throughput constraint puts a lower bound on the performance of the synthesized hardware implementation (i.e. how many data samples the synthesized circuit can process every second). Once the throughput constraint is met, minimizing the hardware area cost is very important for FPGAs due to its high silicon cost.

The HLS methodology proposed in this work is based on three techniques: pipeline scheduling, resource sharing and module selection. Pipeline scheduling determines the exact start time of each operation in an untimed computationally intensive algorithm, and guarantees that the schedule meets the throughput constraint. Module selection decides the most appropriate hardware component for each operation in the algorithm. Resource sharing reuses a hardware component by sharing it between more than one operation. Module selection and resource sharing are very important for minimizing the hardware area of the synthesized circuit.

The primary focus of this work is the integration of these three techniques. Most previous work only combine two of the three techniques, assuming the third one is performed independently. An important claim of this work is that combining the three techniques together can greatly expand the design space. A larger design space almost always identifies superior solutions that couldn't be found by existing techniques.

However, the design space created by combining the three techniques is very difficult to explore due to the close inter-relationship between them and the magnitude of the combined design space. A major contribution of this work is that it thoroughly studied each technique as well as their inter-relationships, and proposed one novel algorithm that concurrently explores these techniques in an efficient way. To the author's best knowledge, this is the first attempt to concurrently explore these three techniques.

1.3 Dissertation Structure

The rest of this dissertation focuses on the detailed discussion of the three synthesis techniques and proposes algorithms that integrate them. In Chapter 2, a detailed overview of the proposed HLS methodology is first presented. This includes the synthesis constraints and optimization goals, design representation and transformation steps. Chapter 2 then provides detailed technical background for scheduling, resource sharing and module selection respectively. Finally, it discusses the combined design space of the three techniques and the complexity to explore it.

Chapter 3 discusses the first important technique of the proposed HLS methodology: pipeline scheduling. It begins with a detailed discussion of pipelining which is a very common design technique for FPGA based designs. Then several important issues of pipeline scheduling are presented. Finally a new, hardware oriented pipeline scheduling algorithm called *HOIMS* is proposed. This algorithm also serves as the backbone of the final pipeline synthesis algorithm, and will be expanded by the integration of the other synthesis techniques.

Chapter 4 studies another important HLS synthesis technique: resource sharing. This chapter first carefully characterizes the area and timing cost of resource sharing overhead components (such as multiplexers) in FPGA architectures. It then analyzes the close inter-relationship between resource sharing and pipeline scheduling. Finally, this chapter proposes a weighted compatibility graph based resource sharing algorithm and integrates it to the pipeline scheduling algorithm proposed in Chapter 3.

Chapter 5 discusses the third important HLS synthesis technique which has limited previous research: module selection. Module selection can be applied in HLS to reduce the hardware area by choosing the appropriate hardware component for each operation in the computationally intensive algorithms. This chapter proposes a novel way to integrate module selection with pipeline scheduling and resource sharing based on a careful study of the inter-relationship between them.

Chapter 6 presents and discusses the experimental results of several pipeline synthesis algorithms which combine pipeline scheduling, resource sharing and module selection. An estimated design space of different synthesis techniques is first presented. The actual design space and the benefits of a combined approach is then presented and discussed with the results of two simpler algorithms: the *ASAP Exploration* algorithm and the *IMS Exploration* algorithm. Finally, the results of the *HOIMS* algorithm are illustrated, analyzed, and compared with the other two algorithms.

Chapter 7 summarizes the work accomplished in this dissertation. It also proposes future directions beyond this work. Several Appendix chapters provide some valuable discussion and supplemental information can be referenced when reading the main chapters.

Chapter 2

Overview

This chapter provides an overview of the proposed HLS methodology and discusses the related technical background. It first discusses the synthesis constraint and optimization goal of the proposed methodology. Then the design representation is discussed. After a brief description of high-level synthesis transformations, it discusses scheduling, module selection, and resource sharing. Finally, the combined design space of the three techniques and exploration complexity are discussed.

2.1 Synthesis Constraint and Optimization Goal

Throughput is one of the most important constraints to synthesize computationally intensive streaming algorithms. These algorithms are normally used in systems whose data inputs are fed into the system at a fixed rate. For example, digital video encoders need to process a certain number of pixels every second; digital communication systems performance is typically measured in terms of transmitted/received symbols per second; encryption and decryption systems care about how many bytes can be encrypted or decrypted per second. Thus the main constraint for synthesizing these computationally intensive algorithms is the number of samples (pixels, symbols and bytes, etc.) the algorithm can process per second, or throughput. For this work, any synthesized implementation must meet a user-defined throughput constraint.

Once the throughput constraint is met, the synthesis algorithm should minimize the hardware area cost of the FPGA implementation. The main reason for this is that the silicon cost of FPGAs is much higher than customized circuits such as ASICs. FPGAs require 39 times more silicon area than ASICs on average to implement the same logic function [16]. Hence, generating the smallest hardware is very important for FPGA-targeted synthesis. On the other hand, there is no benefit to synthesize a hardware implementation that has a higher throughput than needed. Throughput is normally proportional to hardware area cost. Thus, extra and wasted throughput means extra and wasted hardware area. The HLS methodology proposed in this work is a throughput constraint minimum area high-level synthesis approach targeting FPGA architectures.

There has been little previous work on throughput constrained and area optimized high-level synthesis. Most ASIC targeted HLS research are area constrained and optimized for clock frequency or latency, others are frequency or latency constrained and optimized for area [10]. There are also few throughput constrained and area optimized HLS research for FPGAs. Xu and Kurdahi[17] use a layout-driven approach to synthesize an RTL netlist with predictable metrics for FPGA based architectures. In [18], an area and delay estimator for FPGAs is presented, which estimates the maximum number of CLBs consumed by the hardware synthesized from an input MATLAB algorithm, and the delay in the logic elements in the critical path and the delay in the interconnects. Other work focused on minimizing the power consumption at a higher level design abstraction. In [19], an RTL power estimator for FPGAs with consideration of wire length is presented, together with a high level synthesis system that uses the power estimator. Although there has been some research on throughput constrained and area optimized high-level synthesis, the results of this dissertation show that integrating pipeline scheduling, module selection and resource sharing provides a better solution for this challenge.

2.2 Design Representation

Although textual languages are currently the most common form of design representation for HLS tools [20, 21, 22], graphical descriptions can contain all the relevant information necessary for high-level synthesis as well. Some HLS tools use graphical languages such as Carleton's HAL System [15] and UT Austin's DAGER System [23]. For computationally intensive algorithms, a graphical representation is more intuitive because of the reduced control information in these algorithms, and the more explicit data dependency in the graphical representation.

The design representation used in this work is an SDF (Synchronous Data Flow [24]) model. SDF is a model of computation [25] well suited to represent computationally intensive streaming algorithms. It is made up of the computation elements and the relationships between these elements. This is a purely functional model of the original algorithm because it contains no explicit timing or architecture information. An SDF graph can be hierarchical to better support design abstraction of large algorithms. The Ptolemy [26] tool from UC Berkeley provides a good modeling and simulation environment for SDF models, which is used as the front-end for the proposed HLS methodology.

An SDF model is represented in the form of a directed graph (i.e. a graph with directed edges). The nodes of the graph are also called *actors*, which represent the computation elements. The edges of the graph are also called *arcs*, and *tokens* are produced by source actors and consumed by sink actors, which represent the communication between actors. An SDF graph also specifies the relative rates of production and consumption of data tokens for each *firing* of each actor. For simplicity, this work only synthesizes circuits from SDF graphs which have the same token rate for production and consumption (i.e. *homogeneous* SDF). Figure 2.1 illustrates the SDF model of a Biquad filter from the Ptolemy modeling environment.

This work uses a *Dependence Graph (DG)* with attributes as its internal synthesis representation. It is similar to a flattened SDF graph without hierarchy. Specific attributes are attached to the nodes, edges and the graph. These attributes are important for the architectural exploration process. The dependence graph is generated from the SDF model, which will be discussed in Chapter 3.

2.3 Scheduling

Scheduling is the process of assigning a start time to each operation in the dependence graph. The original dependence graph which specifies only the dependencies among operations is called an *unscheduled DG*. After scheduling is performed on the

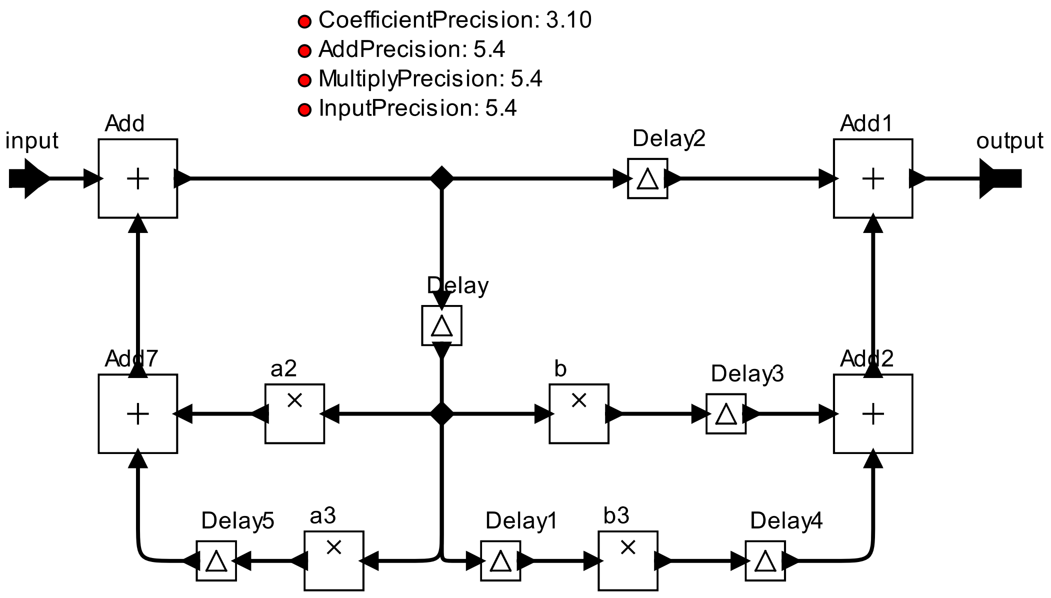


Figure 2.1: The SDF model of a Biquad filter

unscheduled DG, each operation is assigned to a start time, and the dependence graph is called a *scheduled DG*. Figure 2.2 shows an example of an unscheduled DG and the respective scheduled DG after being scheduled. As shown in Figure 2.2, every operation in the DG is assigned to a start time, while the dependencies between operations are maintained. A formal definition of scheduling for an unscheduled $DG = (V, E)$ is:

Definition 1 *Given a set of operations V with integer latency Λ and a partial order on the operations E, find an integer labeling of the operations $\varphi : V \to Z^+$ such that $t_j = \varphi(v_j), t_j \geq t_i + \lambda_i : (v_i, v_j) \in E$ [10].*

As Definition 1 shows, the start time of each operation must satisfy the original dependencies between related operations in the dependence graph. In other words, an operation cannot start until all its predecessors are finished. In definition 1, λ_i is the latency of operation i, which is defined as the number of clock cycles from the consumption of inputs to the generation of outputs for that operation.

Scheduling has a great impact on the performance as well as the area of the synthesized hardware. For each dependence graph, there is great flexibility in the

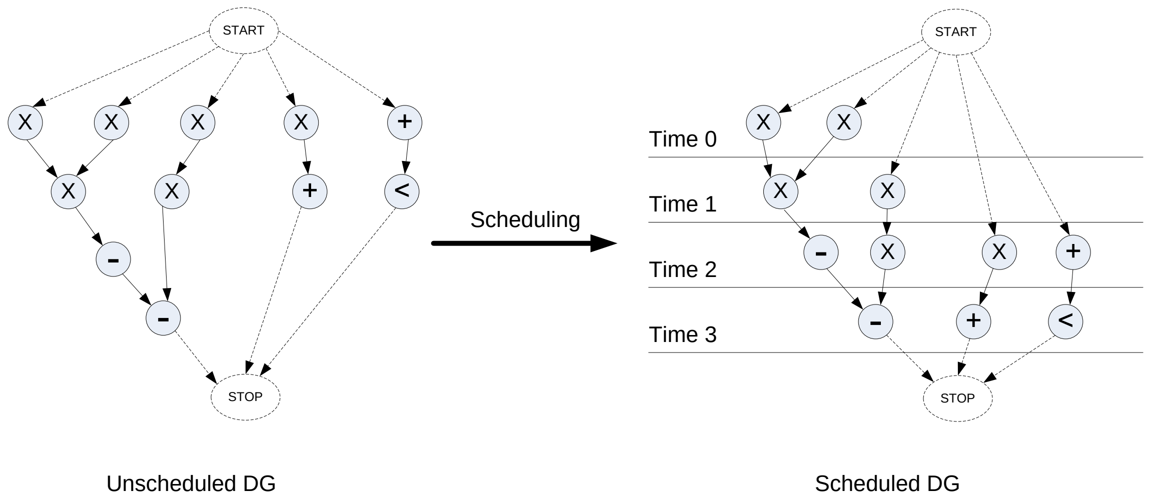


Figure 2.2: An example of an unscheduled DG and the scheduled DG after scheduling, all operations are assumed to have a latency of 1 clock cycle.

ordering of executions of all the operations in the graph. The execution order can be either highly serial for limited concurrency, or highly parallel for extreme concurrency, or anything in between. The amount of concurrency in the schedule directly affects the performance of the circuit. Scheduling also impacts the number of concurrent operations of any given hardware resource type at any step of the schedule, the maximum concurrency among all steps is the minimum number of hardware resources required for that type. Therefore the choice of a schedule also impacts the area of the implementation.

Many different types of scheduling algorithms have been proposed. Some algorithms are called "exact" algorithms which find the optimal scheduling result. The exhaustive scheduling algorithm in Expl [27], and the branch and bound algorithm by Davidson [28] are examples of "exact" algorithms. Although optimal scheduling results can be found with these "exact" algorithms, the computational complexity of these algorithm is typically NP-complete [29]. Thus these algorithms are often used for small size designs only, or used for generating an upper or lower bound for some optimization metrics as a reference point.

Other scheduling algorithms are called "heuristic" algorithms, whose goal is to find the suboptimal result without exponential run time. The First-Come-First-Served (FCFS, also known as As-Soon-As-Possible, or ASAP) scheduling, list scheduling, and critical path first scheduling studies by Davidson et al. are examples of simple scheduling heuristics. A more complex but still heuristic scheduling algorithm is the force-directed scheduling used in HAL [15] and SAM [30] systems. Force-directed scheduling aims at balancing the number of operations in each control step. These algorithms utilize various heuristic techniques to avoid the searching of the entire scheduling space, thus their runtime is typically only a fraction of the "exact" algorithms. However, the limited design space exploration normally cannot find the optimal scheduling result.

2.4 Resource Sharing

Resource sharing [31] is an important technique in hardware synthesis to reduce area cost. It reuses a component by sharing it between more than one operations. If the shared hardware uses less area than allocating a dedicated component for each operation, resource sharing becomes worthwhile. Figure 2.3 illustrates an example of sharing two multiplier operations with one multiplier component.

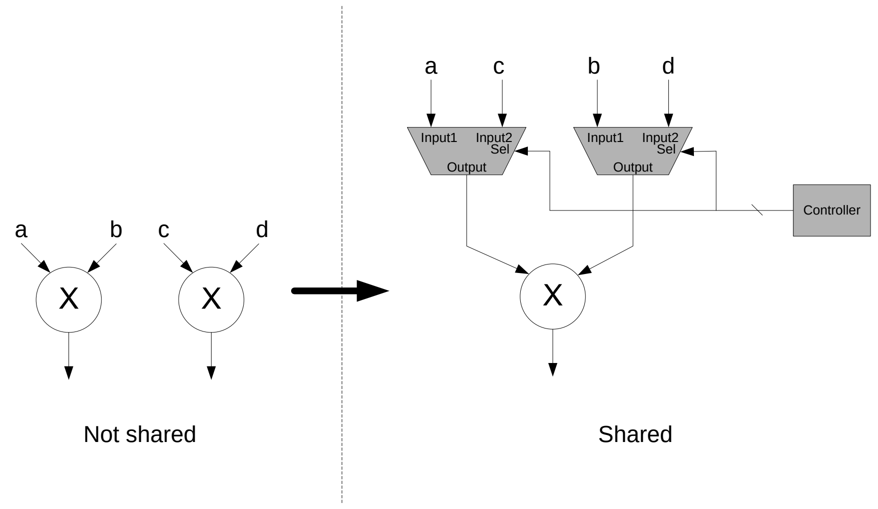


Figure 2.3: Resource sharing example

Resource sharing can only be applied when the resources saved by sharing are larger than the overhead area itself, and the extra hardware for resource sharing does not become the critical path of the implementation. Resource sharing does not come without a cost. Resource sharing overhead appears in the form of both area and time. To steer the input data to the shared circuit module, multiplexers and control logic must be employed to guide the correct input data into the shared module at correct time. As Figure 2.3 shows, the area overhead for resource sharing is made up of two parts: the multiplexer and the controller. The time overhead for resource sharing is the combinational delay of the extra hardware. The area and timing overhead of resource sharing is an important factor to determine if sharing is profitable or not.

This work proposes a resource sharing algorithm based on an FPGA-specific overhead model. The area and timing characteristics of resource sharing overheads in FPGAs are carefully studied and modeled. This model is used as a basis to differentiate between multiple sharing possibilities. This differentiation is explored in the resource sharing algorithm of the proposed algorithm. Thus more accurate resource sharing overhead is accounted for in the sharing algorithm, and more efficient hardware implementation is generated when there is more than one sharing possibility.

2.5 Module Library and Module Selection

Module selection [32] is the process of selecting the most optimized circuit module for each operation in the input algorithms. For example, there are two different circuit modules in the library to implement a multiplier function, one called *Array Multiplier* which has a latency of 8 clock cycles and an area of 366 FPGA slices, and the other called *Sequential Multiplier* which has a latency of 12 clock cycles and an area of 115 FPGA slices. If the synthesis goal is to minimize the hardware area, the *Sequential Multiplier* is a better choice. But if the synthesis goal is to minimize the execution latency, the *Array Multiplier* becomes the better choice.

Module selection is very important to improve the quality of high-level synthesis. Over the years, many numbers of FPGA specific circuit implementations have been published for performing primary arithmetic operations, which are the major building blocks of computationally intensive algorithms. Each implementation has different speed, area, latency etc. To synthesize high quality, minimum hardware FPGA implementations, these circuit modules must be carefully characterized and chosen during the architectural exploration process.

This work proposes a novel way to perform module selection with pipeline scheduling and resource sharing. A large variety of pipelined FPGA circuit modules are characterized and available for each operation. The module selection algorithm in this work proposes several ways to significantly reduce the module selection design space. It also proposes a novel algorithm to iteratively refine the module selection based on previous iteration's scheduling and resource sharing result. As the results in Chapter 6 show, this greatly improves the runtime of the module selection algorithm.

2.6 Design Space Exploration

A very large design space is formed by combining the three synthesis techniques of scheduling, resource sharing and module selection. The design space size (or complexity) of high-level synthesis algorithms can be measured by the number of all feasible implementations that can be generated by the algorithms. Mandal et al [29] show that the complexity of the scheduling problem in high-level synthesis is NP-complete. The complexity of the resource sharing problem is the same as the clique partitioning problem [31], which is also NP-complete. Exploring all module selection possibilities is an exponential search. If there are n_i implementation possibilities for operator i, and there are m_i operations of type i, then the total number of module selection possibilities is $\prod_{i=1}^{N} m_i^{n_i}$, where N is the number of unique operators. The design space created by combining these three techniques is even bigger than the design space of each technique applied alone. Efficiently exploring such a large design space is a major challenge of this combined approach.

Searching for an optimal implementation in such a big design space is further complicated by the close inter-relationship between these techniques. Each technique is dependent on the other two techniques. For example, scheduling depends on the latency of operations which is determined by module selection. It also depends on the sharing decision between operations, because shared operations cannot operate simultaneously. Resource sharing is dependent on module selection because operations with different module selections cannot be shared. Resource sharing is also dependent on scheduling because only operations that don't start simultaneously can be shared. These close inter-relationships make the sequence of these techniques in the combined algorithm extremely hard.

This work proposes a pipeline synthesis algorithm that concurrently performs pipeline scheduling, resource sharing and module selection. Although the combined design space of applying the three techniques together is much larger than the design space encountered when exploring one or two of the techniques, it can always generate much better synthesis results. To efficiently explore the combined design space, this work proposes an integrated algorithm which explores the three techniques together in an iterative way. It is based on the observation that the result of a previous iteration's exploration can be applied to subsequent explorations. This exploration strategy can significantly improve the search efficiency of the combined design space and generate near-optimal results. To the best of the author's knowledge, this is the first work that proposes efficient pipeline synthesis algorithms to explore the combined design space of the three techniques.

2.7 Summary

The goal of this work is to synthesize area efficient FPGA implementations from computationally intensive streaming algorithms under a user-specified throughput constraint. The proposed HLS methodology takes an SDF model as design input and uses a dependence graph with various properties as the internal representation format. It proposes efficient design space exploration algorithms to integrate three important synthesis techniques for FPGA specific high-level synthesis: pipeline scheduling, resource sharing and module selection. The next chapter discusses the pipeline scheduling algorithm proposed for this methodology.

Chapter 3

Pipeline Scheduling

Scheduling is an important step in the architectural exploration process of high-level synthesis. The goal of scheduling is to determine the start time of each operation in the data dependence graph with some predetermined optimization goals. The operation's start time determines the amount of concurrency of the implementation while the concurrency determines the performance and area of the final circuit.

Pipeline scheduling [32] (also called modulo scheduling) is an important extension of traditional non-pipelined scheduling. Pipeline scheduling generates a schedule that allows multiple iterations of the same computation to overlap in execution. It is usually used when a repetitive version of a computation needs to be scheduled. The need to arrange overlapping iterations (i.e. schedule a new computation when the previous one is not finished) makes pipeline scheduling more difficult than traditional scheduling. The terms, issues and algorithms of pipeline scheduling discussed in this chapter are very important for synthesizing throughput constrained and efficient FPGA implementation from an untimed functional specification.

This chapter begins with a detailed discussion of pipelining. It then discusses three important issues in pipeline scheduling: modulo start time, effective delay and feedback constraints. A review of previous pipeline synthesis work is provided afterward. Finally, a new pipeline scheduling algorithm called HOIMS (Hardware Oriented Iterative Modulo Scheduling) is proposed. It is based on the IMS algorithm [33], and addresses important issues for hardware oriented pipeline scheduling.

3.1 Pipelining

Pipelining is an important design technique used to increase the throughput of digital circuits. Used frequently in signal processing and data intensive calculations, pipelining increases the throughput of a computation by dividing a computation into discrete steps and operating on multiple samples simultaneously [34]. To increase the throughput of pipelined computations, stages for different computations are overlapped in time. Pipelining offers increased computational throughput at the expense of latency and inter-stage pipeline registers.

Pipelining also increases the system frequency. System frequency is defined as the reciprocal of the longest combinational delay between any two registers in a circuit. The path that has the longest combinational delay is defined as the "critical path" of the circuit. Pipeline registers are inserted between operations that belong to consecutive stages of pipelining to decrease the delay of the critical path.

Figure 3.1 shows an example of pipelining. In this figure, a 5-tap FIR filter is divided into 5 stages of computation. While stage i is processing the n^{th} input sample, stage $i + 1$ is still processing the $(n - 1)^{th}$ (previous) sample. If each stage takes 2 clock cycles, the pipelined circuit can process a new sample every 2 clock cycles on average, compared to a non-pipelined circuit where it can only process one sample every 10 clock cycles. The throughput of the pipelined circuit is five times that of the non-pipelined circuit. The cost of pipelining is an increased latency. The pipelined circuit now has a longer latency due to the pipeline registers (15 cycles vs 10 cycles). These pipeline registers also increase the hardware area cost.

3.1.1 Pipelining Terminology

From the scheduling perspective, the pipeline is treated as a linear time line that begins with cycle 0 and ends with the last cycle of the pipeline. The following terms will be used to describe the pipeline:

System Data Introduction Interval (δ) The time interval, in clock cycles, between two consecutive input samples.

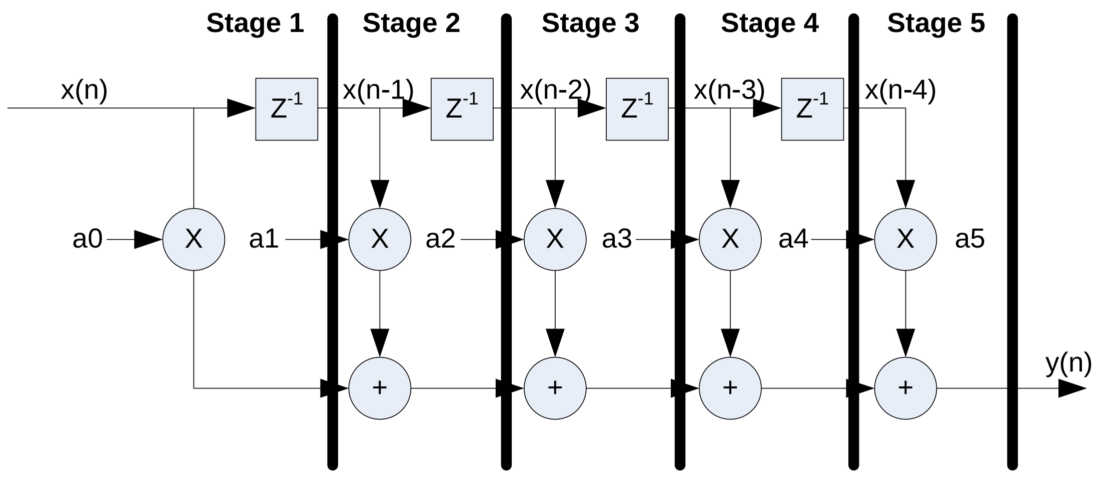


Figure 3.1: Five stage pipeline example for an 5-tap FIR filter

Pipeline Latency (λ) The length, in clock cycles, to complete a single iteration of the computation.

Pipeline Time Step (t) A specific clock cycle within the linear pipeline. A valid pipeline time step is within the range: $0 \leq t \leq \lambda - 1$.

Pipeline Stage (s) The pipeline is divided into discrete pipeline stages that are each δ cycles in length. There are $\lceil \frac{\lambda}{\delta} \rceil$ pipeline stages in the pipeline. The pipeline stage associated with time step t is $s_t = \lfloor \frac{t}{\delta} \rfloor$.

Pipeline Phase (ϕ) Each clock cycle within a pipeline stage is associated with a pipeline phase. The pipeline phase of time step t is $\phi_t = t \bmod \delta$.

Figure 3.2 illustrates this terminology with an example. In this figure, δ is 5 so a new computation will be initiated every 5 clock cycles. It has a pipeline latency (λ) of 10, so the pipeline time step (t) has a range from 0 to 9. It has 2 ($\lceil \frac{10}{5} \rceil$) pipeline stages and each stage has 5 phases.

The *system data introduction interval (δ)* is the most important parameter of every pipelined system. $\delta = 1$ indicates a *fully pipelined* circuit with a new computation initiated every clock cycle. $\lambda > \delta > 1$ indicates a *partially pipelined* circuit with a new computation initiated every δ cycles. *Non-pipelined* circuits occur when $\delta \geq \lambda$.

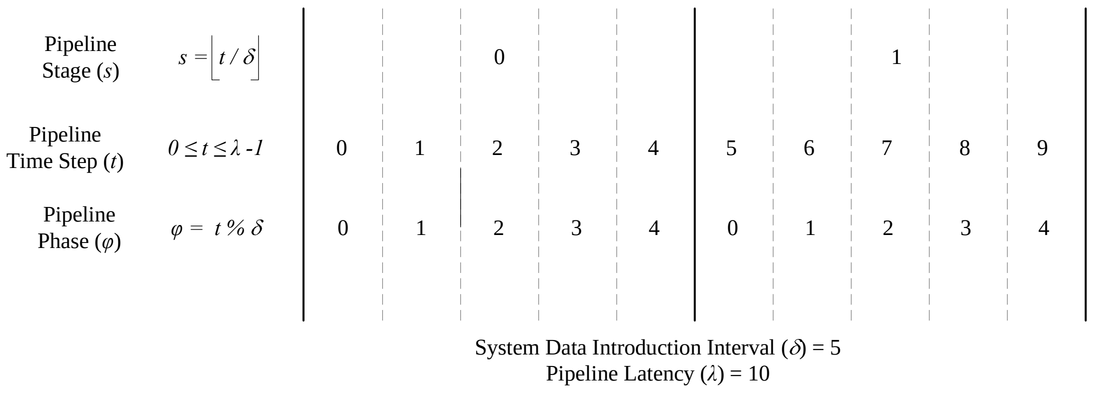

Pipeline Stage (s)
$s = \lfloor t/\delta \rfloor$
0
1
Pipeline Time Step (t)
$0 \leq t \leq \lambda - 1$
0 1 2 3 4 5 6 7 8 9
Pipeline Phase (φ)
$\varphi = t \% \delta$
0 1 2 3 4 0 1 2 3 4
System Data Introduction Interval (δ) = 5
Pipeline Latency (λ) = 10

Figure 3.2: Illustration of pipelining terminologies

The system data introduction interval and throughput are closely related. They are related by the system clock frequency (f) as shown in Equation 3.1. Throughput can be increased by increasing frequency or decreasing δ. Due to the integer constraint of system data introduction interval, δ should be represented by f and T as in Equation 3.2.

$$T = \frac{f}{\delta}. \tag{3.1}$$

$$\delta = \left\lfloor \frac{f}{T} \right\rfloor. \tag{3.2}$$

3.1.2 Pipelining Design Space

The pipelined circuits in this work are synthesized under a fixed throughput constraint. This throughput constraint, measured in samples per second, specifies the number of iterations of the computation that must be initiated each second. This is different from some other pipeline scheduling work where other constraints might be applied, such as system frequency, hardware resources and latency, etc.

The throughput requirement for a computationally intensive algorithm might be very different. As Table 3.1 shows, the throughput requirements might be very different for various video standards all using the same algorithm (such as a color space conversion). As a result, the final implementations for different throughput constraints can be very different, from a highly sequential, low throughput and low hardware area implementation to a highly parallel, high throughput and high hardware area implementation, as well as other implementation possibilities in between.

Table 3.1: Different throughput requirements for different DTV standards, assuming 30 frames/sec for each standard, therefore, the $throughput = width * height * 30$.

DTV Standard	Width	Height	Throughput (MSamples/Sec)
HDTV 1080p	1920	1200	69.12
HDTV 720p	1280	720	27.65
VGA	640	480	9.22
DVB-H QVGA	320	240	2.30
DVB-H QCIF	176	144	0.76

Equation 3.1 suggests that a pipelined circuit may meet the throughput constraint at a variety of clock frequencies. For example, a throughput constraint of 25 MSamples/sec may be met with a system clock frequency of 100 MHz and an initiation interval of 4 (i.e. $f = 100$ MHz and $\delta = 4$). Other combinations of system clock frequency and initiation interval that meet this constraint include: $(f, \delta) = (25$ MHz, 1), (50 MHz, 2), (150 MHz, 6), etc. The number of discrete (f, δ) combinations that meets the throughput constraint is large and represents a wide variety of implementation alternatives.

The size of the pipelining design space is limited by the feasible system data introduction interval values. The minimum system data introduction interval (δ_{min}) is limited by the feedback cycles in the data-flow graph which will be discussed later. The maximum system data introduction interval (δ_{max}) is limited by the slowest operation inside the data-flow graph:

$$\delta_{max} = \left\lfloor \frac{f_{max}}{T} \right\rfloor, \tag{3.3}$$

where f_{max} is the maximum possible system frequency which will be shown in Equation 5.1.

Figure 3.3 illustrates a sample pipeline scheduling design space. In this figure, a 30MSamples/Sec throughput constraint is assumed. Ideally, different (f,δ) pipelining pairs should be explored to find the best possible implementation, such as ($30MHz$,1), ($60MHz$,2), ($90MHz$,3), etc. However, any $\delta < \delta_{min}$ will violate the recurrence constraint, and any $\delta > \delta_{max}$ will violate the system maximum operating frequency constraint. The valid (f,δ) pairs presents the large pipeline scheduling design space.

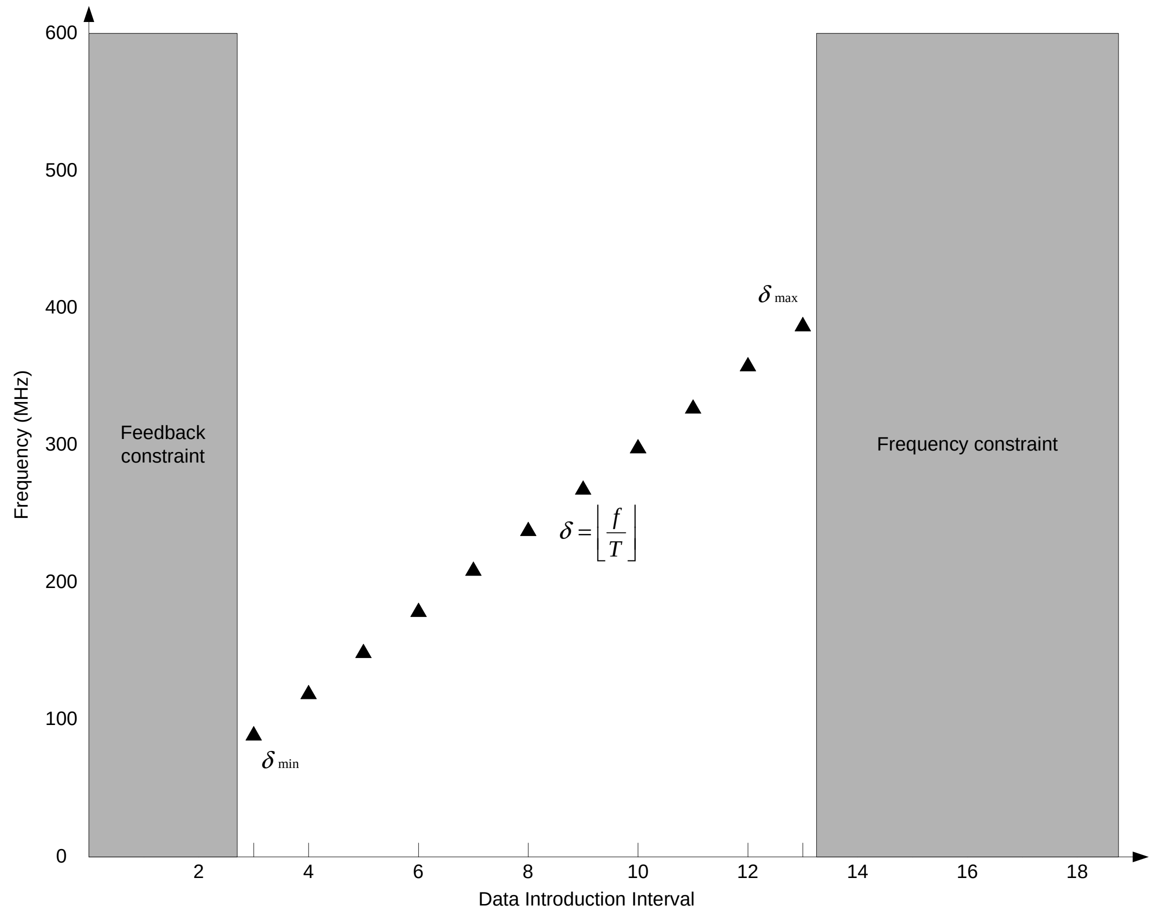


Figure 3.3: Pipelining design space illustration

3.2 Pipeline Scheduling Overview

Pipeline scheduling is an extension of traditional non-pipelined scheduling. Although it is similar to non-pipelined scheduling where the optimal start time of each operation needs to be identified, the repetitive computation due to pipelining makes it more complicated. Operations in a pipeline need to be *re-started* every δ cycles to process new input data. Different stages of a pipeline execute concurrently, which introduces a new type of dependency between operations. The new dependency, called "loop-carried dependency" in software context, also imposes possible feedback constraints in the original computation model. This section will study these pipeline scheduling specific issues.

3.2.1 Modulo Start Time

In a pipelined system, input samples are introduced into the system every δ cycles, so the computation and each operation must be *restarted* every δ cycles accordingly. An *iteration* is defined as one execution of all operations in the dependence graph for a single computation. Because the operation is repeated every δ cycles, pipeline scheduling only needs to find the start time for each operation during the *first* iteration. The start time for other iterations ($i > 0$) can be determined as follows:

$$t_s(n)^i = t_s(n)^0 + i * \delta, \tag{3.4}$$

where $t_s(n)^i$ is the start time of operation n in the i^{th} iteration, and $t_s(n)^0$ is the start time in the *first* iteration.

The feasible start time of each operation in pipeline scheduling can be limited to a window of size δ. Equation 3.4 suggests that if an operation cannot be scheduled at time t, it cannot be scheduled at any time $t + i * \delta$. If $t_{s_{min}}(n)$ is the earliest start time of operation n in the first iteration, then it is enough to only try scheduling the operation inside a window between $t_{s_{min}}(n)$ and $t_{s_{min}}(n) + \delta - 1$. If no feasible time can be found to schedule the operation within this window, the operation cannot be scheduled at any time according to Equation 3.4. So the latest start time of operation n can be represented as:

$$t_{s_{max}}(n) = t_{s_{min}}(n) + \delta - 1. \tag{3.5}$$

3.2.2 Effective Delay

Any scheduling algorithm must preserve the data dependencies between operations. An operation can start only after all of its input operations have been finished. In a pipelined circuit, different iterations of a dependence graph execute simultaneously. Data dependencies between two operations can be categorized into two types. The first type is called *intra-iteration dependency*, meaning the two operations are executed in the same iteration. This kind of dependency exists in both pipelined and non-pipelined systems. The second type is called *inter-iteration dependency*, meaning the two dependent operations are executed in two distinct iterations.

This work uses a non-negative integer edge weight, called "distance", to represent both types of data dependencies. Each "distance" is equal to the number of sample delays between the two operations in the SDF graph. Figure 3.4 shows the dependence graph for the input SDF model of the Biquad filter. Notice that all sample delay nodes in Figure 2.1 are converted into the non-negative "distance" values of the dependence edges in Figure 3.4. For example, the inter-iteration dependent operations *Add* and *Add1* are connected by an edge with a "distance" of 1, and the intra-iteration dependent operations *a2* and *Add7* are connected by an edge with a "distance" of 0.

Because pipeline scheduling only needs to decide the start time for the first iteration for each operation, the delay constraint described in Definition 1 in Section 2.3 for non-pipelined scheduling should be extended to consider the iteration difference between operations. Consider a successor j of an operation i with a dependence edge from i to j having a distance of $d(i, j)$. If $t_s(p)$ is defined as the start time of an

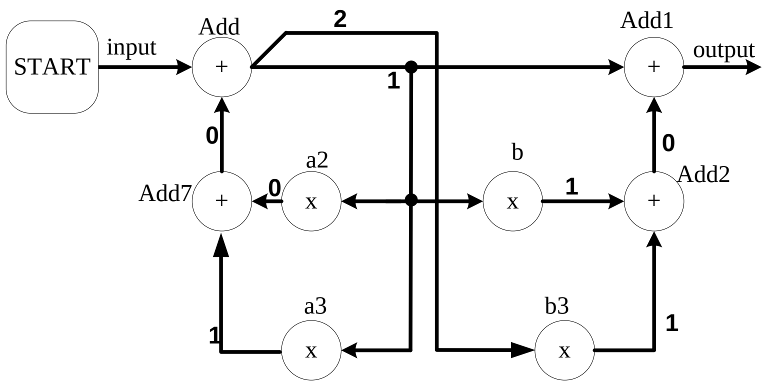


Figure 3.4: Dependence graph of the Biquad filter in Figure 2.1, integers on edges showing “intra” or “inter” iteration dependence between operations

operation p at the current iteration, the relationship between $t_s(i)$ and $t_s(j)$ is:

$$t_s(j) + \delta * d(i,j) \geq t_s(i) + \lambda_i \ . \tag{3.6}$$

This means that operation j at d iterations later cannot start until the operation i of current iteration finishes. Equation 3.6 can be rewritten as:

$$t_s(j) \geq t_s(i) + \lambda_i - \delta * d(i,j). \tag{3.7}$$

The “effective delay” between i and j is defined as:

$$\lambda_{eff}(i,j) = \lambda_i - \delta * d(i,j) \ . \tag{3.8}$$

This equation includes both the delay of the predecessor operation, as well as the inter-iteration ($d(i,j) > 0$) and intra-iteration ($d(i,j) = 0$) delay between two operations.

3.2.3 Feedback Constraints

Unlike many other scheduling algorithms, pipeline scheduling allows feedback in the dependence graph, which limits the throughput of the pipelined circuit. This section discusses the definition and implication of feedback constraints in pipeline scheduling, as well as an algorithm to compute the minimum system data introduction interval for pipeline scheduling caused by the feedback constraints.

Feedback constraints may occur in pipeline scheduling. If an operation in one iteration is directly or indirectly dependent on itself in a previous iteration, it is called a *recurrence* [33]. Pipeline scheduling represents this recurrence as a cycle within the dependence graph. Each cycle (C) within a dependence graph must be broken by an edge with positive "distance", representing a data dependence between different iterations of the computation. The recurrence creates feedback in the dependence graph.

The presence of feedback in the dependence graph limits the ability to increase throughput using pipelining. The recurrence constraint on the data introduction interval can be expressed in terms of the *Delay* and *Distance* of the cycles within the dependence graph. The delay of an elementary recurrence cycle, $Delay(c)$, is the sum of the module latencies (λ_m) for all operations within the cycle. The distance of the cycle, $Distance(c)$, is the sum of the sample delays within the cycle. For any operation in this cycle, suppose its start time for the current iteration is T_s, its start time for $Distance(c)$ iterations later is $T_s + Distance(c) * \delta$. The later start time must be after the completion of this operation's predecessors (i.e. all operations in this cycle), which is the cycle's accumulated latency. Thus the *Delay* and *Distance* of all recurrence cycles must satisfy the following relationship:

$$Delay(c) \leq \delta_c \times Distance(c). \tag{3.9}$$

Equation 3.9 implies a lower bound on the initiation interval of the resulting pipeline schedule. The minimum initiation interval, termed the recurrence minimum initiation interval ($RecMII$) in software pipelining, is the smallest δ_c that satisfies the above relationship for all cycles (C) in the dependence graph[1]. Thus,

$$\delta_{min} = \max_{c \in C} \left\lceil \frac{Delay(c)}{Distance(c)} \right\rceil . \tag{3.10}$$

Figure 3.5 shows an example of feedback constraints within the Biquad filter. There are two cycles in this figure ($c1$ and $c2$). The module latency (λ_m) of each operation within the two cycles is labeled beside each operation. For cycle $c1$, $Delay(c1)$ = 1 + 2 + 1 = 4, $Distance(c1)$ = 1. So the minimum data introduction interval for cycle $c1$ is 4/1=4. For cycle $c2$, $Delay(c2)$ = 1 + 4 + 1 = 6, $Distance(c2)$ = 2. So the minimum data introduction interval for cycle $c2$ is 6/2=3. According to Equation 3.10, the minimum system data introduction interval δ_{min} can be calculated as max(4, 3)=4.

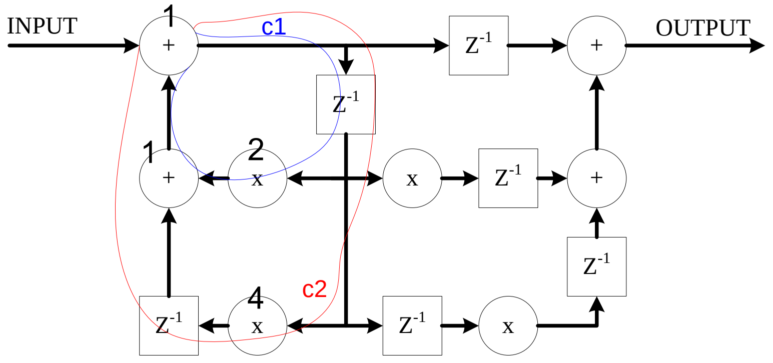


Figure 3.5: Illustration of feedback and its constraints on the minimum system data introduction interval (δ_{min}). The numbers on the operations represents the module latency (λ_m)

.

It is important to compute the minimum system data introduction interval of a dependence graph. As shown in Equation 3.1, the minimum value of δ determines the upper bound on throughput of a system if its frequency is fixed. For throughput constrained pipeline synthesis, δ_{min} determines the minimum clock frequency (f_{min})

[1]Software pipelining also introduces a *resource* constrained minimum initiation interval or ($ResMII$). No such constraint exists for hardware synthesis as the resources are not known at the start of the synthesis process.

at which the circuit must operate. The (f_{min}, δ_{min}) pair also represents the lower bound of the pipelining design space, because any δ smaller than δ_{min} makes the dependence graph unschedulable.

This work computes the minimum data introduction interval based on SCC (Strongly Connected Component) decomposition of a dependence graph. Identifying SCCs in a directed graph is much more efficient than finding all cycles in the graph. The complexity of the Kosaraju SCC detection algorithm [35] is $O(V + E)$ where V and E are the number of nodes and edges in the directed graph. A matrix called *minDist* can be used to test if a certain δ is valid for an SCC. The rows and columns of this matrix correspond to the nodes in the SCC, and the matrix entry [i,j] specifies the minimum permissible interval between the time at which operation i is scheduled and the time at which operation j, *from the same iteration*, is scheduled. If $MinDist$[i,i] is positive for any i, it means that node i must be scheduled later than itself, which is clearly impossible. This indicates that the δ is too small and must be increased until no diagonal entry is positive. The algorithm to compute the *minDist* matrix is described in Appendix B.1. The complexity of this algorithm is $O(N_{scc}^3)$ where N_{scc} is the size of the largest SCC.

3.3 Previous Pipeline Scheduling Work

Although pipelining is a common technique in circuit design, pipeline scheduling for hardware has not been studied extensively. Pipelining is normally categorized into two types[11]. The first is *functional pipelining*, which is defined as using non-pipelined components but putting pipelining registers on the circuit interconnection. Functional pipelining is uniquely characterized by the system data introduction interval δ. Since functional pipelining only puts registers on inter-connections, the critical path of the circuit might be the non-pipelined component which has the longest combinational delay. The second type of pipelining is called *structural pipelining* where pipelined components are used, but pipelining on the circuit itself is not performed. For structural pipelining, there is no overlapped execution between consecutive iterations, so the system data introduction interval is not applicable. Because the pipelining registers exist only inside the components, the critical path of the circuit might be the inter-connections between operations.

The Sehwa [32] project is the first synthesis work in the literature featuring functional pipeline scheduling. In that project, the theoretical foundation for pipelining a loop without loop-carried dependencies was presented. It has been shown that given a constraint on the number of resources, a pipelined data path can be implemented with minimum data introduction interval. Since the data introduction interval is fixed, the objective is to minimize the latency of the circuit. Sehwa tackles this problem using two polynomial-time pipeline scheduling algorithms. The first is called "feasible scheduling" which schedules with constraints on the total implementation cost. The second is called "maximal scheduling", which schedules for maximum performance assuming there is no cost constraint. Sehwa also incorporates an exhaustive algorithm for optimal scheduling. Here the search time is reduced by using the feasible or maximal schedule as an upper bound. The algorithms are invoked iteratively, and each scheduling cycle is guided by the performance and cost estimation for the previous schedule.

Force-directed scheduling algorithm was augmented in HAL [36] to support pipeline scheduling. The intent of the force-directed scheduling algorithm is to reduce the hardware resources by balancing the concurrency of the operations but without increasing the total latency. [36] proposed an improved force-directed scheduling that can be integrated into specialized or general purpose high level synthesis systems. HAL supports both fixed global timing constrained scheduling to minimize area, and fixed hardware resource constraints to minimize latency as well. The HAL system described in this paper also makes use of a stepwise refinement approach. The system does a preliminary allocation and uses that information to establish a schedule estimate. The allocation is repeated using the schedule to perform a much more detailed analysis and an improved selection of resources based on operation concurrency. The scheduler is then reinvoked and the final schedule is established by optimizing the use of the preselected resources. Further optimization of algorithm efficiency is described in [37].

The PLS pipeline scheduler [38] also focused on minimizing the latency of the schedule with a resource constraint as Sehwa did, but PLS can be applied to DFGs with or without loop-carried dependencies. It showed that latency has a strong relationship with the cost of registers and controller. The saved silicon area could be used to allocate additional resources and improve the throughput. It proposed an algorithm that iteratively uses forward scheduling and backward scheduling to achieve this goal.

3.4 HOIMS

This work proposed a novel pipeline scheduling algorithm called HOIMS (Hardware Oriented Iterative Modulo Scheduling). This algorithm is based on the IMS (Iterative Modulo Scheduling) algorithm which is targeted for software pipelining. This section will first briefly describe the original IMS algorithm. It will then discuss some important issues for hardware oriented pipeline scheduling. Finally, the HOIMS algorithm is presented and discussed.

3.4.1 IMS Algorithm Overview

The IMS algorithm was first proposed by Rau[33] from Hewlett Packard laboratory. IMS is targeted for software pipelining[39] of a loop body on microprocessors, and thus it is formulated as a resource constrained (due to the fixed number of functional units in a microprocessor), minimum data introduction interval pipeline scheduling algorithm. The main steps of the IMS algorithm is illustrated in Algorithm 3.1.

The resource constrained IMS algorithm is similar to traditional acyclic list scheduling. The outer-most loop of the IMS algorithm searches for the *minimum* system data introduction interval that can pipeline schedule the loop body (line 2). This search starts from the minimum possible data introduction interval (line 1, function MinII() is discussed in Section 3.2.3). With each candidate δ value, it checks the pipeline schedulability of the dependence graph (line 12). If it is schedulable, the schedule is recorded and the algorithm terminates; otherwise, it increments the δ

Algorithm 3.1: IMS algorithm

```
   IMS($budget$, $DG = (V, E)$)
   begin
1      $\delta \leftarrow$ MinII();
2      repeat
           Initialize all operations to be never scheduled;
3          HeightR($\delta$);
4          Insert all operations into $Q$;
           Schedule($START$, $0$), $budget \leftarrow budget - 1$;
5          while ($Q \neq \emptyset$ and $budget > 0$) do
6              $v \leftarrow$ HighestPriorityOperation($Q$);
7              $t_{s_{min}} \leftarrow$ CalculateEarlyStart($v$);
8              $t_{s_{max}} \leftarrow t_{s_{min}} + \delta - 1$;
9              $t_s \leftarrow$ FindTimeSlot($v$, $t_{s_{min}}$, $t_{s_{max}}$);
10             Schedule($v$, $t_s$), $budget \leftarrow budget - 1$;
11             $Q \leftarrow Q +$ UnscheduleConflicts();
12         $schedulable \leftarrow Q = \emptyset$;
13         $\delta \leftarrow \delta + 1$;
       until ($schedulable$ = true) ;
   end
```

value by one (line 13) and the pipeline scheduling is performed again with the new δ value.

For each candidate δ value, the algorithm keeps a priority queue (Q) of unscheduled operation nodes (line 4). The priority of each node is the calculated by the HeightR() function (line 3). HeightR() is a direct extension of the height-based priority [40] that is popular in acyclic list scheduling [41]. Each scheduling step takes the highest priority node in the queue (line 6). It then calculates the scheduling window of that operation (line 7 and 8) as discussed in Section 3.2.1. The earliest time slot to schedule it within that window (line 9) is then calculated. Finally the algorithm schedules the operation (line 10), while all conflicting nodes are unscheduled and added back into the priority queue (line 11). The scheduling continues until the queue is empty or the scheduling step threshold is reached, whichever is true (line 5). The former condition means a valid pipeline scheduling and bounding is found successfully, while the later one means the scheduling fails with the current δ value.

3.4.2 Hardware Specific Issues for Pipeline Scheduling

Traditional IMS scheduling is not sufficient for custom hardware oriented high-level synthesis. The target hardware for the IMS algorithm is a pre-fabricated microprocessor, which has both a fixed frequency and a fixed number of functional units. Hardware oriented pipeline scheduling algorithms must consider hardware specific issues when improving the IMS algorithm. With custom hardware, system frequency and hardware resources are not fixed. Instead, the scheduling algorithm needs to determine the optimal system frequency and hardware resources that can meet the synthesis goals.

The throughput constrained pipeline scheduling algorithm in this work explores different system frequencies. Incrementing the system data introduction interval alone with a fixed system frequency is equal to decreasing the system throughput as implied by Equation 3.1. To meet the throughput constraint, increasing the data introduction interval must increase the system frequency accordingly. However, real hardware circuits cannot run at infinite frequency, so the data introduction interval for hardware oriented pipeline scheduling should be bounded.

The minimum hardware oriented pipeline scheduling algorithm in this work allocates new resources when necessary. This is very different from the fixed number of functional units in pre-fabricated microprocessors. The limitation of hardware resources will prevent a feasible pipeline schedule even when the candidate δ value is larger than the minimum required one. In hardware oriented pipeline scheduling, new hardware resources can be allocated during the scheduling process. The allocation and scheduling minimize the hardware area of the synthesized circuit.

3.4.3 HOIMS Algorithm

Based on the IMS algorithm, a hardware oriented pipeline scheduling (HOIMS) algorithm is proposed. It addresses the specific issues for hardware oriented pipeline scheduling as discussed above. This algorithm serves as the backbone of the final pipeline synthesis algorithm, and will be used and expanded throughout this work. The main steps of the HOIMS algorithm are shown in Algorithm 3.2.

Algorithm 3.2: HOIMS algorithm

```
HOIMS($T$,$DG(V,E)$) begin
1   $\delta_{min} \leftarrow$ MinII();
2   $\delta_{max} \leftarrow$ MaxII();
3   $S_{best} \leftarrow \emptyset$;
4   for $\delta \leftarrow \delta_{min}$ to $\delta_{max}$ do
5       $f_\delta \leftarrow \delta \cdot T$;
6       Initialize all operations to be never scheduled;
7       HeightR($\delta$);
8       Insert all operations into $Q$;
9       Schedule($START$, $0$);
10      while $Q \neq \emptyset$ do
11          $v \leftarrow$ HighestPriorityOperation($Q$);
12          $t_{s_{min}} \leftarrow$ CalculateEarlyStart($v$);
13          $t_{s_{max}} \leftarrow t_{s_{min}} + \delta - 1$;
14          $(t_s, m) \leftarrow$ DynamicAllocate($v$,$T_{s_{min}}$,$T_{s_{max}}$);
15          Schedule($v$, $t_s$, $M$);
16          $Q \leftarrow Q +$ UnscheduleConflicts($v$, $t_s$, $M$);
17      if Cost($S_{current}$) < Cost($S_{best}$) then
18          $S_{best} \leftarrow S_{current}$;
end
```

The HOIMS algorithm is structurally similar to the IMS algorithm, but extended with numerous custom hardware synthesis specific features. The outer-most loop (line 4) iterates through *all* the feasible data introduction intervals with corresponding clock frequencies that all meet the throughput constraint. For each pipelining design point (a specific (f, δ) pair), iterative modulo scheduling is performed (line 5 to line 16) to find a corresponding pipelined schedule. Several algorithm steps are identical to those in the IMS algorithm, such as the *MinDist* matrix calculation algorithm used in determing the minimum system data introduction interval, the HeightR(δ) function to determine the schedule priority for each operation (line 7 and line 11), and the CalculateEarlyStart() function to calculate the earliest possible start time for each operation as discussed in Section 3.2.2. The details of the *MinDist* matrix calculation and HeightR(δ) function are discussed in Appendix B.1 and B.2 respectively.

HOIMS explores the entire pipelining design space as discussed in Section 3.1.2. For software IMS, the system frequency and hardware resources are fixed, and the exploration strategy is to find the *minimum* data introduction interval, where a pipeline schedule is feasible. The result is a pipeline schedule with maximum possible throughput. In HOIMS, system throughput is an input constraint that must be guaranteed. So, in addition to increasing the data introduction value, the corresponding system frequency is also increased proportionally (see line 5). This (f, δ) design space is bounded as discussed in section 3.1.2 due to the maximum operating frequency of hardware circuit modules.

HOIMS compares and selects a pipeline schedule that has the minimum hardware area cost. Unlike the IMS algorithm where the algorithm terminates once a schedule is found, the HOIMS algorithm compares the pipeline schedule at each pipelining design point with the best schedule (S_{best} in line 17). If the current schedule has a hardware area less than the best schedule, S_{best} is updated with the current schedule (line 18). The result of this comparison is a *minimum* hardware pipelined schedule for a *fixed* throughput constraint.

Like the traditional latency constrained, minimum hardware scheduling algorithm, the HOIMS algorithm dynamically allocates hardware resources. In Algorithm 3.2, this dynamic allocation is performed by the DynamicAllocate() function (see line 14). DynamicAllocate() returns not only the time slot (t_s) for the operation to schedule, but also the hardware resource for the operation to bind to. The hardware resource can be either a newly allocated circuit module, or a previously allocated circuit module that is free at the preferred scheduling time slot t_s, depending on the algorithm in the DynamicAllocate() function. A simple dynamic resource allocation algorithm would be to re-use as much as possible the previously allocated hardware resources. If no such resources are available, a new hardware resource would be allocated.

The HOIMS algorithm will be expanded in the following chapters with dynamic resource allocation and sharing, as well as module selection. In Chapter 4, the algorithm for DynamicAllocate() will be proposed. The sharing algorithm will not only take FPGA specific resource sharing overhead into account, it will also explore the sharing design space for each operation. In Chapter 5, the HOIMS algorithm will be further expanded with the integration of module selection. Unlike software IMS, the availability of multiple circuit modules for a single operation provide tradeoffs between area and performance, which can greatly affects the scheduling and sharing algorithm.

3.5 Summary

This chapter describes one important technique for design space exploration: pipeline scheduling. Pipelining is very important for implementing computationally intensive algorithms on FPGAs. Pipeline scheduling is an extension of traditional non-pipelined scheduling. It is more difficult due to the overlapping iterations. This work proposes a hardware oriented pipeline scheduling algorithm called HOIMS. It is based on the efficient IMS algorithm which is targeted for software pipelining on resource constrained microprocessors. To the author's best knowledge, HOIMS is the first to explore the entire pipelining design space and provides hardware specific features. It will be integrated with module selection and resource sharing techniques in the following chapters.

Chapter 4

Resource Allocation and Sharing

Resource allocation and binding are very important in the architectural exploration process of high-level synthesis. Resource allocation allocates the hardware components needed to implement the computationally intensive algorithms. Resource binding determines which hardware component should be used for each operation in the dependence graph. The number of hardware components and the binding of these components to the specific operations directly impact the area and performance of the final hardware circuit.

Resource sharing is an essential component of resource allocation to synthesize efficient FPGA implementations. Resource sharing assigns multiple operations in the dependence graph to a single hardware component. Although this can significantly reduce the synthesized hardware area, the overhead cost caused by resource sharing must be carefully quantified to justify the sharing benefits.

This chapter discusses important issues and algorithms to integrate resource allocation and sharing techniques into HOIMS. It begins with an overview of resource allocation and sharing. It then discusses resource sharing overhead for FPGA implementations. The inter-relationship between resource sharing and pipeline scheduling is then discussed. After a review of previous work on resource sharing, a new resource allocation and sharing algorithm based on weighted compatibility graphs is proposed and integrated into the HOIMS algorithm.

4.1 Resource Allocation and Sharing Overview

Resource sharing is the assignment of a hardware component to more than one operation. Resource allocation may infer some resource sharing, even though they do not imply a particular binding. For example, there are only two hardware multiplier components allocated for a dependence graph which contains three multiplier operations. Although this allocation mandates that at least two operations should share one hardware component, the allocation does not necessarily imply a specific binding, because any two operations can be bound to one of the hardware components, and the other operation be bound to the other hardware component.

Although resource sharing can avoid allocating a dedicated hardware component for every operation, it comes with the price of sharing overhead. To support resource sharing, multiplexers, controllers and interconnections are required to steer data from multiple sources to a single hardware component. Successful resource sharing algorithms must carefully balance the circuit area saved by sharing against the overhead area introduced by that sharing, thus reducing the overall hardware area. The resource sharing overhead for FPGAs has been carefully studied in this work and is discussed in Appendix C.

Resource sharing or binding can be performed *after* or *before* scheduling depending on the style of dataflow. For resource dominated circuits (i.e. the majority of circuit area is functional units rather than the overhead hardware which guides data between functional units), resource sharing is normally performed after scheduling. This is because the area and latency for this type of circuits are dominated by resource usage and the schedule. Thus resource sharing can maximally assign non-concurrent operations to a single hardware component without considering the impact of sharing overhead. For general circuits (i.e. the overhead hardware area is comparable to functional unit area), resource sharing is preferably performed before scheduling, so that the steering logic area and delay can be derived from the binding or sharing. Thus the overall area and timing of the synthesized circuit can be estimated with more accuracy[10].

4.2 Resource Sharing and Pipeline Scheduling

The use of multi-cycle, pipelined circuit modules in this work creates a close relationship between resource sharing and pipeline scheduling. On one hand, the pipeline schedule's data introduction interval (δ) limits the sharing capability of the circuit modules. On the other hand, resource sharing between certain operations may affect the pipeline schedulability of a dependence graph.

4.2.1 Multi-cycle, Pipelined Circuit Module

Unlike most previous high-level synthesis work, this work emphasizes the use of multi-cycle, pipelined circuit modules. Pipelined modules allow the circuit to operate at a higher clock frequency than possible without module pipelining and is essential for FPGAs. It also allows the use of multi-cycle circuit modules. Although pipelined modules may take longer latency to operate on a single data input than non-pipelined modules, they allow *overlapped* operations on a single circuit module by introducing new input data before the operation on the existing data has completed.

Multi-cycle, pipelined circuit modules are characterized with similar parameters of pipelining. Module latency (λ_m) is the number of clock cycles to complete a single computation on a circuit module. Module data introduction interval (δ_m) is the minimum number of clock cycles between two consecutive inputs to a circuit module. Such a module may initiate a new computation at least δ_m clock cycles after the input of the previous data. Module frequency (f_m) is the maximum operating clock frequency of a circuit module. It is determined by the longest combinational delay inside the circuit module. Module area (A_m) is the amount of hardware resources (i.e. FPGA logic slices) the circuit module requires.

Multi-cycle circuit modules can be classified based on the relationship between λ_m and δ_m. Fully pipelined modules occur when $\delta_m = 1$ and $\lambda_m \geq 1$. Partially pipelined modules occur when $1 < \delta_m < \lambda_m$. Non-pipelined modules occur when $\lambda_m = \delta_m$.

4.2.2 Resource Sharing Capability with Pipeline Scheduling

The sharing capability of a pipelined circuit module is determined by the system data introduction interval (δ) and the module's data introduction interval (δ_m). The maximum number of operations that can share a pipelined circuit module is:

$$\text{share}_{max} = \left\lfloor \frac{\delta}{\delta_m} \right\rfloor . \tag{4.1}$$

As shown in Equation 4.1, a multi-cycle pipelined circuit module may be shared only if the data introduction interval of the module is less than or equal to half the data introduction interval of the global pipeline (i.e. $\delta_m \leq \frac{1}{2}\delta$ thus $share_{max}$ is larger than one). It is important to note that δ_m of any circuit module allocated for a pipeline must be less than or equal to δ, otherwise, $share_{max}$ becomes zero meaning no operations can be assigned to this circuit module. This requirement ensures that the module will complete initiation on a given sample in time for the circuit to operate on the next sample. For non-pipelined circuit modules (i.e. combinational circuit modules), their outputs can be registered and thus be treated as pipelined circuit modules with $\delta_m = 1$.

Figure 4.1 illustrates Equation 4.1 with a two dimensional figure. The y-axis shows the progress of time in terms of clock cycles. The x-axis shows the location of the data inside a pipelined circuit module at each clock cycle. Assume a multi-cycle pipelined circuit module has a module data introduction interval of 3 ($\delta_m = 3$), and that the global system data introduction interval of the pipeline is 5 ($\delta = 5$). Assume that value $x(n)$ is fed into the module at clock cycle 0 ($t_{x(n)} = 0$). Although another value y can be fed into the module at clock cycle 3 or later (i.e. $t_y \geq t_{x(n)} + \delta_m$), it is not possible due to the system pipelining. The global system data introduction interval of the pipeline mandates a new value $x(n+1)$ be fed into the module at clock cycle 5. So introducing another value at cycle 3 or 4 prevents value $x(n+1)$ at cycle 5, which is not possible. Thus the sharing capability of this circuit module under this pipelining condition is $\left\lfloor \frac{5}{3} \right\rfloor = 1$.

The exploration of the pipelining design space (i.e. iterating through feasible (f, δ) pairs) has a big impact on resource sharing capability. When δ is small, the resource sharing capability of the circuit is limited (i.e. small $\left\lfloor \frac{\delta}{\delta_m} \right\rfloor$). In the extreme case where δ=1, no resource sharing is possible. As δ increases, the overall hardware area of the circuit decreases due to the benefits of more resource sharing (i.e. larger

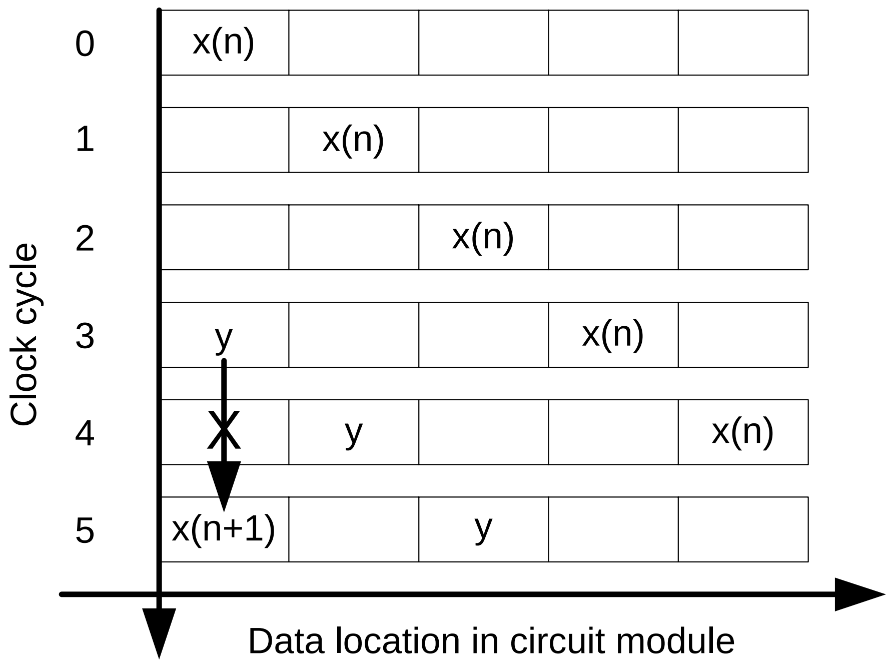


Figure 4.1: Illustration of the sharing capability of a pipelined circuit module in the context of a pipelined schedule, assuming $\delta = 5, \delta_m = 3, \lambda_m = 5$.

$\left\lfloor \frac{\delta}{\delta_m} \right\rfloor$). When exploring very large δ values, the corresponding system frequency becomes very high. At such high frequencies, only deeply pipelined or sequential circuit modules can operate. For deeply pipelined modules, the sharing capability is high, but the higher cost of deeply pipelined modules might overcome the resource sharing savings. For high speed sequential circuit modules, due to their relatively high module data introduction interval (i.e. small $\left\lfloor \frac{\delta}{\delta_m} \right\rfloor$), the resource sharing capability is limited.

4.2.3 Resource Sharing and Pipeline Schedulability

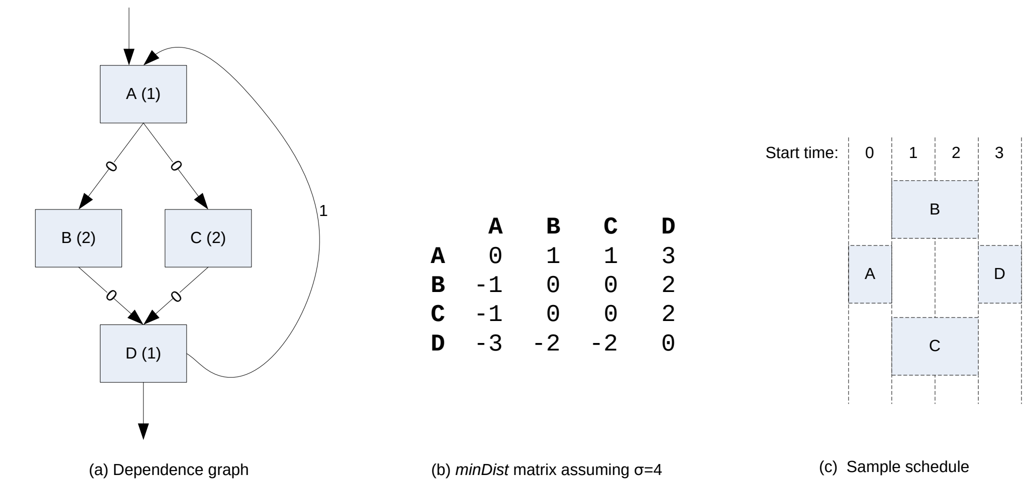
As discussed in Section 3.2.3 of the pipeline scheduling chapter, the *minDist* matrix can be used to determine the pipeline schedulability of a dependence graph with feedback edges. The dependence graph is pipeline schedulable only if none of the diagonal elements in the *minDist* matrix is positive (i.e. matrix element $minDist[P][P]$, also called slack value t_{slack} of operation P, must be ≤ 0). However, this schedulability check does not take possible resource sharing between these operations into account. It is possible that the *minDist* matrix has no positive diagonal elements, yet the dependence graph is still not pipeline schedulable.

Pipeline schedulability under the context of resource sharing can be checked by the earliest start time ($t_{s_{min}}$) and the slack value (t_{slack}) of the shared operations. The time from $t_{s_{min}}$ to $t_{s_{min}} + \text{abs}(t_{slack})$ is defined as the *scheduling window* (w) of an operation. If two operations share a circuit module whose module data introduction interval is δ_m, the start time of the two operations must be separated by at least δ_m cycles. This mandates a relative scheduling constraint between these two operations. If the scheduling window of the two operations cannot honor this constraint, these two operations cannot be shared. In other words, sharing these two operations will make the dependence graph unschedulable.

Figure 4.2 shows such an example. A dependence graph and its *minDist* matrix with $\delta = 4$ are shown as part (a) and (b) of this figure. The latency of the operations are labeled beside each operation's name. Since none of the diagonal members of the *minDist* matrix are positive, this graph is schedulable without resource sharing (i.e. each operation is allocated with a dedicated hardware component). A feasible schedule is illustrated as part (c) of Figure 4.2. Now assume operation B and C are shared, and the circuit module they shared has a $\delta_m = 1$. This requires that the start time of B and C be separated by at least one clock cycle. However, the slack value of both operations is zero, and their earliest start time with regard to the same operation A is the same ($minDist[A][B] = minDist[A][C] = 1$). So the scheduling window of operation A and B is the same (i.e. $w_a = w_b = [1,\ 1]$). Obviously, the start time of the two operations cannot be separated by 1 clock cycle. Thus, sharing between operation B and C makes this graph unschedulable.

4.3 Previous Resource Sharing Work

Resource sharing is a well known technique and there is much relevant research in high-level synthesis research. A common method to formulate and solve the resource sharing problem is partitioning a compatibility graph into a set of cliques, where each clique represents the operations that share a single hardware component



$$\begin{array}{c|cccc} & A & B & C & D \\ A & 0 & 1 & 1 & 3 \\ B & -1 & 0 & 0 & 2 \\ C & -1 & 0 & 0 & 2 \\ D & -3 & -2 & -2 & 0 \end{array}$$

Figure 4.2: Resource sharing may prevent a dependence graph from schedulable even with a valid *minDist* matrix

[10]. A compatibility graph $CG(\mathcal{V}, \mathcal{E})$ is defined as an undirected graph, whose vertex set $\mathcal{V}$ is in one-to-one correspondence with the operations in the dependence graph, and whose edge set $\mathcal{E}$ denotes the *compatible* operations pairs. Two operations are *compatible* if they can be implemented by the same component type and they can be scheduled non-concurrently. A group of mutually compatible operations correspond to a subset of vertices that are all mutually connected by edges in the compatibility graph (i.e. a clique). An optimum resource sharing is one that minimizes the number of required hardware resources. Since one can associate a hardware component with each clique, the resource sharing problem is equivalent to partitioning the compatibility graph CG into a minimum number of cliques [31]. However, general clique partitioning algorithms are NP hard. Thus quite a few heuristics are proposed to simplify the algorithm and obtain sub-optimal solutions.

Springer and Thomas [42] investigated the features of high-level representation and high-level synthesis algorithms that give rise to special compatibility graphs. They provided insights to why and how interval and circular arc graphs occur in high level synthesis algorithms, which are the two special compatibility graphs that have been exploited by previous work to perform clique partitioning in polynomial time. They also introduced two additional ones, the chordal and the comparability graphs, to be used in high level synthesis. Chordal graphs are a special type of conflict graph; they can be recognized and colored in $O(V + E)$ time, where E is the number of edges and V is the number of nodes in the conflict graph. A comparability graph is a special type of compatibility graph. The cliques in a comparability graph are covered by directed paths. This makes finding cliques very easy, which can help speed up existing clique partitioning heuristics and create better clique partioning algorithms for mapping objects onto shared resources.

Raje et al. [31] proposed a heuristic algorithm to perform resource sharing for both registers and functional units considering several resource sharing costs. In that paper, they claimed that most clique-partitioning based resource sharing algorithms use local and inaccurate cost-functions which result in inefficient results. So they presented several global cost functions for estimating merging cost, interconnect cost, and control cost. These cost functions can be applied to guide the clique partitioning algorithms for faster algorithm convergence and more efficient and accurate resource sharing.

Ku and Micheli studied the problem of resource sharing before scheduling in [43]. In that book, they proposed a *weakly compatibility graph* where all operations of the same type are connected. A *conflict resolution* task is then performed to serialize those compatible operations. Serialization between two operations is marked by a constraint edge between these operations, which will be honored during the scheduling step. They also proposed a *strongly compatibility graph* where two operations of the same type and that are either alternative or data dependent are connected. This does not require any serialization constraint on the schedule. They claimed that although performing resource sharing before scheduling can estimate more accurately the overall area and delays, no efficient algorithm is known to compute minimum-area (or minimum-latency) sharing under latency (area) constraints, aside from enumerative techniques.

Mondal and Memik [44] proposed an algorithm to perform resource sharing on a pipeline scheduled FPGA circuit. Resource sharing is allowed between different basic blocks in the same pipeline stage, without violating the performance constraint.

In that paper, they create a direct relationship between available time slack on operations and the multiplexing overhead due to sharing. Flexibility is maximally exploited without violating any throughput constraint. They proposed an optimal algorithm for constant slack resource sharing and a heuristic for non-constant slack resource sharing. However, the paper only uses a simple linear timing model for multiplexers and doesn't include the area overhead.

This dissertation proposes a resource sharing algorithm which has several distinctions from previous work. First, it performs resource allocation and sharing *during* pipeline scheduling, while most previous work do them *after* or *before* scheduling. Second, the overhead cost used by the resource sharing algorithm in this work is based on actual characterization of the sharing overhead in FPGA technology (see Appendix C). Third, this work uses a *weighted* compatibility graph to represent the cost associated with different sharing possibilities, so that more area efficient hardware architectures can be synthesized. The *weight* idea is similar to the "projected area saving" in [31], but different in the cost function.

4.4 Weighted Compatibility Graph

One way to improve the compatibility graph used in most previous work is to add a *weight* to the graph. If the compatibility graph is not weighted, two cliques with the same number of compatible operations are considered to have the same hardware cost. For example, Figure 4.3 shows a sample compatibility graph with four operations (part (a)). If the hardware component has a sharing capability of two (i.e. the size of each clique cannot exceed two), two clique partitioning solutions are possible. The first solution is forming one clique with A and B, while forming the other clique with C and D (see part (b) of Figure 4.3). The second solution is forming one clique with A and D, while forming the other clique with B and C (see part (c) of Figure 4.3). Either of these two solutions can be used because they both have the same number of cliques, and each clique has the same number of operations. However, if sharing A and B saves more hardware area than sharing A and D does, then the first clique partitioning solution is obviously better than the second one. In this case, the first solution should have a larger *weight* than the second solution.

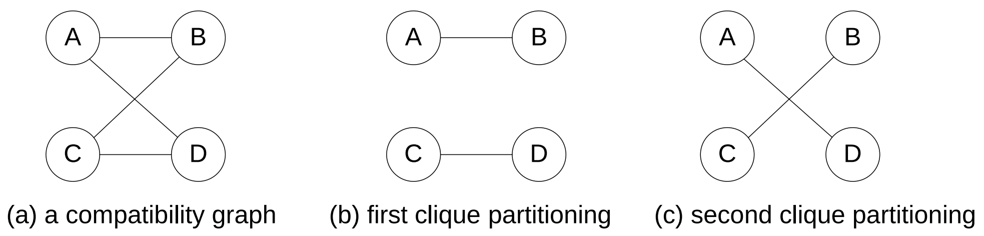

A
B
C
D
(a) a compatibility graph
A
B
C
D
(b) first clique partitioning
A
B
C
D
(c) second clique partitioning

Figure 4.3: A sample compatibility graph and two possible clique partitioning.

This work proposes a weighted compatibility graph to represent the cost associated with different sharing choices. This technique generates the resource sharing that results in minimum overall hardware area. The weighted compatibility graph is defined as an undirected graph $WCG = (V, E)$, where V is the vertex set of WCG and E is the edge set of WCG. Each vertex $v \in V$ represents a “real” operation (i.e. an operation requiring a hardware component) in the corresponding dependence graph, such as an *ADD* or *MULTIPLY* operation. It is a subset of the vertex set of the dependence graph because a vertex in a dependence graph can also be a pseudo vertex (i.e. an operation that doesn’t require a hardware component), such as a *PORT* or a *START* vertex. Each undirected edge ($e \in E$) of WCG represents a compatible relationship between the two operations that are connected by e. In other words, the two operations can be implemented by the same type of hardware component. Every edge has a weight (w) representing the hardware area that can be saved by sharing the two operations. The weight is based on three similarity relationships between the two operations: port similarity, source similarity and sink similarity.

4.4.1 Port Similarity

The bit-width of compatible operations in a dependence graph may be different. The greater the similarity between these operations’ bit-widths, the greater the hardware utilization ratio will be when these operations are shared. Port similarity

(s_{port}) represents the bit-width similarity between the input ports of two compatible operations. It is defined as:

$$s_{port} = \frac{\sum_{i=1}^{N} \frac{w(i)_{smaller}}{w(i)_{bigger}}}{N}, \tag{4.2}$$

where $w(i)_{smaller}$ is the smaller bit-width of the i^{th} port of the two operations, $w(i)_{bigger}$ is the bigger bit-width of the i^{th} port of the two operations. (This work only allows operations with the same number of input ports to be shared, so N is the number of input ports of either operation.) The maximum value of s_{port} is 1 when the two operations have the same bit-width. In this case, the shared hardware component is fully utilized by both operations. When the bit-width of two operations are different (i.e. $s_{port} < 1$), the shared hardware component is under-utilized by the smaller bit-width operation.

Figure 4.4 shows an example of port similarity and its relationship with the shared hardware utilization. Operation A has two input ports with each port's bit-width equal to 10. Operation B has two input ports, but the bit-width of its ports is 5. The port similarity s_{port} is equal to $(5/10 + 5/10)/2 = 1/2$. To share the two operations, a hardware component with each port's bit-width of 10 must be used, and each port should be connected to the output of a 10-bit multiplexer. Hence, the shared hardware is under-utilized by the smaller bit-width operation. Assume the area for the 10-bit component is 100, and 50 for the 5-bit component. The saved hardware area with sharing is: $100 + 50 - 100 - 2 * Mux_2(10) = 50 - 2 * Mux_2(10)$. However, if the bit-width of operation B's ports is also 10, the port similarity s_{port} is equal to $(10/10 + 10/10)/2 = 1$. In this case, the shared hardware is fully utilized by both operations, and the saved hardware area becomes: $100 + 100 - 100 - 2 * Mux_2(10) = 100 - 2 * Mux_2(10)$. As shown by this example, the larger the port similarity, the more shared hardware utilization will be, and the more hardware area can be saved.

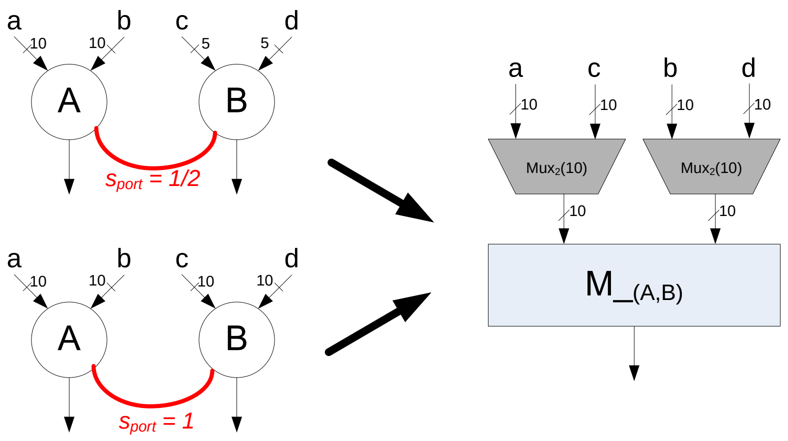


Figure 4.4: Port similarity (s_{port}) example

4.4.2 Source Similarity

Source similarity (s_{source}) represents the similarity of the source operations that feed the input ports of the two operations. Common sources can reduce the multiplexer and routing resources when the two operations are shared. This is defined as:

$$s_{source} = \frac{N_{common\_source}}{N}, \tag{4.3}$$

where N_{common_source} is the number of input ports which have common sources, and N is the number of input ports of the operation. Figure 4.5 shows an example of source similarity. In part (a), the input sources of the two operations are the same operations (i.e. $s_{source} = 2 \ / \ 2 = 1$). Sharing the two operations does not require a single multiplexer. In part (b), there are no common source operations for the two operations (i.e. $s_{source} = 0 \ / \ 2 = 0$). Thus it requires two multiplexers to guide the input data. As shown in this example, the larger the source similarity is, the more hardware area that can be saved by sharing the two operations.

4.4.3 Sink Similarity

Sink similarity (s_{sink}) represents the similarity of sink operations that are fed by the outputs of the two operations. Common sink operations can reduce the routing

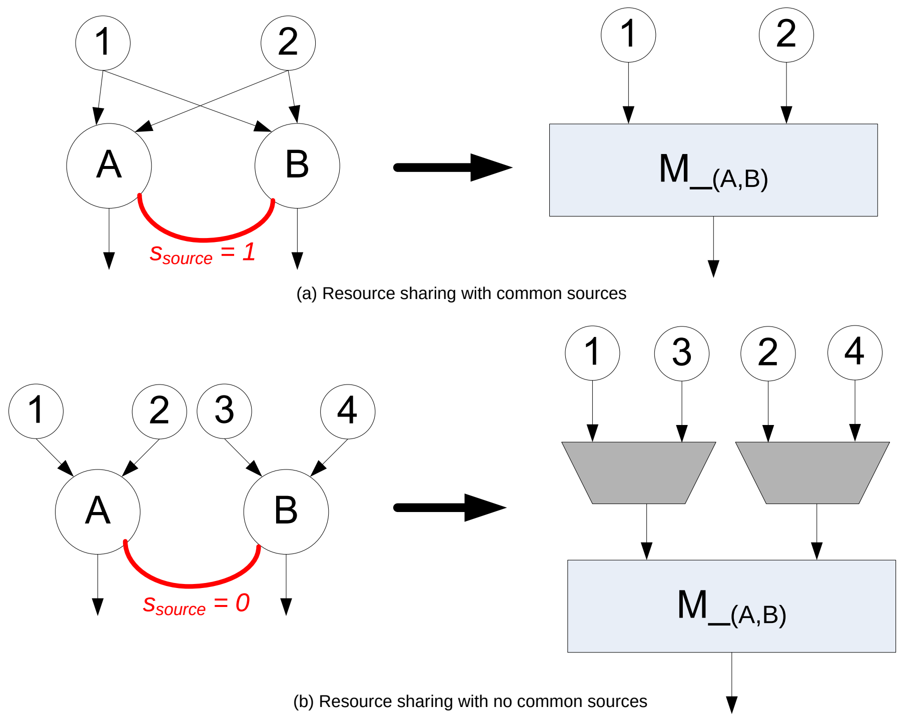

1
2
A
B
s_{source} = 1
M_(A,B)
(a) Resource sharing with common sources
1
2
3
4
A
B
s_{source} = 0
1
3
2
4
M_(A,B)
(b) Resource sharing with no common sources

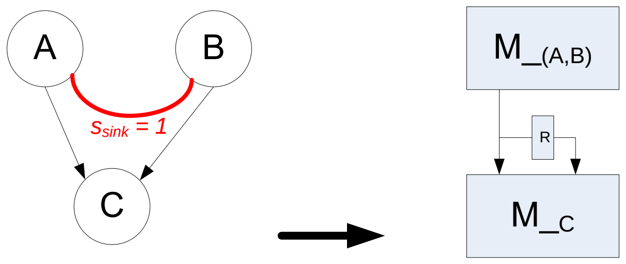
Figure 4.5: Source similarity (s_{source}) example

resources when the two operations are shared. It is defined as:

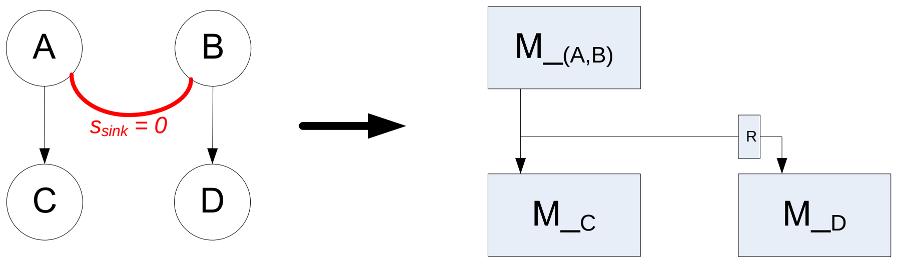
$$s_{sink} = \frac{N_{common\_sink}}{N}, \tag{4.4}$$

where N_{common_sink} is the number of common sink operations between the two operations, and N is the number of total sink operations of the two operations. Figure 4.6 shows an example of sink similarity. In part (a), the outputs of operations A and B are fed to the same sink operation (i.e. $s_{sink} = 1 \ / \ 1 = 1$). In part (b), the outputs of the two operations are fed into two different sink operations ($s_{sink} = 0 \ / \ 2 = 0$). As shown in Figure 4.6, sharing in part (b) requires a longer routing net than sharing in part (a). Thus, the larger sink similarity, the more hardware area can be saved by sharing the two operations.

A heuristic minimum cost clique partitioning is performed on the weighted compatibility graph to generate the resource sharing that saves the most hardware area. The three similarities discussed above are added up to form the weight (w)



(a) Resource sharing with common sinks



(b) Resource sharing with no common sinks

Figure 4.6: Sink similarity (s_{source}) example

for each edge in the WCG. The weight of each clique is the aggregated weight of all edges which belong to that clique. The clique weight provides a cost function to differentiate multiple sharing possibilities, thus exploration of the resource sharing design space can be performed more efficiently. The clique partitioning is performed simultaneously with the pipeline scheduling step - it partitions the WCG into a list of cliques so that the sum of all the clique's weight is minimal. The details of the algorithm is explained in the next section.

4.5 Resource Allocation and Sharing in HOIMS

The HOIMS algorithm performs dynamic resource allocation and resource sharing during pipeline scheduling. These two tasks are performed when the function DynamicAllocate() is called in Algorithm 3.2 of the previous chapter. For each operation to be scheduled, the scheduling algorithm determines the start time window for that operation, then DynamicAllocate() determines the start time along with the hardware component which the operation will be bound to. The hardware component can either be newly allocated, or a previously allocated component. The detailed algorithm steps of this function is illustrated in Algorithm 4.1.

Algorithm 4.1: Resource sharing exploration in HOIMS

```
  Result: $t_s$, $m$, $cOps$
  DynamicAllocate($op, t_{min}, t_{max}$) begin
1     $mAlloc \leftarrow$ allocatedComp($op$);
2     if notWorthShare($op$) or $mAlloc = \emptyset$ then
          $t \leftarrow t_{min}$;
          $m \leftarrow$ create a new component for $op$;
          $cOps \leftarrow$ null;
3         return $t$, $m$, $cOps$;
      $maxSimCompC \leftarrow$ null;
      $maxSimCompNC \leftarrow$ null;
4     for $t \leftarrow t_{min}$ to $t_{max}$ do
5         (conflict, $maxSimComp$) $\leftarrow$ findMaxSimComp($t$, $op$, $mAlloc$);
          if (conflict) then
              if $maxSimCompC < maxSimComp$ then
                  $maxSimCompC \leftarrow maxSimComp$;
          else
              if $maxSimCompNC < maxSimComp$ then
                  $maxSimCompNC \leftarrow maxSimComp$;
6     if ($maxSimCompNC$ != null) then
          return $maxSimCompNC$;
7     if ($maxSimCompC \geq 0$) then
          if worthUnschedule() then
              return $maxSimCompC$;
      return new component for op;
  end
```

The algorithm first gets a list of allocated components that are compatible with the operation (line 1). If the list is empty or the operation is not worth sharing (line 2, i.e. the sharing overhead is larger than the hardware area being saved), the earliest possible scheduling time t_{min} and a new component is returned, without any conflicting operations (line 3). If the operation is worth sharing and the allocated component list is not empty, the algorithm iterates through every feasible clock cycle within t_{min} and t_{max} (line 4). For each feasible clock cycle, findMaxSimComp() (line 5) returns the maximum “similarity” component, if there are components whose bound operations do not start in that cycle. This is called a *non-conflicting component*. Otherwise, findMaxSimComp() returns the maximum “similarity difference” component, which is called a *conflicting component*. In either case, the component is compared to the best component previously found, which is updated if needed. Finally, if there exists a best *non-conflicting component* (line 6, $maxSimCompNC$), it is returned to bind the operation. Otherwise, if unscheduling the conflicted operations on the *conflicting component* is worthwhile (line 7), i.e. saves more hardware area by binding the current operation than binding the conflicted operations, it is returned. If neither conditions is met, the function allocates a new component and returns it with the earliest possible scheduling time t_{min}.

The findMaxSimComp() function performs the minimum cost clique partitioning on the weighted compatibility graph to find the best resource sharing for each component. If there is more than one allocated component that the current operation can be bound to without resource conflict, the “average similarity” of each component is calculated. This is calculated by summing up the weight (w) between the current operation and all operations that have been bound to that component, and the sum is then divided by the number of currently bound operations. The component with the biggest “average similarity” is returned. If no such *non-conflicting component* can be found, the “average similarity difference” is calculated. This is calculated by the difference of “average similarity” between the conflicted operations and the current operation. The component with the biggest “average similarity difference” is returned.

Resource allocation and sharing is performed simultaneously with scheduling in HOIMS. This prevents shared operations from being scheduled at the same time. More importantly, it schedules the operations based on the sharing cost between them. Although the IMS algorithm [33] discussed in the previous chapter performs resource sharing during scheduling, it does not explore the resource sharing design space. Instead, it always schedules the operation at the earliest start time, and unschedules the conflicting operations. By using a weighted compatibility graph to trade off between different sharing possibilities, the HOIMS algorithm can generate more efficient binding than un-weighted resource sharing. Also, by scheduling based on resource sharing cost, the HOIMS algorithm requires much less unscheduling steps than the IMS algorithm. These results will be discussed in Chapter 6.

4.6 Summary

This chapter discusses an important synthesis technique used frequently in high-level synthesis: resource allocation and sharing. Resource sharing is used to reduce the overall hardware area cost. However, it is not normally performed for synthesis tools targeting FPGA implementation, due to the relatively high sharing cost of FPGAs. Appendix C carefully characterizes the area and timing overhead cost in FPGAs, and shows that the overhead cost is still much smaller compared with large circuit modules that are commonly used in computationally intensive algorithms. The close relationship between resource sharing and pipeline scheduling is also discussed. A weighted compatibility graph is proposed as a cost function to perform resource sharing exploration. The weighted compatibility graph is superior to previous non-weighted versions because it not only differentiates between multiple resource sharing possibilities, but also provides an estimation of sharing overhead cost. A detailed resource allocation and sharing algorithm that utilizes the weighted compatibility graph under the context of pipeline scheduling is proposed and discussed. This algorithm is integrated into the HOIMS algorithm, thus making it perform pipeline scheduling and resource sharing exploration simultaneously.

Chapter 5

Module Selection

Module selection is the process of choosing a specific circuit module for each operation in the dependence graph. Candidate circuit modules vary in latency, module data introduction interval, maximum operating clock frequency, area cost, etc. Performing module selection in high-level synthesis can have a large impact on system frequency, throughput and hardware area of the final circuit implementation. Unlike scheduling and resource sharing, there has been limited investigation of module selection and its role within high-level synthesis.

This work proposes a novel way to integrate module selection with pipeline scheduling and resource sharing. To do so, a large variety of pipelined circuit modules are characterized and made available for each operation, providing the synthesis tool with significant flexibility when creating a hardware architecture. Module selection can greatly improve the quality of the synthesized circuit. To our knowledge, this is the first work that proposes the use of module selection for multi-cycle, pipelined circuit modules together with the other two techniques of pipeline scheduling and resource sharing.

This chapter addresses the important issues and algorithms of module selection. It begins with an overview of module selection and an FPGA specific circuit module library. Previous work on module selection is then discussed. It then describes the inter-relationship between module selection and other techniques of pipeline scheduling and resource sharing. Finally, this chapter describes how module selection algorithms can be integrated into the HOIMS algorithm.

5.1 Module Selection Overview

Module selection is the process of choosing a specific circuit module from the circuit module library for each operation in the dependence graph. Given an adder operation in the dependence graph and two circuit modules in the library, a *ripple-carry adder* and a *carry look-ahead adder*, both circuit modules fulfill the required functionality of the operation, but with different area and clock frequency. The decision of which circuit module to select can be affected by synthesis constraints and optimization goals. For example, if clock frequency is constrained for the synthesis, circuit modules that are slower than the frequency constraint cannot be selected. If the optimization goal of the synthesis is to minimize area, then circuit modules with smaller area should be preferred. Module selection is defined formally as follows:

Definition 2 *Module matching is a mapping between an operation v of a dependence graph $G(V, E)$ and a set of compatible circuit modules M. This is represented as: $\mathcal{T}(v) : v \longrightarrow \{m_1, m_2, ..., m_n\}$, where n is the number of compatible circuit modules for v, and $m_i \in M$.*

Definition 3 *Module selection is the assignment of a particular circuit module to an operation. This is represented as: $\mathcal{M}(v) : v \longrightarrow m$, where $m \in \mathcal{T}(v)$.*

5.1.1 Module Selection in HLS

Module selection is not commonly performed in high-level synthesis. Instead, a *fixed* circuit module is assigned to each operation in the dependence graph, and remains unchanged during the architectural exploration process. For example, all multiplier operations in the dependence graph are assigned to the same *parallel multiplier* circuit module, and all adder operations are assigned to the same *ripple carry adder* circuit module. If the circuit module library has other implementation options, the fixed module assignment misses the opportunity to select the optimal circuit module and thus generates inferior synthesis results. The fixed module assignment also limits the design space of scheduling and resource sharing. Different circuit modules may have different latencies and different sharing capabilities, thus more scheduling and resource sharing possibilities can be explored. A larger design space can greatly improve the quality of high-level synthesis.

Module selection is especially important for FPGA specific HLS. Over the years, countless FPGA-specific circuit implementations have been published for performing many primary arithmetic functions and DSP kernels [45, 46, 47]. This large set of novel implementation techniques demonstrates the unique features of various FPGA families and interesting new ways to perform arithmetic when using programmable logic. These implementations vary in their frequency, latency, pipelining characteristics, and FPGA resource requirements. With all the varieties among these implementations, time-area trade offs can be exploited during the synthesis process.

5.1.2 Sample FPGA Circuit Modules

To enable the selection among multiple circuit modules, this work created a variety of circuit modules for the Xilinx Virtex4 architecture, as shown in Table 5.1. These circuit modules were created based on a variety of sources from both academic and industrial organizations. Each circuit module was characterized using a unified set of module parameters. The characterization was performed using the standard Xilinx FPGA implementation tools (XST, MAP, PAR and TRCE) included in Xilinx ISE 6 software with service pack 3. In some cases, the modules were provided with architecture-specific netlists, bypassing HDL synthesis. In order to give better estimates, the designs were synthesized multiple times using faster timing constraints to obtain better speed estimates.

Table 5.1 shows a variety of multiplier circuit modules with different implementation styles. The first three multipliers (*Array Multiplier, Booth Multiplier and CoreGen Parallel Multiplier*) use a parallel architecture (i.e. partial products are generated by the whole multiplicand and multiplier), while the other multipliers (*CoreGen Sequential Multiplier, Shift & Add Multiplier and Bit-Serial Multiplier*) use a sequential architecture (i.e. partial products are generated sequentially by part of the multiplicand and multiplier). For the parallel multipliers, even with the same level of pipelining, different implementation styles differ greatly in area cost and frequency.

Table 5.1: Sample FPGA circuit modules. The slice count (A_m) and the frequency (f_m) are characterized by the FPGA implementation tools shown in Figure A.1.

Circuit Module	λ_m	δ_m	A_m	f_m
Multipliers				
Array Multiplier 1	1	1	162	41
Array Multiplier 2	2	1	192	72
Array Multiplier 3	4	1	270	127
Array Multiplier 4	8	1	366	203
Array Multiplier 5	16	1	540	296
Booth Multiplier 1	1	1	180	69
Booth Multiplier 2	2	1	213	90
Booth Multiplier 3	4	1	305	150
Booth Multiplier 4	8	1	407	223
CoreGen Parallel 1	2	1	172	112
CoreGen Parallel 2	4	1	198	257
CoreGen Sequential	12	8	115	291
Shift & Add Multiplier	16	16	108	307
Bit-Serial Multiplier	32	32	33	401
Dedicated Multiplier 1	2	1	N/A	211
Dedicated Multiplier 2	4	1	N/A	654
Adder/Subtracters				
Ripple Carry Adder/Sub 1	1	1	9	370
Ripple Carry Adder/Sub 2	2	1	29	469
Bit-Serial Adder/Sub	16	16	27	401

For example, an *Array Multiplier* with two pipeline stages runs at 72MHz and requires 192 FPGA slices, while a *Booth Multiplier* with the same level of pipelining can run at 90MHz, but requires more hardware resources (213 slices). Sequential multipliers are smaller than the parallel versions and they run at a higher clock rate. However, their latency can be long and they normally cannot accept a new data input every clock cycle. A detailed description for all the modules in Table 5.1 is given in Appendix A.1.

Table 5.1 also shows a variety of pipelining options for the same circuit module. For example, the five *Array Multipliers* are all implemented with the same architecture, but each has a different pipelining depth. Different pipelining depths result in different hardware area costs and frequencies. Note that the relationship between area and frequency increase is not linear with respect to the levels of pipelining.

5.2 Module Selection and Pipeline Scheduling

Module selection and pipeline scheduling are closely inter-related. On one hand, module selection determines the latency of each operation, through the assignment of a particular circuit module to each operation. The latency of each operation is vital to schedule the dependence graph. On the other hand, the possible feedback constraints in pipeline scheduling place latency bounds on operations along the feedback paths, thus limiting the circuit modules that can be selected for those operations. Pipeline scheduling also limits the circuit modules that can be selected during the exploration of the pipelining design space (feasible (f, δ) pairs).

5.2.1 Module Selection and Scheduling Order

Prior work in high-level synthesis performed module selection *after* scheduling is finished. The assumption is that each operation can be finished within one clock cycle, regardless of its function and the circuit module selected for it. This assumption limits the circuit modules to only combinational ones that execute in a single clock cycle.

For multi-cycle circuit modules, scheduling before module selection is difficult. Before an operation in the dependence graph can be scheduled, the start time and latency of all its immediate predecessors must be known. The latency of an operation is determined by module selection: the latency of the operation is the latency of the module selected to implement the operation. Without module selection, the completion time of an operation is not known and data dependencies may be violated. This requires that before scheduling an operation, module selection and scheduling of all its predecessor operations be completed.

Because of this requirement, module selection for multi-cycle operations is usually performed *before* scheduling [48, 49]. A typical design space exploration strategy with this sequence is a nested loop as shown in Figure 5.1. The outer loop iterates through the module selection design space, and the inner loop explores the scheduling design space. Each module selection iteration searches among the candidate circuit modules within the library and selects a circuit module for each operation in the dependence graph. Thus the latency and finish time of each operation is known and scheduling can proceed. The inner loop tries different schedules and determines the best schedule based on the current module selection, while meeting other constraints and goals.

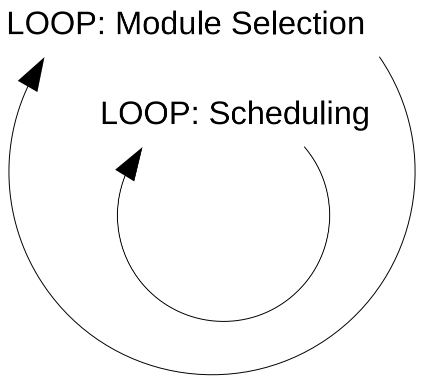


Figure 5.1: Module selection before scheduling

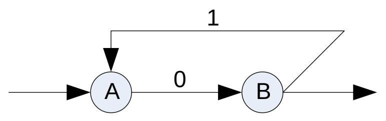
The module selection design space exploration in the outer loop can be more efficient if the scheduling inner loop can provide useful information back to the module selection process. For example, the start time difference between an ASAP (As Soon As Possible) schedule and an ALAP (As Late As Possible) schedule [10] indicates the "mobility" of an operation. If the mobility of an operation is positive, module selection for that operation can be changed to use some "longer latency but cheaper" circuit module without changing the schedule. The feedback from scheduling to module selection suggests that module selection and scheduling should be performed in an iterative way to guide the module selection design space exploration.

Some approaches perform module selection and scheduling *simultaneously* [50, 51]. While more computationally demanding, these approaches allow the scheduler to explore the impact of module selection on the circuit schedule within a larger design space. Performing module selection during scheduling explores a large set of implementation possibilities and allows the scheduler to balance the area cost of each operation implementation with the importance of the operation within the schedule.

5.2.2 Feedback Constraint and Module Selection

Feedback constraints limit the module selection possibilities. As discussed in Chapter 3, feedback constraints are represented as cycles in the dependence graph. According to Equation 3.9, the accumulated latency of any elementary cycle cannot exceed the cycle's distance multiplied by the system data introduction interval. The latency of a cycle is the accumulated latency of the selected circuit modules for each operation inside the cycle. This latency constraint imposes limitations on the module selection of operations in each cycle.

The feedback constraint limitation applies to the *combination* of module selections for all operations in an elementary cycle, not to the module selection for a *single* operation in that cycle. Figure 5.2 shows an example of this limitation. The feedback edge from operation B to operation A has a distance of 1, while the feed-forward edge from A to B has zero distance. If the system data introduction interval (δ) is 3, then the cycle latency of the cycle cannot be greater than $D * \delta = 3$. Both operations have the same compatible circuit modules as shown in part (b) of Figure 5.2. The feedback constraint does not limit the module selection for either operation. However, the combination of module selection which makes the cycle latency exceed 3, i.e. $A \rightarrow M_2$ and $B \rightarrow M_2$ is not permitted.



Operation	*Circuit Module*	*Latency*
A/B	M_1	1
A/B	M_2	2

(a) (b)

Figure 5.2: Illustration of feedback constraint limits cycle's module selection

5.2.3 Module Selection and Pipelining Design Space

In the context of pipeline scheduling, module selection has a direct impact on the pipelining design space. The maximum system frequency (f_{max}) is determined by the module selection. A computationally intensive circuit is resource dominant, which means its maximum frequency is determined by the slowest circuit module in the circuit. If there are M circuit modules that can implement the function of operation n, and there are N operations in the dependence graph, the maximum system frequency is:

$$f_{max} = \min\left(f_{max}(n)\right), f_{max}(n) = \max\left(f_i(n)\right) \text{ for } 1 \leq n \leq N, 1 \leq i \leq M, \tag{5.1}$$

where $f_i(n)$ is the frequency of each compatible circuit module for operation n, and $f_{max}(n)$ is the maximum frequency among these modules. According to Equation 3.2, module selection determines the upper bound of the pipelining design space.

For dependence graphs with feedback edges, each elementary cycle's latency is determined by the module selection. The minimum cycle latency for each cycle (c) is:

$$Delay_{min}(c) = \sum \lambda_{min}(n) \; \forall n \in c, \tag{5.2}$$

where $\lambda_{min}(n)$ is the minimum latency among all compatible circuit modules for operation n. Thus the minimum system data introduction interval is:

$$\delta_{min} = \max_{c \in C} \left\lceil \frac{Delay_{min}(c)}{Distance(c)} \right\rceil . \tag{5.3}$$

Thus module selection determines the lower bound of the pipelining design space for dependence graphs with feedback edges. Note that for feed-forward only dependence graphs, δ_{min} is always 1.

Each (f,δ) pair in the pipeline scheduling design space also has a direct impact on the module selection. As discussed in the previous chapter, the data introduction interval of any circuit module allocated for a pipeline must be less than or equal to the system data introduction interval. Also, each circuit module cannot be slower than the system clock frequency. Thus, when pipeline scheduling is performed at each (f, δ) pair, module selection for each operation must conform to the following two criteria:

$$\delta_m \leq \delta \text{ , and} \tag{5.4}$$

$$f_m \geq f. \tag{5.5}$$

This indicates that at each (f,δ) pair of the pipeline scheduling design space, module selection candidates are limited and the design space of module selection might change.

5.3 Module Selection and Resource Sharing

Module selection and resource sharing are inter-related. If two or more compatible operations are assigned to different circuit modules, they cannot be shared. If two or more operations are shared, they must be assigned to the same circuit module. To share operations with different bit-widths, the *same* circuit module should be instantiated with a bit-width configuration that covers the bit-width requirement of all shared operations. For example, to share a 4x4 and an 8x8 multiplier on a *Booth Multiplier* circuit module, the circuit module must be instantiated with an 8x8 bit-width configuration.

Module selection affects the amount of resource sharing. Selecting more specific circuit modules for each operation will decrease the possibility of resource sharing between operations. Selecting more generic modules for each operation will increase the possibility of resource sharing. For example, if the four multiplier operations in the Biquad filter (see Figure 3.4) are all assigned to the same circuit module such as the *Array Multiplier 1* in Table 5.1, and the circuit module is configured to cover the bit-width requirement for all of the four operations, it is more likely that the four operations will be shared in the resource sharing process. However, if module selection assigns a different circuit module to each multiplier operation, the resource sharing algorithm will not be able to share them even though each operation has a unique start time.

With resource sharing, the average cost of a circuit module is no longer the "actual" hardware area of the module itself. Instead, it should include the resource sharing overhead hardware cost, and be divided by the number of operations that can share this module. The *amortized cost* of a circuit module is defined as:

$$A'_m = \frac{A_m + A_{share}}{\text{share}_{max}}, \tag{5.6}$$

where A_m is the hardware cost of the circuit module itself, A_{share} is the hardware cost for resource sharing this module and $share_{max}$ is the largest number of operations that can share this module as defined in Equation 4.1.

The amortized cost of a circuit module affects the module selection algorithm described in the previous chapter. The selection of the lowest circuit module area (A_m) discourages sharing between operations, while the selection of A'_m encourages the sharing between operations.

5.4 Previous Module Selection Work

Unlike pipeline scheduling and resource sharing, there has been limited investigation of module selection and its role within architectural exploration. Several projects have included module selection during high-level synthesis but their scope is relatively limited. A major contribution of this work is integrating module selection of multi-cycle, pipelined circuit modules with the other two high-level synthesis techniques of pipeline scheduling and resource sharing.

Thomas and Leive considered the first module selection problem [52] and opened the door to further research on the subject. In that paper, they proposed a solution to the *general* module selection problem for non-pipelined designs where module selection is done after scheduling and allocation. Each module is evaluated by an individual optimization function, from which the best module is selected for each operation type. The optimization function is a function of weighted area, delay and power of the operation. However, optimizing every operation type doesn't necessarily lead to the globally optimal solution.

The Schwa project was augmented to support module selection for pipeline scheduling in [49]. In that paper, Jain proposed a rigorous technique for module selection for pipelined designs. This technique is based on the ability to predict the location in the design space of the area-time trade-off curve for a given design and a given module set. This predictive ability, in turn, is based on the straightforward optimization criteria for digital design that all modules are utilized as many cycles as possible. This work was further augmented to support multi-cycle circuit modules during scheduling in [53] and later constrained pipeline scheduling using an ILP [54]. All the module selection techniques in these papers are based on operation type (i.e. all operations of the same type are assigned to the same circuit module) instead of each operation, and the selection criteria is minimum area-time product.

Bakshi and Gajski proposed an iterative module selection algorithm in [55] to optimize the hardware area for pipelined circuits. In this paper, a CF (Commonality Factor) function based on the topology of the circuit is used to calculate the priority of each operation for area reduction. The algorithm starts with the fastest component and iteratively "slowsdown" the highest priority operations until there is no area improvement. The underlying premise of this paper is that slow, inexpensive components should be used for non-critical paths, while fast and expensive components should be used for critical paths. This paper is later extended in [56] to incorporate multi-cycle but non-pipelined circuit modules.

All the papers discussed above use some form of heuristics to find a near-optimal module selection solution. Other have used an exhaustive approach to find the optimal solution. In [57], module binding and module selection problems are solved concurrently for non-pipelined designs using a mixed inter-linear programming (MILP) technique. Shen and Jong formulated the module selection as a multi-objective optimization problem in [48]. They proposed a branch and bound algorithm for simultaneously optimizing power, delay and area globally. The basic drawback of these exhaustive techniques is that their computational complexity are exponential, so they are not practical for realistic examples.

5.5 HOIMS with Module Selection

The HOIMS algorithm is expanded with module selection to explore a larger pipeline synthesis design space. Without module selection, a fixed circuit module is assigned to each operation in the dependence graph. In this case, the exploration of the pipelining design space is limited because a fixed circuit module may only be operational at certain pipelining design points. Resource sharing is also limited with a fixed circuit module assignment because different circuit modules may have different sharing capabilities, which might generate different circuit architectures with different hardware costs. The integration of module selection in the HOIMS algorithm can explore more scheduling and sharing possibilities. Such an enlarged design space can greatly improve the synthesis quality of the HOIMS algorithm.

Module selection and its exploration has three major steps in HOIMS. The first step generates the set of candidate circuit modules for each operation. This set determines the size of the module selection design space. The second step picks an initial module selection as the starting point of the module selection exploration. Pipeline scheduling and resource sharing are then performed. The third step modifies the previous module selection, and performs another iteration of scheduling and resource sharing. The third step is repeated until no module selection improvement is feasible.

The modified HOIMS algorithm that includes module selection is illustrated in Algorithm 5.1. The pipeline scheduling with resource sharing remains the same as in Algorithm 3.2. However, various module selection related algorithms are integrated into different places of the previous HOIMS algorithm. Before the pipelining design space is explored, a candidate circuit modules set is determined for each operation based on the input library of circuit modules and the synthesis constraints (line 1). At each pipelining design point (f, δ), an initial module selection is performed (line 8) based on a non-dominated module set (line 7). The initial module selection is corrected if necessary (line 9) before the scheduling is started. The loop at line 11 first performs pipeline scheduling and resource sharing based on the current module selection, then it updates the module selection and the loop continues. The following sections will discuss these module selection and exploration algorithms in detail.

Algorithm 5.1: HOIMS with module selection

```
HOIMS($DG$, $T$, $LIB$) begin
1     CandidateModules($T$, $LIB$);
2     $\delta_{min} \leftarrow$ MinII();
3     $\delta_{max} \leftarrow$ MaxII();
4     $S_{best} \leftarrow \emptyset$;
5     for $\delta \leftarrow \delta_{min}$ to $\delta_{max}$ do
6         $f_\delta \leftarrow \delta \cdot T$;
7         RemoveDominated($\delta, f_\delta$);
8         InitialMS($\delta$,$f_\delta$);
9         CorrectMS($DG$);
10        CreateWCG();
11        repeat
12            Initialize all operations to be never scheduled;
13            HeightR($\delta$);
14            Insert all operations into $Q$;
15            Schedule($START$, $0$);
16            while $Q \neq \emptyset$ do
17                $v \leftarrow$ HighestPriorityOperation($Q$);
18                $t_{s_{min}} \leftarrow$ CalculateEarlyStart($v$);
19                $t_{s_{max}} \leftarrow t_{s_{min}} + \delta - 1$;
20                $(t_s, m) \leftarrow$ DynamicAllocate($v$,$T_{s_{min}}$,$T_{s_{min}}$);
21                Schedule($v$, $t_s$, $M$);
22                $Q \leftarrow Q +$ UnscheduleConflicts($v$, $t_s$, $M$);
23            if Cost($S_{current}$) $<$ Cost($S_{best}$) then
                  $S_{best} \leftarrow S_{current}$;
          until (not MS_Improvable()) ;
end
```

5.5.1 Candidate Module Set Generation

The candidate module set is a list of circuit modules for an operation in the dependence graph that are both functionally compatible with the operation and performance compatible with the throughput constraint. A circuit module is functionally compatible with an operation if it can perform the function of that operation. For example, an ALU circuit module is compatible with an "add" or "sub" operation because it can perform either operation's function. A circuit module is performance compatible with the throughput constraint if the throughput of the circuit module is greater than or equal to the system throughput constraint:

$$\frac{f_m}{\delta_m} \geq T, \tag{5.7}$$

where f_m is the module frequency, δ_m is the module data introduction interval and T is the throughput constraint. There must be at least one circuit module in the candidate module set for each operation in the dependence graph. Otherwise, there will be no feasible circuit module to implement that operation, thus the dependence graph cannot be implemented under the throughput constraint.

The candidate module set for an operation is illustrated in Figure 5.3. This figure uses the same axis as Figure 3.3 for the pipelining design space. The round dots represent the pipelining design space for a throughput constraint of 50 MSample/Sec. The small rectangles represent all the compatible circuit modules for this operation in the library. The location of each circuit module in this figure is in accordance with the module's operating frequency (f_m) and module data introduction interval (δ_m). Thus only the circuit modules above the (f, δ) dots have a module throughput larger than the system throughput constraint, so M_1, M_2, M_3 and M_4 forms the candidate module set for this operation.

The candidate module set should be further filtered at each pipelining design point (f, δ). According to Equation 5.5, only circuit modules on the left and upper side of (f, δ) can be used for that pipelining option. For example, when the system

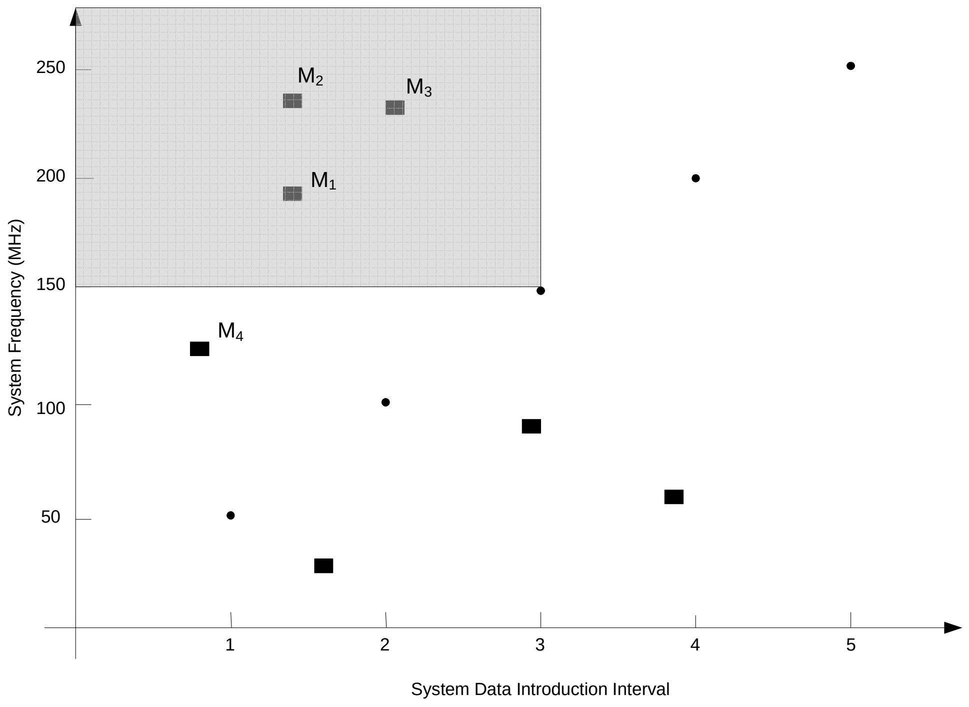


Figure 5.3: Example of candidate module set

data introduction interval is 3 in Figure 5.3, only the circuit modules inside the gray area can be used for the (150MHz, 3) point.

Some circuit modules might be *dominated* by other modules. For example, M_1 and M_2 are two circuit modules in the candidate module set for an operation at (150MHz, 3) in Figure 5.3. Module M_1 and M_2 have the same module data introduction interval (i.e. same sharing capability) and latency, but the area of M_2 is larger than the area of M_1. In this case, module M_2 is dominated by module M_1, because selecting module M_2 always results in larger circuit area than when module M_1 is used. Dominated circuit modules can be removed from the candidate module set to reduce the module selection design space.

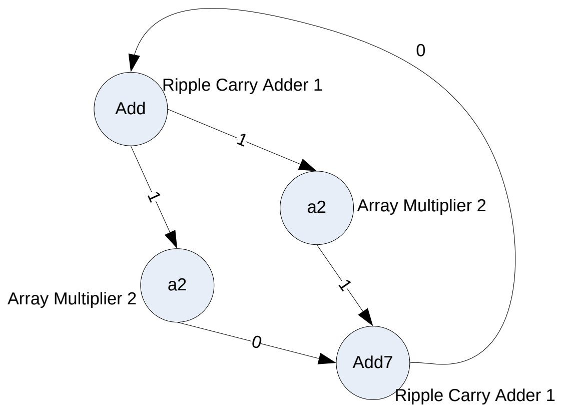
5.5.2 Initial Module Selection and Correction

After the non-dominated candidate circuit module set is determined at each pipelining design point, the module selection design space should be explored to find the optimal module selection that generates the minimum hardware area FPGA implementation. However, enumerating all module selection possibilities is an exponential search process as discussed in Section 2.6. The module selection exploration strategy employed in HOIMS starts with an initial selection and iteratively refines it.

For each operation in the dependence graph, the initial module selection assigns the *minimum area* circuit module among its candidate module set to that operation. The selected circuit module is also configured to have the same bit-width as the operation. The initial assignment thus creates the most *specific* module selection for the dependence graph as discussed in Section 5.3. However, if there is a feedback constraint, the initial module selection might not be valid because it may prevent the dependence graph from being schedulable with the current system data introduction interval value.

As discussed in Section 3.2.3 of the pipeline scheduling chapter, the *minDist* matrix can be used to determine the pipeline schedulability of a dependence graph with feedback edges. For example, the Biquad filter dependence graph (see Figure 3.4) contains one SCC which is composed of operation *Add*, *a2*, *Add7*, and *a3*. The SCC is reproduced in Figure 5.4 part (a) for convenience. A sample module selection for operations inside this SCC is also shown as the text beside each operation. The *minDist* matrix for this SCC with this module selection is shown as part (b) of Figure 5.4, assuming a system data introduction interval of 4. Notice the diagonal matrix elements for operation *Add*, *a2* and *Add7* are all positive, which means the sample module selection will result in an infeasible pipeline scheduling.

The function CorrectMS() (line 9) in Algorithm 5.1 checks the schedulability of the current module selection and modifies it until the graph is schedulable or no further modification is possible. The algorithm for correcting the initial module selection is illustrated in Algorithm 5.2. For each SCC of the dependence graph, it first creates a list of modules for each operation in the SCC, the latency of each module in the list must be smaller than the currently selected module (line 1). These modules are sorted from lowest δ_a to highest δ_a (δ_a is the relative area increment per latency decrease, as shown in Equation 5.8). The purpose of this sorting is to



(a) SCC of Biquad filter with module selection

	Add	a2	a3	Add7
Add	2	-3	-3	1
A2	5	2	2	4
A3	1	-2	-2	0
Add7	1	-2	-2	0

(b) *minDist* matrix with δ = 4

Figure 5.4: Module selection for the Biquad filter's SCC and the corresponding *minDist* matrix

reduce the latency of the operation with minimum area increase. For example, if two circuit modules have the same latency and both are smaller than the current module's latency, then the module with smaller area cost will be selected as a replacement of the current module:

$$\delta_a = \frac{A_{new} - A_{old}}{\lambda_{m_{old}} - \lambda_{m_{new}}}. \tag{5.8}$$

Algorithm 5.1 then checks the schedulability of the current SCC (line 3). If it is schedulable, the current module selection for this SCC is valid (line 4), so the outer-most loop continues and the next SCC is processed. If not, the operations with positive slack are recorded and sorted based on descending slack value (line 5). The next loop iterates through these operations and tries to find the first operation whose latency can be reduced (line 6). If none of the operations have a shorter latency module, the current module selection for this SCC cannot be further modified for a valid schedule, so the function returns false (line 7). If a shorter latency module is found for an operation, it is set as the current module selection for that operation, and the schedulability check loop is repeated (line 2).

Algorithm 5.2: Correcting the Initial Module Selection

```
  boolean correctMS($DG$) begin
      foreach $scc$ of $DG$ do
1         setup smaller latency modules for each $v \in scc$;
          $validMS \leftarrow$ false;
2         while not $validMS$ do
3             $(slack, posSlackOps) \leftarrow$ minDist($scc$);
              if $slack \leq 0$ then
4                 $validMS \leftarrow$ true;
              else
5                 sort $posSlackOps$;
                  $update \leftarrow$ false;
                  foreach $v \in posSlackOps$ do
6                     if $setSmallerLatencyModule(v)$ then
                          $update \leftarrow$ true;
                          break;
                  if not $update$ then
                      break;
7         if not $validMS$ then
              return false;
      return true;
  end
```

5.5.3 Module Selection Refinement

After a schedulable module selection is obtained, pipeline scheduling and resource sharing are explored for the dependence graph. The module selection is then refined based on the current scheduling and sharing result. Another iteration of scheduling and sharing is then performed based on the modified module selection. This refinement is repeated until no module selection improvement can generate a better hardware implementation. This iterative module selection exploration includes two parts: local module selection refinement and global module selection refinement.

The local module selection refinement tries to improve the module utilization efficiency for each operation. For operations that are not shared, their module selections are updated with the circuit module which has the lowest area cost. For operations shared with others, their module selections are updated with the circuit module which has the lowest amortized area cost (see Equation 5.6).

The global module selection refinement tries to improve the module utilization efficiency between operations. Algorithm 5.3 illustrates the outline for this global refinement. All circuit module instances that are not fully utilized are collected first in line 1. The refinement is performed for each operation in the dependence graph that is worth sharing (see line 2 and 3). The algorithm then iterates through the module instances (line 4). If the module type of an instance is the same as the operation's current module selection, or if the instance is fully utilized or free, it is skipped (see line 5). Otherwise, the feasibility of the module instance is checked for this operation (line 6). The module instance is feasible if it is both compatible (i.e. in the operation's candidate module set) and changeable (i.e. the latency of the module will not make the SCC this operation belongs to unschedulable). A feasible module instance results in the module selection change for this operation (line 7). The utilization status of the current and new module instance are also updated (line 8).

Algorithm 5.3: Global Module Selection Refinement

```
  globalMSRefine() begin
1     initialize $mi2free[]$;
2     foreach $op \in DG$ do
3         if notWorthShare($op$) then
              continue; ;
4         foreach $mi \in mi2free[]$ do
5             if moduleType($mi$)=MS($op$) or fullOrEmpty($mi$) then
                  continue;
6             if compatible($op, mi$) and changeable($op, mi$) then
7                 setMS($op, mi$);
8                 update $mi2free[]$ for current and new module of $op$;
                  break;
  end
```

5.5.4 Bit Width Morphing

To generate the most area efficient implementation for a bit-accurate computational algorithm, the bit width of the circuit module to which an operation is bound must be equal or larger than the operation's bit width. Unlike the previous two algorithms, which assume a single bit width that covers all operations in the dependence graph, the HOIMS algorithm selects the most efficient bit width circuit module for each operation. It also morphs the operation's preferred bit width for best resource sharing between operations with different bit widths.

The initial module selection not only picks the cheapest circuit module for each operation, it also sets the bit width of the circuit module to match the bit width of the operation. This avoids the waste of unused extra bits for operations with wider circuit modules. However, mapping an operation to a wider circuit module may decrease the overall area due to resource sharing. When searching for compatible module instances (see line 1 in Algorithm 4.1), circuit modules with bit width $w_{compatible}$ are returned. Equation 5.9 shows the method for calculating the compatible bit width. This ensures that the area of the multiplexer needed for sharing will not exceed the module area for the operation's current bit width:

$$MuxArea_{w_{compatible}} \leq ModuleArea_{w_{current}}. \tag{5.9}$$

The preferred bit width of an operation is morphed during the module selection improvement step, based on the current schedule (s). Algorithm 5.4 illustrates the steps for bit width morphing with the current schedule (s). The algorithm iterates through each shared circuit module in the current schedule (line 1). For each shared module m_s, all its allocated circuit module instances ($m_s.mi$) are sorted in descending order according to their bit width (line 2). A map from each module instance to the number of free slots ($mi2free[]$) is initialized based on the current schedule (line 3). Then the algorithm goes through each instance (mi_1) and tries to increase its utilization ratio (line 4). If the module instance is fully utilized, the next instance is checked. Otherwise, the next width compatible module instances mi_2 are explored for possible bit width morphing (line 5). Each operation bound to mi_2 is checked (line 6). If bit width morphing is set for an operation *op*, the corresponding entry in $mi2free[]$ is updated (line 7). This algorithm sets the preferred bit-width of an operation to its closest bit-width module instance which is under-utilized.

Algorithm 5.4: Bit Width Morphing

```
  bitWidthMorph($s$) begin
1     foreach $m_s \in s.sharedModule()$ do
2         sort($m_s.mi$);
3         initialize $mi2free[]$;
4         foreach $mi_1 \in m_s.mi$ excluding the last one do
              if $mi2free[mi_1].full()$ then
                  continue;
              $mi_2 \leftarrow mi_1$.next();
5             while not $mi2free[mi_2].empty()$ do
                  if $mi_2.width() == mi_1.width()$ then
                      $mi_2 \leftarrow mi_2$.next();
                      continue;
                  if $mi_2.width() < mi_1.width()$ - $w_{compatible}$ then
                      break;
                  $bOps \leftarrow mi_2$.boundOps();
                  sort($bOps$);
6                 foreach $op \in bOps$ do
                      if $op.width() < mi_1.width()$ - $w_{compatible}$ then
                          break;
                      bit width morph $op$ with $mi_1$.width();
7                     update $mi2free[mi_1]$ and $mi2free[mi_2]$;
                  $mi_2 \leftarrow mi_2$.next();
  end
```

The bit width morphing information obtained in Algorithm 5.4 is used in the next iteration of pipeline scheduling. When a new module instance is created for an operation, the bit width of the new instance prefers the morphed width instead of the operation's native bit width. This creates a more *general* resource that is compatible with more operations, and encourages resource sharing between operations with similar bit-widths.

Integrating module selection with pipeline scheduling and resource sharing completes the HOIMS algorithm. Module selection adds another dimension to the whole pipeline synthesis design space and can significantly improve the quality of the synthesized circuit, as will be shown in the next chapter. However, the exponential design space of module selection makes an exhaustive search impractical. The iterative module selection refinement algorithm proposed in this chapter makes the exploration of the module selection design space more efficient.

5.6 Summary

This chapter describes a very important yet seldom applied technique for high-level synthesis: module selection. Many FPGA specific circuit modules have been proposed for implementing various computation tasks over the years. Module selection leverages these implementation varieties to synthesize a minimum overall area FPGA implementation. Selection between multi-cycle and pipelined circuit modules within the context of pipeline scheduling is novel compared with previous module selection work. Because of the exponential nature of the module selection design space, an iterative module selection refinement algorithm is proposed in this chapter. It is based on the close inter-relationship between module selection and the other two synthesis techniques of pipeline scheduling and resource sharing. The goal of this module selection refinement algorithm is to efficiently explore the huge pipeline synthesis design space while obtaining a good synthesis result. The next chapter will present and discuss experimental results for the combined algorithm.

Chapter 6

Experimental Results

This work is the first to perform concurrent pipeline scheduling, resource sharing and module selection. Most previous work combines just two of the three techniques, assuming the third one is performed independently. Combining the three techniques together creates a much larger design space than traditional non-combined approaches and significantly improves the quality of the synthesized circuit.

However, combining the three techniques is very difficult for two reasons. The first reason is the close inter-relationship between the three techniques as discussed in the previous two chapters. This inter-relationship makes the exploration sequence of these techniques in the combined algorithm difficult to manage. The second reason is the size of design space exposed by the combination of these techniques. Each technique alone has an exponential design space as discussed in previous chapters, and a combined approach makes the design space even larger. This work demonstrates that it is feasible to efficiently explore such a large design space and generate close to optimal results using novel algorithms.

This chapter presents and discusses the experimental results of three combined algorithms, the *ASAP exploration* algorithm, the *IMS exploration* algorithm, and the HOIMS algorithm proposed in the previous chapters. It begins with an analysis of the pipeline synthesis design space generated by appyling module selection only, resource sharing only, and combined module selection and resource sharing, within the context of pipeline scheduling. It then illustrates and analyzes the pipeline synthesis area results from the first two algorithms for several sample computational kernels. The results from the combined approach are compared quantitatively to the results from traditional non-combined approaches.

This chapter then presents and discusses the area results from the HOIMS algorithm, which explores a bigger resource sharing design space using weighted compatibility graph, and a bigger module selection design space for non-uniform bit-width dependence graphs. Finally, the computational complexity and runtime of these algorithms are presented and discussed. This chapter shows that HOIMS is able to find better solutions than the two previous algorithms and do it efficiently.

6.1 Pipeline Synthesis Design Space Analysis

The design space of module selection or resource sharing or a combination of these two techniques within the context of pipeline scheduling can be best visualized with a two dimensional diagram as shown in Figure 6.1. The y-axis of the diagram represents the estimated area of the synthesized circuit (in FPGA slices). The x-axis represents unique (f, δ) design points of the pipeline design space, ordered by increasing values of δ and corresponding f. All design points meet the system throughput constraint (i.e. $f = \delta * T$).

Several hypothetical lines are shown on this figure to demonstrate the impact of module selection and resource sharing on the design space. The thin solid line represents the design space associated with module selection only. The dotted lines represent design spaces associated with resource sharing only. These dotted lines are representative of most previous work where scheduling is performed without the exploration of module selection design space (i.e. assuming a fixed module selection). The thick solid line represents the design space for the combined module selection and resource sharing exploration strategy. The design space associated with each of these lines will be discussed in detail below.

6.1.1 Module Selection Only

The *module selection only* line represents a fictitious design space in which the lowest cost circuit module is chosen for each operation in the dependence graph at each (f, δ) design point. No resource sharing between operations is performed. Changes in the area cost of synthesized architecture are associated only with changes

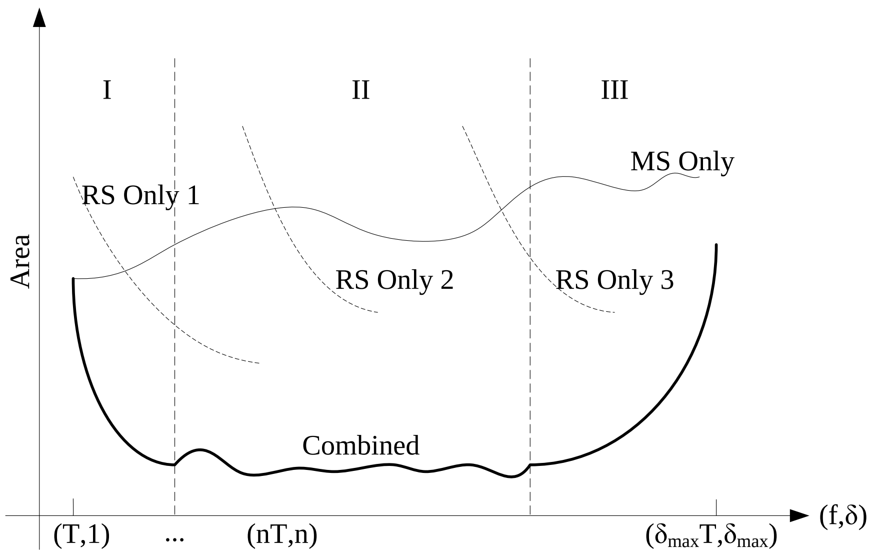


Figure 6.1: Projected pipeline synthesis design space with module selection and/or resource sharing under a fixed throughput constraint

in the corresponding module selection that is possible at each (f, δ). The actual shape of the line will depend on the circuit module library and the operations within the dependence graph.

The area cost of the module selection only method will change up or down along the (f, δ) design points of the x-axis. In some cases, the cost will increase as the frequency increases, because more deeply pipelined, faster, and more expensive circuit modules are necessary. In other cases, the cost will decrease as δ increases because a less expensive, sequential circuit module (i.e. larger δ_m) may be usable at the higher values of δ.

6.1.2 Resource Sharing Only

The *resource sharing only* method assumes a fixed module selection for each operation in the dependence graph and maximally shares the module instances between operations. Three dotted lines shown in Figure 6.1 represent three examples of the design space associated with the resource sharing only method. Each line represents the design space associated with a unique module selection. A typical module selection in the resource sharing only method binds all operations of the same type to the same circuit module. Thus, the difference in the area cost between the three lines is due solely to the difference in their module selection and the resource sharing capability of that module selection.

As Figure 6.1 shows, the design space of the resource sharing only method does not cover the full range of (f, δ) values. Because the selected modules will only operate within a fixed frequency limit, the upper bound of the design space is thus limited by the slowest module in the module selection. The lower bound of the design space is limited by the minimum feasible system data introduction interval with that module selection. For dependence graphs without feedback, this is equal to the maximum module data introduction interval (δ_m) among all operations for that module selection. For graphs with feedback, it is also affected by loop latency and loop distance (see Equation 3.10). Note that with a fixed module selection, the breadth of the design space can be very limited.

Within the limited design space, these sample resource sharing only lines show that the cost of the circuit decreases with larger values of δ. The decreasing area is due to the ability to increase the sharing of the allocated modules among operations (see Equation 4.1). The amount of increased sharability also depends on the module selection assumed. More resource sharing can be obtained with increased δ if more general circuit modules are used than if more specific circuit modules are used.

6.1.3 Combined Module Selection and Resource Sharing

The third design space represents combined module selection and resource sharing (thick solid line). This space represents the best architecture identified when both techniques are used during architecture exploration. As shown in Figure 6.1, this combined design space should be similar to a "bathtub" curve and can be divided into three stages.

The first stage (stage I) occurs at small values of δ. The small values of δ in this stage limit the ability to exploit resource sharing (see Equation 4.1). In the extreme case where δ=1, no resource sharing is possible[1] and the area cost of the

[1]Resource sharing between mutually exclusive paths is still possible when δ=1.

combined technique will be the same as the module selection only technique. As δ increases in this stage, the area cost of the architecture decreases due to the benefits of resource sharing. As described earlier, increasing (f, δ) may require the use of more expensive circuit modules, which might offset some of the benefits by doing resource sharing. In stage I, however, the benefits of resources sharing normally overcome the increasing cost of more expensive module selection, thus generating a lower area cost solution with increasing f and δ.

Stage III represents the other extreme of the design space with very large values of δ and very high operating frequencies. At these high frequencies, only deeply pipelined or sequential circuit modules can operate. If deeply pipelined modules are selected, the advantages of large resource sharing capability might not justify the high cost of these modules. If sequential modules are used, their low area cost might not justify the limited resource sharing capability due to their larger module data introduction interval (see Equation 4.1). In either case, inefficient circuit architectures will be generated.

Stage II represents the interesting design space where the trade-offs between module selection and resource sharing can be explored and an optimal synthesis results can be obtained. Module selection between *specific* (less resource sharing) and *general* (more resource sharing) can be explored with the feedback from resource sharing. The architecture of the dependence graph and the operation variety have a big impact on the trade-off decision. In some cases, a more specific module selection is favored due to the difference between operations and limited sharing possibility because of dependence constraints. In other cases, a more general module selection might be favored due to the similarity between operations and good sharing possibility. This stage is called the *valley* which contains a slowly changing design space and the optimal design points.

6.1.4 Pipeline Synthesis Design Space Summary

This section shows that combining module selection and resource sharing within the context of pipeline scheduling creates a much larger design space than traditional non-combined approaches. First, traditional pipeline synthesis approaches only explore a fixed frequency with a fixed data introduction interval. Instead, this work explores all feasible (f, δ) pairs, which not only creates a much larger pipelining design space, but a larger module selection and resource sharing design space as well.

Second, during the exploration of the pipelining design space, module selection and resource sharing techniques can be used to largely reduce the area cost of the synthesized result. However, *module selection only* just explores the module selection design space without the sharing possibilities among operations, and *resource sharing only* just explores the sharing design space without the alternative module selection possibilities for operations. Integrating these two techniques creates a larger design space and finds better solutions than when either technique is applied alone.

6.2 Pipeline Synthesis Area Results

The next two sections will illustrate and analyze the area results from three combined algorithms, the *ASAP Exploration* algorithm [58], the *IMS Exploration* algorithm [59], and the HOIMS algorithm discussed in previous chapters. These three algorithms were developed successively during this research of integrating module selection, resource sharing and pipeline scheduling to synthesize area efficient pipelined circuits for computationally intensive algorithms.

The *ASAP Exploration* algorithm performs pipeline scheduling, module selection, and resource sharing concurrently. It is a recursive branch and bound algorithm that explores every module selection and resource sharing possibilities. This scheduler is a relatively simple adaptation of a non-pipelined ASAP list scheduler. The objective of this algorithm is to identify the lowest area cost architectural solution that meets the constraint of a user specified throughput. The details of the algorithm are summarized in Algorithm D.1 of Appendix D.

The *IMS Exploration* algorithm combines module selection with a backtracking scheduler based on IMS [33]. IMS performs pipeline scheduling and resource sharing simultaneously. The scheduler is iterative and allows backtracking (unschedule and reschedule) to find solutions that are otherwise not attainable by non-backtracking scheduling approaches. This algorithm uses a heuristic to significantly reduce the module selection design space and obtains sub-optimal results. The details of the algorithm are summarized in Algorithm E.1 of Appendix E.

There are two main purposes of this section. The first is to demonstrate quantitatively the advantage of a combined approach than traditional non-combined approaches. Comparing the results with previous work is difficult because of limited research on throughput constrained synthesis and different circuit modules used in synthesis. However, synthesis under fixed frequency and data introduction interval, the *resource sharing only* approach, and the *module selection only* approach are representative of previous work, thus will be compared with the results from the combined approach. The second purpose is to show the relationship between throughput constraint and the resulting pipeline synthesis design space.

A number of signal processing kernels were tested. The detailed test setup and results are described in Appendix D and E. Although these results are obtained from the *ASAP Exploration* algorithm and the *IMS Exploration* algorithm, the HOIMS algorithm generates similar results. This section illustrates some of these test results to show the important characteristics of the design space, and to demonstrate the benefits of the combined approach. Although these test-cases are relatively small compared to real life applications, their results are easier to be analyzed to demonstrate the advantages and disadvantages of these algorithms.

6.2.1 Area Results for a Single Throughput Constraint

Figure 6.2 illustrates results of the *ASAP Exploration* algorithm for the IDCT example. In this figure, the area of the combined resource sharing and module selection is shown as bars, which is segmented into the circuit module area (colored as gray) and resource sharing area overhead (colored as dark). The solid line shows the area cost with module selection only method. The two dotted lines represent the area cost of two different resource sharing only methods.

As Figure 6.2 shows, the actual pipeline synthesis design space of the IDCT example has a similar shape to the projected design space described in Figure 6.1.

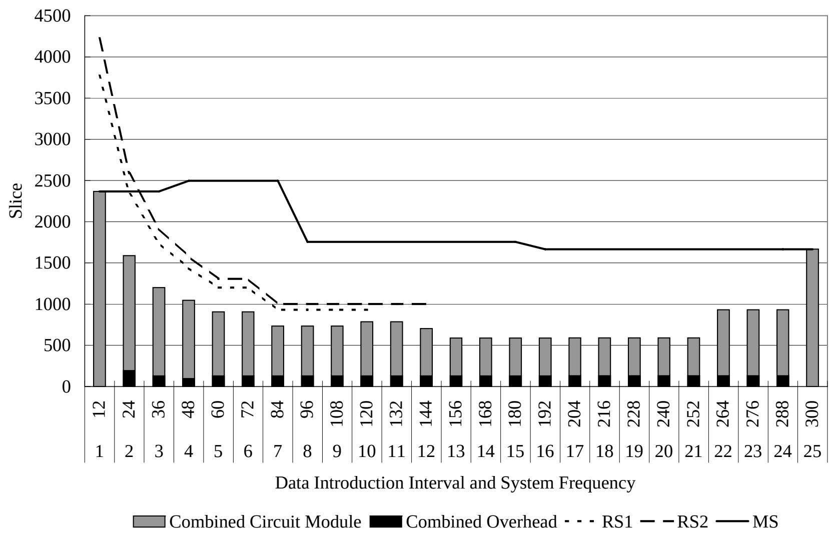


Figure 6.2: *ASAP Exploration* area result with different synthesis techniques for the IDCT example under a 12 MSamples/Sec throughput constraint

The figures for other kernels (see Appendix D) have the same shape with different scales, they are omitted here for brevity. At the left of the curve, increases in δ will reduce the cost due to the benefits of resource sharing. In the middle of the curve, the area costs are all similar (δ=13 to 21). In this stage, one *CoreGen Parallel 2* multiplier module is allocated and shared among all multiplier operations in the dependence graph. The design point at (156 MHz, 13) represents the lowest cost within the entire design space. At the right of the curve, (δ=22 to 25), a more expensive multiplier (*Array Multiplier 5*) is allocated and shared, yielding higher area cost than those in stage II.

The best design results are found when both resource sharing and module selection are applied. In this case, the algorithm is able to apply *both* techniques to provide a superior result than the use of either technique alone. In many cases, the advantages of both techniques are combined and applied during the scheduling and allocation of the pipeline. For the IDCT example, the average hardware area within the pipelining design space obtained with the combined approach is 43% smaller than when only resource sharing is applied, and 53% smaller than when only module selection is applied. Table 6.1 shows the average area comparison between different synthesis techniques for various test-cases. These results suggest that module selection and resource sharing are complementary and can be used together to identify lower cost circuit pipelines.

Most previous HLS research only explore the design space with a fixed clock frequency. For the IDCT example, as Figure 6.2 shows, the largest area solution ($\delta = 1$) is 3 times bigger than the smallest area solution ($\delta = 13$), with the combined approach. When only module selection is applied, the biggest area solution is 50% bigger than the smallest area solution. When only resource sharing is applied, the biggest area solution is about 3 times bigger than the smallest area solution on average. Thus, exploring a bigger pipelining design space with different clock frequencies can significantly reduce the hardware area of the synthesized circuit.

Table 6.1: Average area comparison between different synthesis techniques.

Circuit	Combined	RS1	RS2	MS
Color Space Conversion	340.32	492.5 (31%)	519.67 (35%)	561.84 (39%)
FIR	488.3	833.7 (41%)	842.9 (42%)	1078.6 (55%)
FFT	641.3	986.7 (35%)	995.9 (36%)	1231.7 (48%)
Linear Interpolator	562.8	1027.7 (45%)	1031.2 (45%)	1305.6 (57%)
IDCT	890.72	1541.9 (42%)	1577.2 (44%)	1911.5 (53%)

6.2.2 Area Results for Multiple Throughput Constraint

The relationship between throughput constraint and the pipeline synthesis design space can be seen by visualizing the exploration results for a given dependence graph under a range of throughput constraints. The more complete view also reveals some characteristics of the design space that are not able to be observed under a single throughput constraint. This enlarged design space was computed for the Biquad and FIR filters using the *IMS Exploration* algorithm with a wide range of throughput values, because it has a much shorter runtime than the *ASAP Exploration* algorithm.

These results are visualized in the three dimensional plots of Figure 6.3 and Figure 6.4.

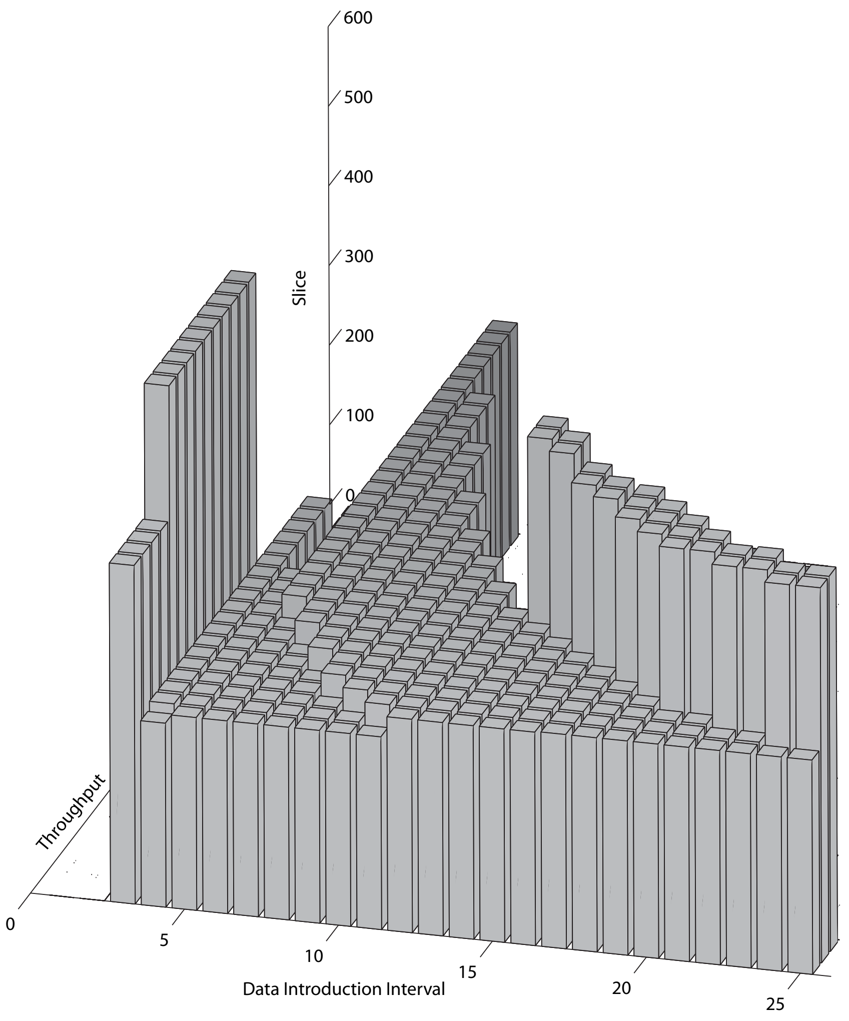


Figure 6.3: 3D Design Space for the Biquad Filter

These plots demonstrate the wide range of implementation possibilities for a single dependence graph. Each design point is represented as a bar in the three dimensional bar graph. The height of the bar (z axis) is the estimated area cost of the minimum cost implementation identified during architectural exploration. The y axis represents the throughput constraints and the x axis represents the (f, δ) pairs

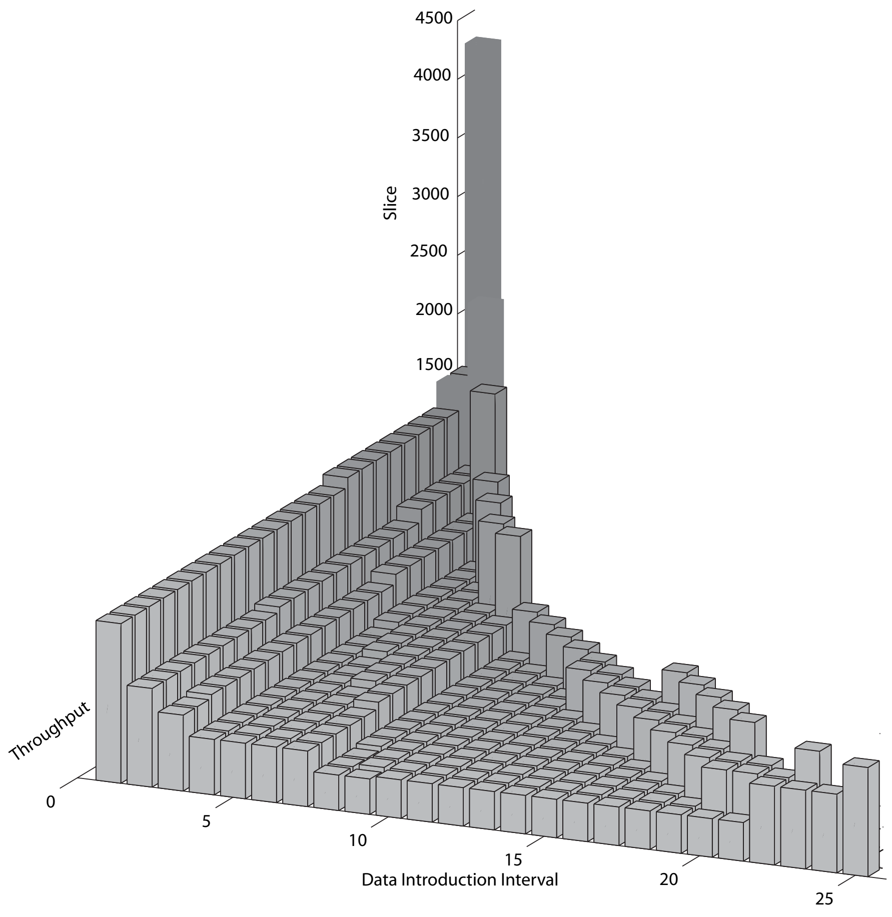


Figure 6.4: 3D Design Space for the FIR Filter

(increasing δ). The single throughput design space plots can be obtained from a cross section perpendicular to the throughput axis of these three dimensional plots.

Several important observations can be seen from these figures. The first observation is that the size of the design space shrinks as the throughput constraint increases. In both figures the maximum data introduction interval δ_{max} decreases with increasing throughput as shown in Equation 3.3. Also, the δ_{min} increases for dependence graphs with feedback as shown in Figure 6.3. Higher frequency constraints require the use of more deeply pipelined, higher latency circuit modules. Use of higher latency modules increases the cycle delay of feedback loops, thus increasing δ_{min} as shown in Equation 3.10.

Another observation from these full design space figures is the presence of a *valley* (i.e. stage II of Figure 6.1). This somewhat continuous valley represents the stage II region of the design space for multiple throughput constraints (see Figure 6.1). This valley contains the minimum cost design points for a range of throughput constraints. As expected, increasing the throughput constraint increases the minimum area cost of the architecture. Table 6.2 demonstrates this by tabulating the minimum area cost of the FIR filter for several throughput constraints. Note that the minimum area increases as the throughput constraint increases.

Table 6.2: Minimum area FIR design points.

Throughput	Minimum Area
12 MSamples/sec	301 Slices
15 MSamples/sec	327 Slices
33 MSamples/sec	524 Slices
65 MSamples/sec	754 Slices
86 MSamples/sec	984 Slices
129 MSamples/sec	1647 Slices
258 MSamples/sec	4383 Slices

6.3 HOIMS Area Results

Although the *ASAP Exploration* algorithm and *IMS Exploration* algorithm demonstrate the advantage of a combined approach, they are limited in design space exploration efficiency, resource sharing design space exploration, and module selection exploration for dependence graphs with non-uniform bit-width operations. The purpose of this section is to illustrate and analyze the advantages of the HOIMS algorithm compared to the previous two algorithms. The first subsection shows the improvements by exploring the resource sharing design space with weighted compatibility graph. The second subsection shows the improvements by using bit-width morphing technique for non-uniform bit-width dependece graphs.

6.3.1 Uniform Bit-Width Area Results

The HOIMS results for the Biquad example are illustrated in Figure 6.5. To compare with the previous results, it assumes uniform bit-width for all operations, which is the same as the *ASAP Exploration* and *IMS Exploration* algorithms. This figure only shows the area result for the combined module selection and resource sharing method along the whole pipeline design space (the grey bar). The corresponding result in Figure E.2 is repeated in this figure (the dark bar) for comparison.

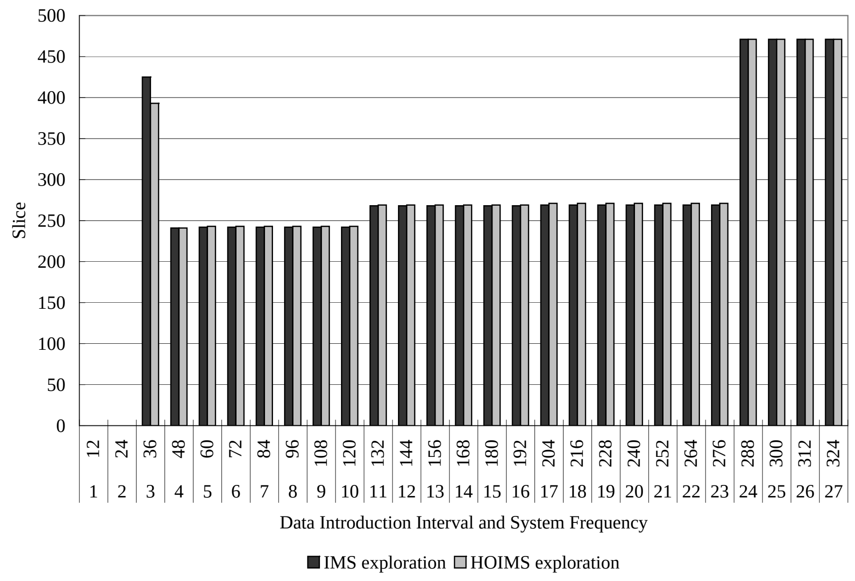


Figure 6.5: Area comparison between HOIMS and *IMS Exploration* for the Biquad example with 12M Samples/Sec throughput constraint

As Figure 6.5 shows, the HOIMS exploration approach yields almost an identical result as the *IMS Exploration* does for all the feasible (f, δ) design points. The only notable difference is at (f, δ) = (36MHz, 3). With the *IMS Exploration* algorithm, two *Array Multiplier 1* are allocated, with two multiplier operations bound to each of them. This requires four 2-input multiplexers plus the 3 to 2 encoder logic. The HOIMS algorithm still allocates two *Array Multiplier 1*. However, the binding of the operations is quite different. One module instance is bound to three multiplier operations and the other is bound to one multiplier operation. This requires only two 3-input multiplexers without the encoder logic. The different binding is caused by the different resource sharing algorithm used in the two approaches. In the *IMS Exploration* approach, the ASAP scheduling time of each operation is always favored and conflicting operations are always unscheduled (i.e. different binding possibilities are not weighted). In HOIMS exploration approach, the whole scheduling window for each operation is explored and different bindings are compared and selected (i.e. weighted clique partitioning). The result of this exploration is a more area efficient implementation as shown in this case.

The 8-tap FIR filter was also tested with the HOIMS exploration approach. To compare the result with *ASAP Exploration* and *IMS Exploration* result, the same throughput constraint (12 MSamples/Sec), the same circuit module library, and the same assumption (uniform bit-width) were used. In Figure 6.6, the *ASAP Exploration* result, the *IMS Exploration* result and the HOIMS exploration result are displayed with white, dark and grey bars respectively. As the figure shows, the HOIMS algorithm generates a more area efficient or the same implementation for all of the pipeline design points than the *IMS Exploration* approach. Its result is very close to the *ASAP Exploration* approach. At (f, δ)=(24MHz, 2) and (36MHz, 3), it generates better result than the ASAP exploration algorithm. The reason for this is the same as the Biquad example, where the non-backtracking ASAP scheduling algorithm results in inefficient utilization of allocated circuit modules, thus using more multiplexers.

At the (f, δ) = (288MHz, 24) pipelining design point, the HOIMS algorithm generates a bigger area solution than the *ASAP Exploration* algorithm. The *ASAP Exploration* algorithm allocated 3 *CoreGen Sequential* for the 8 multiply operations in the dependence graph. The HOIMS algorithm initially selected the *Shift & Add Multiplier* for each multiply operation because it is the smallest circuit module that can run at 288MHz. The module selection refinement algorithm then changed the module selection to the circuit module which has the smallest amortized cost (see Equation 5.6. So the new module selection became *Array Multiplier 5*. No further module selection refinement is performed. The amortized cost equation only considers the maximum number of operations that can be shared on a circuit module. In this

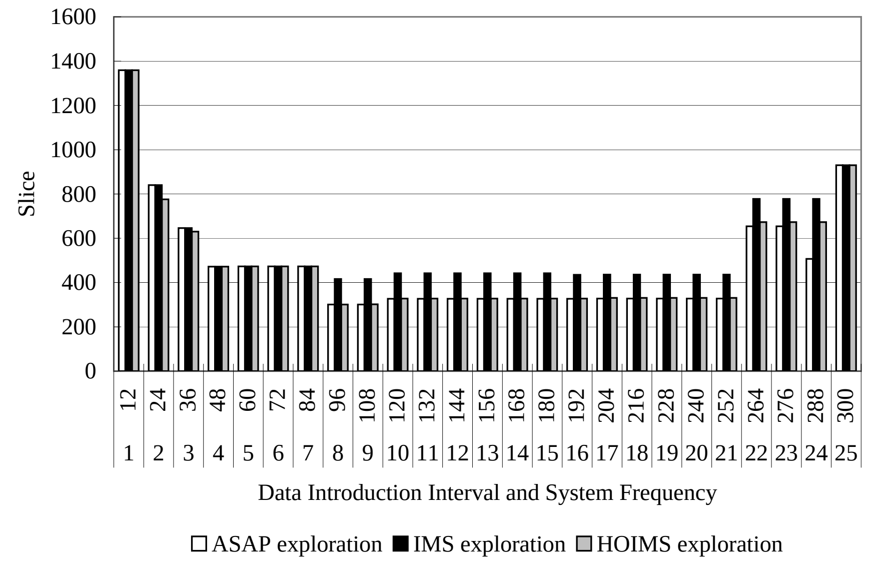


Figure 6.6: Area comparison between *ASAP Exploration*, *IMS Exploration* and HOIMS algorithm for the FIR filter with 12 MSamples/Sec throughput requirement

case, the amortized cost for the *Array Multiplier 5* is about 540 / 24 = 22.5. However, the actual number of operations that can be shared is only 8, so the actual amortized cost is about 540 / 8 = 67.5. The sharing capability of the *CoreGen Sequential* is 3, so its amortized cost is 115 / 3 = 38.3 The inaccurate calculation of the amortized cost caused the selection of the *Array Multiplier 5* over the *CoreGen Sequential*, thus generating an inferior result compared to the exhaustive *ASAP Exploration* algorithm. If the actual amortized cost is calculated, the *CoreGen Sequential* will be selected, and resulting the same hardware cost as the *ASAP Exploration* algorithm.

The uniform bit-width results of the HOIMS algorithm illustrate three advantages over the *ASAP Exploration* and *IMS Exploration* algorithms. First, the exploration of a larger scheduling design space can find better resource sharings than the non-backtracking scheduling algorithm in the *ASAP Exploration* approach. Second, the resource sharing exploration based on the weighted compatibility graph clique partitioning can generate more area efficient circuit architectures than the unweighted approach in the *IMS Exploration* algorithm. Third, the more targeted module selection exploration approach yields better results than the untargeted exploration approach in the *IMS Exploration* algorithm.

6.3.2 Non-uniform Bit-Width Results

All of the previous results are based on the assumption that all of the shared operations have a uniform bit-width. However, operations with different bit-width are very common for computationally intensive algorithms. Uneven bit-width operations make the architectural exploration process more complicated in several ways. First, module selection needs to determine not only the circuit module for each operation, but also the optimal bit-width of the module. Second, different bit-widths may result in a different candidate module sets at each pipelining design point due to different module frequencies, which makes the module selection between different operations more complicated. Third, resource sharing between different bit-width operations requires the "morphing" of smaller bit-width operations to the biggest bit-width size. The trade off between sharing to save circuit module area and the wasting of wider multiplexers requires careful consideration. This section will use some bit-width differentiated computing models to illustrate how the HOIMS algorithm addresses this problem and presents its experimental results for these models.

Figure 6.7 shows the Biquad example with non-uniform bit-widths for the operations (number in the parentheses following each operation name). The throughput constraint remains the same (12 MSamples/Sec). Although the circuit module library is the same as in previous results, and the area and timing characteristics for each circuit module are now assumed proportional to the actual bit-width. For example, the *Array Multiplier 1* in Table 5.1 consumes 162 slices and runs at 41MHz for 16-bit, a 10-bit *Array Multiplier 1* is assumed to consume 162 * (10 / 16) = 101 slices and can run at 41 * (16 / 10) = 65MHz.

Figure 6.8 shows the HOIMS exploration result for the non-uniform bit-width Biquad filter (solid line). The HOIMS result for the uniform bit-width model is repeated in this figure as a comparison (dotted line). As shown in Figure 6.8, the HOIMS exploration for the non-uniform bit-width model generates 11% less hardware area than the uniform bit-width model on average. The module selection refinement and bit-width morphing algorithms are the key in generating such improvement.

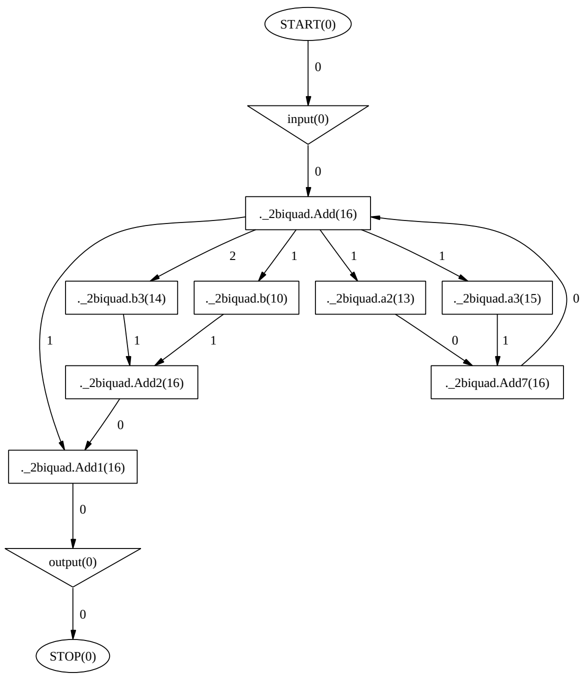


Figure 6.7: Non-uniform bit-width dependence graph of the Biquad filter

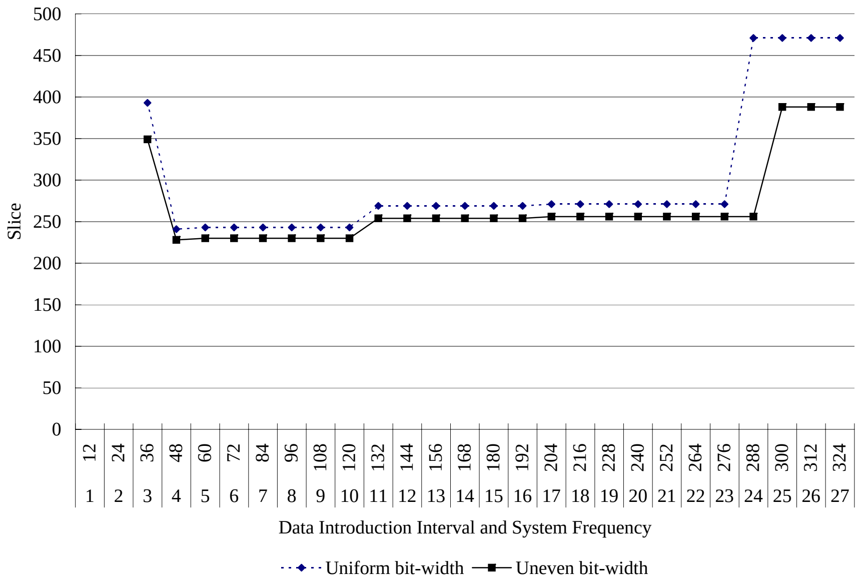


Figure 6.8: Area comparison between uniform bit-width and non-uniform bit-width Biquad filter with 12 MSamples/Sec throughput constraint

The area improvement at the pipelining design point of (36MHz, 3) in Figure 6.8 is the result of the bit-width morphing algorithm. The initial module selection for all multiplier operations is the same with *Array Multiplier 1*. The bit-width of the module is set to be the same as the operation bit-width, which is the most efficient selection without resource sharing. The first iteration of the integrated pipeline scheduling and resource sharing algorithm binds *a3*(15 bit) to a 15-bit module, *a2*(13 bit) to a 13-bit module, and both *b3*(14 bit) and *b*(10 bit) to a 14-bit module, as shown in part (a) of Figure 6.9. This binding is not optimal since none of the modules are fully utilized. Although module selection refinement is not necessary at this point (both the lowest absolute cost module and the lowest amortized cost module are the same), bit-width morphing results in a more efficient binding as shown in part (b) of Figure 6.9. All of the *b3, b* and *a3* operations are bound to a 15-bit module so it is fully utilized, and the *a2* operation is bound to a 13-bit module. The more efficient bit-width allocation results in a total area cost of 349 slices compared to the original 474 slices.

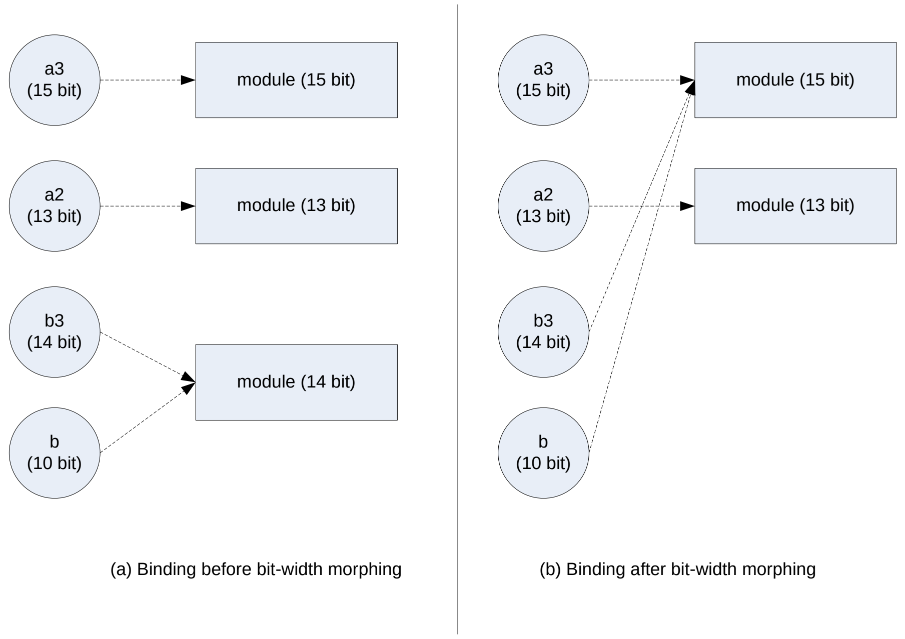


Figure 6.9: Effects of bit-width morphing at (36MHz, 3) for the non-uniform bit-width Biquad filter with HOIMS

In contrast, the area improvement at the pipelining design point of (48MHz, 4) in Figure 6.8 is the result of the module selection refinement algorithm. At this pipelining design point, the candidate module set for the multiplier operations is different. The *Array Multiplier 1* is not in the candidate module set for operation *b3* and *a3*, because it cannot run at this frequency with these two operation's bit-width configurations. So *Coregen Parallel 1* is assigned to these two operations. Both *Array Multiplier 1* and *Coregen Parallel 1* are in the candidate module set for operation *a2* and *b*. Initially *Array Multiplier 1* is assigned to these two operations because it is smaller than *Coregen Parallel 1.* Although this is the most efficient module selection for each operation alone as shown in part (a) of Figure 6.10, it is not efficient from a global resource usage point of view, because each module is utilized only 50% of the time. The module selection refinement algorithm changes the module choice for *a2* and *b* to *Coregen Parallel 1* as shown in part (b) of Figure 6.10. Although it is more expensive for each single operation, resource sharing between operations reduces the overall area from 358 slices to 228 slices.

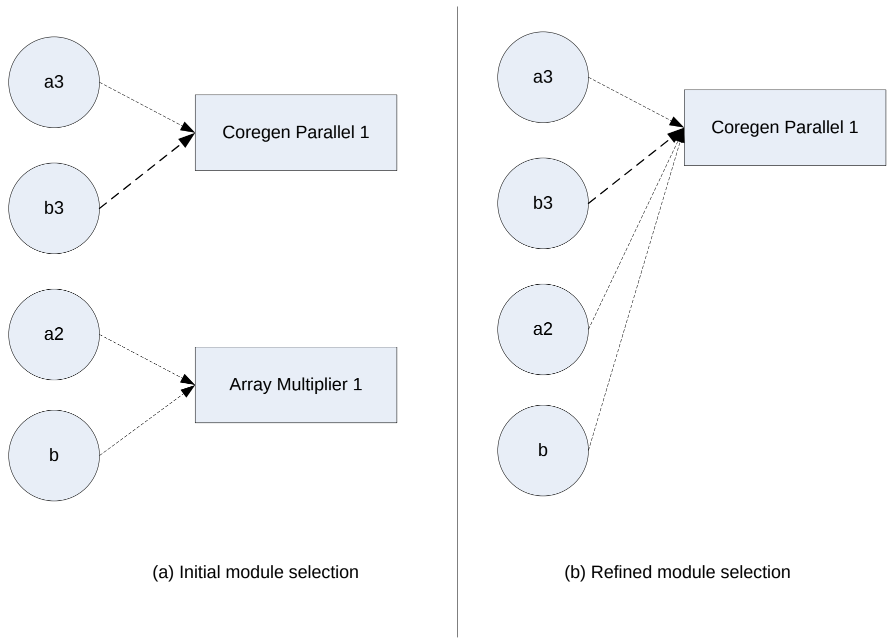


Figure 6.10: Effects of module selection refinement at (48MHz, 4) for the non-uniform bit-width Biquad filter with HOIMS

The 8-tap FIR filter with non-uniform bit-width was tested with the HOIMS exploration algorithm. The same throughput constraint (12 MSamples/Sec) and same circuit module library were used. The area result is shown in Figure 6.11 with the solid line. The uniform bit-width result is repeated in this figure for comparison. As shown in Figure 6.11, the non-uniform bit-width filter generates more area efficient architectures than the uniform bit-width model, especially at small data introduction interval values.

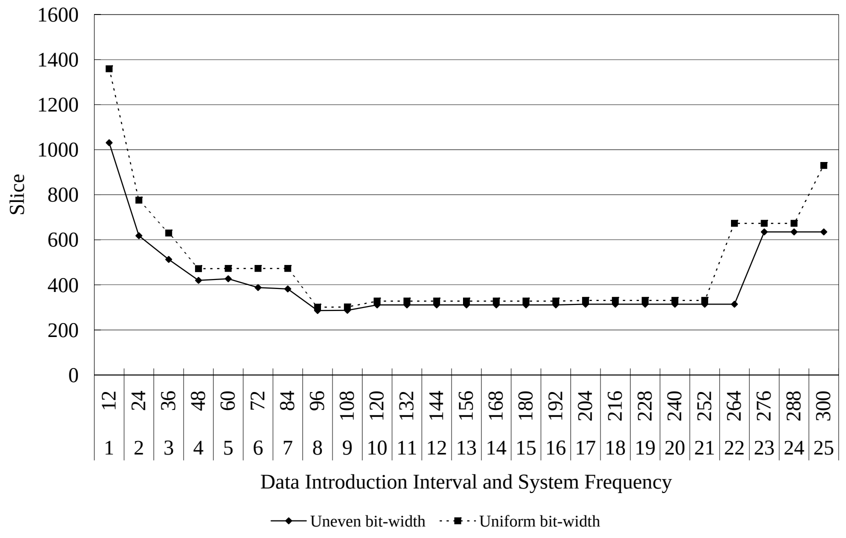


Figure 6.11: Area comparison between uniform bit-width and non-uniform bit-width FIR filter with 12 MSamples/Sec throughput constraint

6.4 Runtime Results

In addition to the area cost of the synthesized circuit, runtime is another important quality measure of pipeline synthesis algorithms. Runtime is the amount of time the synthesis algorithm takes to generate the final schedule and binding from a dependence graph. It is often represented by the *computational complexity* of the algorithm [60]. The complexity of the algorithm is determined by the size of the design space that the synthesis algorithm explores and the efficiency of such exploration.

Table 6.3: Bit-width morphing and binding at (24MHz, 2) for the FIR filter.

Operation	Original Bit-width	Bound Module
Mult	13	$ArrayMultiplier1\_13\_1$
Mult4	11	$ArrayMultiplier1\_13\_1$
Mult7	15	$ArrayMultiplier1\_15\_1$
Mult8	14	$ArrayMultiplier1\_15\_1$
Mult5	10	$ArrayMultiplier1\_10\_1$
Mult6	9	$ArrayMultiplier1\_10\_1$
Mult2	12	$ArrayMultiplier1\_12\_1$
Mult3	12	$ArrayMultiplier1\_12\_1$

The *ASAP Exploration* and *IMS Exploration* algorithms explore the pipeline synthesis design space in different ways, thus they have different algorithm complexity and runtime. The *ASAP Exploration* algorithm explores all of the module selection design space, all of the resource sharing design space of scheduled operations, but a limited portion of scheduling design space due to its non-backtracking scheduling. Its computational complexity is $O(M^N N^2)$, where M is the average number of compatible circuit modules for each operation, and N is the total number of operations in the dependence graph. The *IMS Exploration* algorithm explores part of the module selection design space with a heuristic module selection algorithm, more of the scheduling design space than the *ASAP Exploration* algorithm with an iterative backtracking scheduling approach, and part of the resource sharing design space which is performed concurrently with scheduling. The experimental computational complexity of the *ASAP Exploration* algorithm is $O(N^3 \ln(N))$.

The computational complexity of the HOIMS algorithm is difficult to calculate. There are three main loops of the HOIMS algorithm as shown in Algorithm 3.2. The outermost loop (line 5) explores the pipelining design space. The size of this design space is determined by the throughput constraint and the maximum system clock frequency. It can be treated as a constant. The second loop (line 16) explores the pipeline scheduling and resource sharing design space. In the worst case where each operation is sharable with any other operations, the computational complexity of this loop is $O(N^2 R)$, where N is the number of operations in the dependence graph and R is the average scheduling number for each operation. The third loop (11) explores the module selection design space based on the module selection refinement heuristic. The computational complexity of this loop is difficult to calculate, because the number of iterations is based on the candidate module set size of each operation as well as the scheduling and sharing result with each module selection refinement. The average number of iterations from the experimental results is close to the average module set size for each operation (M). So the computational complexity of HOIMS is $O(MN^2)$.

Figure 6.12 illustrates the logarithmic computational complexity of the three algorithms, assuming $R = 2$ and $M = 4$. As the figure shows, the computational complexity of the *ASAP Exploration* algorithm is exponential and is not practical for large designs. The growth of the *IMS Exploration* algorithm is much slower than the *ASAP Exploration* algorithm, and the HOIMS algorithm is even slower. A runtime report for the three algorithms is shown in Table 6.4. This is roughly in consistent with the computational complexity of the three algorithms. The HOIMS algorithm actually performs better than Figure 6.12 shows. The reason for this is that the actual number of module selection refinement iterations, the average operation rescheduling count, and the resource sharing exploration count is less than the above analysis.

Table 6.4: Runtime report for *ASAP Exploration* algorithm, *IMS Exploration* algorithm and HOIMS algorithm.

Algorithm	Circuit	Time (Seconds)
ASAP	Color Space Conversion	0.8
	FIR	955.9
	FFT	5053.7
	Linear Interpolator	32036.5
	IDCT	543142.8
IMS	Biquad	36.4
	FIR	273.6
HOIMS	Biquad	2.6
	FIR	4.5

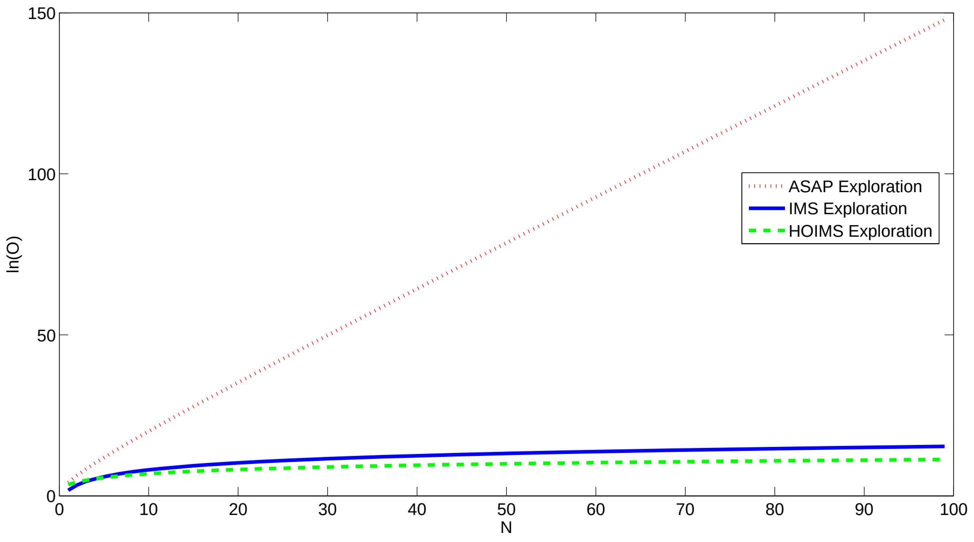


Figure 6.12: Logarithmic computational complexity compare.

Figure 6.13 illustrates the number of module selection iterations performed by the HOIMS algorithm for the Biquad example. The solid line shows the iteration number for the uniform bit-width Biquad filter, and the dotted line shows the iteration number for the non-uniform bit-width Biquad filter. As the uniform bit-width result shows, the module selection iterations explored at each (f, δ) by HOIMS is obviously less than the exponential count of the *ASAP Exploration* algorithm, it is also much less than the *IMS Exploration* algorithm. In the *IMS Exploration* algorithm, $Nlog_2N$ iterations of module selection are performed. In the case of the Biquad example, $N = 12$ so it tries 48 different module selections at each pipelining design point. The HOIMS algorithm only takes 4.7 iterations of module selection on average as Figure 6.13 shows.

The non-uniform bit-width Biquad filter requires slightly more module selection exploration iterations than the uniform bit-width version (6.56 versus 4.7 for the uniform bit-width model). The non-uniform bit-width among the operations requires a bigger module selection design space which is explored by the bit-width morphing technique as described in Section 5.5.4, so more iterations of module selection refine-

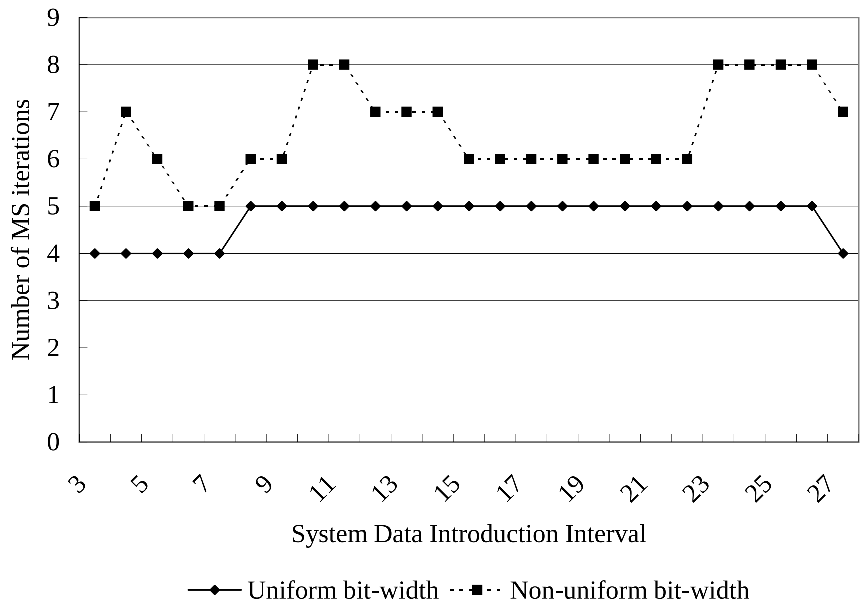


Figure 6.13: Number of module selection iterations by HOIMS at each pipelining design point for the Biquad example

ment and bit-width morphing are necessary to find the optimal circuit module and bit-width configuration for each operation.

The module selection iteration count for the 8-tap FIR filter also demonstrates the efficiency of the HOIMS algorithm. The average module selection iteration count for the FIR filter with uniform bit-width is 4.6 times as shown by the solid line in Figure 6.14. This is much less than the *IMS Exploration* algorithm which requires $15log_2 15 = 60$ iterations of module selection exploration. The module selection iteration count for the FIR filter with non-uniform bit-width is illustrated by the dotted line in Figure 6.14. Similar to the Biquad example, it performs slightly more iterations of module selection refinement (6.1 versus 4.6 on average).

The HOIMS algorithm also explores the pipeline scheduling and binding (possibly resource sharing) design space more efficiently than the other two algorithms. The *ASAP Exploration* algorithm only explores the ASAP scheduling design space, and all of the resource sharing design space among all previously scheduled operations under that schedule. The *IMS Exploration* algorithm explores more of the pipeline scheduling and resource sharing design space with its backtracking schedul-

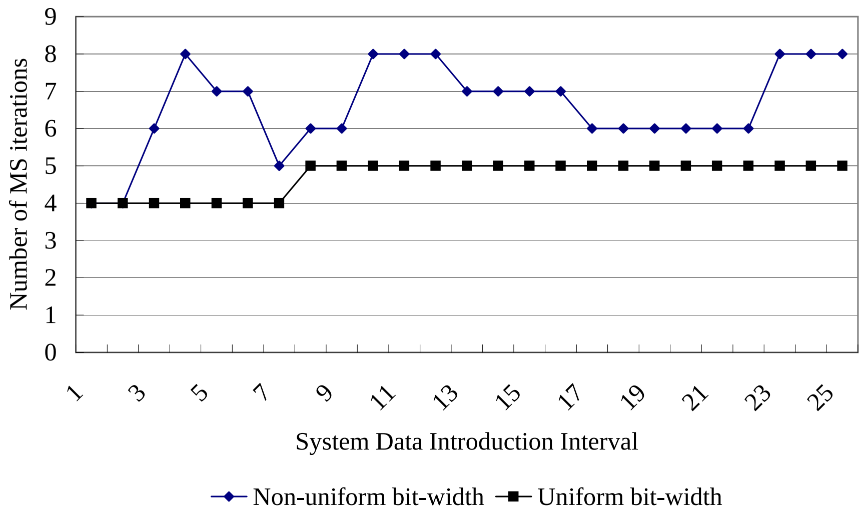


Figure 6.14: Number of module selection iterations by HOIMS at each pipelining design point for the FIR filter

ing approach. However, because it doesn't differentiate between multiple sharing possibilities, it requires quite a few iterations of unscheduling and rescheduling until it finds a schedule (along with binding and sharing). The HOIMS algorithm uses the weighted compatibility graph to explore the resource binding/sharing and pipeline scheduling design space as described in Section 4.5. This approach greatly reduces the number of unscheduling and rescheduling steps compared to the *IMS Exploration* algorithm. The experimental results show that the *IMS Exploration* algorithm requires N (number of operations in the dependence graph) iterations of unscheduling and rescheduling until it finds a valid pipeline schedule and binding, while the HOIMS algorithm requires no unscheduling for the Biquad and FIR examples.

6.5 Summary

This chapter presents and discusses the experimental results of three pipeline synthesis algorithms that concurrently explores pipeline scheduling, module selection and resource sharing. It focuses on the benefits of the combined design space and the efficiency of the synthesis algorithms that explore such a space. The analysis of the pipeline synthesis design space shows that combining module selection and resource sharing while performing pipeline scheduling can significantly increase the entire design space and reduce the overall hardware area cost. The results from the *ASAP Exploration* and *IMS Exploration* algorithms are in accordance with such an analysis. The results from the HOIMS algorithm show that it further improves the quality of the synthesized circuit. The bit-width morphing capability of the HOIMS algorithm generates more efficient hardware when the operations have non-uniform bit-width. The algorithm complexity and runtime results from these three algorithms are finally presented and discussed. Although the combined design space is intractable, the iterative module selection refinement and weighted compatibility graph based resource sharing of the HOIMS algorithm enable it to explore the combined design space very efficiently while generating good results.

Chapter 7

Conclusion and Future Work

This work demonstrates a high-level synthesis methodology that automatically synthesizes area efficient FPGA implementations from untimed computationally intensive algorithms under a throughput constraint. Automatically generating hardware implementations from untimed algorithms can significantly improve the productivity of implementing these algorithms in hardware. This methodology can also greatly enhance the synthesized hardware quality by automatically evaluating a huge number of design alternatives which all meet the throughput constraint. Although we used an FPGA-specific circuit module library and addressed FPGA-unique resource sharing cost issue, this research can be applied to ASICs as well, given a corresponding circuit module library and sharing cost.

This work proposes a novel pipeline synthesis algorithm that combines three closely inter-related synthesis techniques: pipeline scheduling, resource sharing and module selection. Pipeline scheduling not only generates a pipelined schedule from an untimed algorithm description, it also explores the entire pipelining design space to evaluate different circuit architectures that meet the throughput constraint. Resource sharing is performed concurrently during pipeline scheduling. This work proposes a resource sharing algorithm that is based on a unique weighted compatibility graph, which differentiates all the sharing possibilities for each operation. Module selection is integrated with the other two techniques in the proposed pipeline synthesis algorithm. This work proposes a novel iterative module selection refinement algorithm that efficiently explores the exponential module selection design space.

This work demonstrates that combining module selection with resource sharing during pipeline scheduling can generate 43% smaller circuit architecture than when either technique is used independently on average. By trading off between using specific circuit modules (i.e. less resource sharing) and using general circuit modules (i.e. more resource sharing), this combined pipeline synthesis approach explores a much bigger design space than when either technique is applied alone. The result of the combined approach shows that the best hardware architecture is always obtained at some pipelining design point, and by exploring both module selection and resource sharing.

This work also demonstrates that efficiently exploring the combined design space of pipeline scheduling, resource sharing and module selection while generating superior results is feasible. The design space of each technique is exponential, and the combined design space of these three techniques is even bigger. Three combined algorithms are discussed in this work, and each explores the combined design space in a different way. The HOIMS exploration algorithm is the best algorithm in this work, which efficiently explores the combined design space and generates superior results.

This work provides three major contributions to FPGA specific high-level synthesis for computationally intensive algorithms. First, this work proposed a throughput driven pipeline synthesis methodology that explores the entire pipeline design space. Most pipeline synthesis algorithms do not explore the whole pipelining design space to discover the most efficient hardware architecture under the throughput constraint. For this work, given a fixed throughput constraint, it explores *all* feasible frequency and data introduction interval design points that meet this throughput constraint. This expanded pipelining design space exploration results in hardware architectures that are far superior to those resulting from previous pipeline synthesis work.

Second, this work proposes a unique module selection algorithm which not only considers different module architectures, but also different pipelining options for each architecture. This not only addresses the unique architecture of most FPGA circuit modules, it also performs retiming at the high-level synthesis level. This is very important when the whole pipelining design space is explored, because different pipelined circuit modules can be used under different system frequency and data introduction interval requirements.

Third, this work proposes a novel approach for integrating three deeply inter-related synthesis techniques: pipeline scheduling, module selection and resource sharing. To the author's best knowledge, this is the first attempt to do this. Although the three techniques are deeply inter-related and extremely difficult to integrate, a study of each technique and their inter-dependency has been providedw. Efficient integrated algorithms were then proposed and it was able to identify more efficient hardware implementations than when only one or two of the three techniques are applied.

When implementing computationally intensive streaming algorithms in FPGAs, this work can be applied to significantly improve the design productivity and quality. First, with this work, algorithm designers need to only specify the *untimed function* of the algorithm, and the hardware architecture of the FPGA implementation will be automatically created. Automatic transformation from untimed algorithm to concrete hardware implementation is key to solve the design productivity gap challenge as discussed in Chapter 1. Second, this work can be used by algorithm designers to explore a large number of micro-architectures. The different architectures generated by this work vary in module selection, scheduling and resource sharing, so that the designers can choose the ones that meet specific requirements. Third, this work can be used to evaluate various FPGA circuit modules. As dicussed in Chapter 5, candidate circuit modules have a big impact on the synthesis result. Since the HOIMS algorithm can efficiently explore the module selection design space under the context of pipeline scheduling and resource sharing, it can be applied to test the impact of candidate circuit modules on the synthesis result.

7.1 Future Work

There are several areas where this work can be further improved: programming languages support, hybrid timing model for the circuit modules, coarser grain circuit modules and more accurate cost estimation. Improving these areas will have a positive impact on the applicability of the synthesis algorithm, so that it can support more complexed design input and targeted for more hardware architectures. The proposed future work can also improve the quality of the synthesized circuit.

When more complicated computationally intensive algorithms need to be represented, the SDF specification format becomes inconvenient. High level programming languages offer more flexibilities and control. In this case, control constructs must be supported in the synthesis process. The scheduling algorithm should be extended to determine the start time of operations which belong to different control sequences, so should the resource sharing algorithm. With programming languages, not only traditional compiler optimization techniques [61] such as loop unrolling, loop merging, arithmetic optimization and data-flow optimization, but also custom hardware specific optimizations [62] such as bit-width optimization, multiplexer reduction etc, are essential in generating efficient hardware implementaitons.

The multi-cycle, pipelined circuit module timing model used in this work can be extended to a hybrid timing model [63], where combinational delays are also modeled in addition to possible pipeline stages. This hybrid timing model can not only represent more circuit modules, but also allow *chaining* in the hardware implementation, which can reduce the circuit latency. The hybrid timing model also expands the pipelining design space for a finer grain of frequency exploration.

The granularity of the circuit modules can be improved in future work. This work utilizes only simple circuit modules that implement basic arithmetic functions such as add and multiply. Coarser grain circuit modules which implement more complex functions such as square root, complex multiply and FFT butterfly, can be included in the circuit module library. Because coarser grain circuit modules can *cover* a subgraph of the dependence graph, they can be used to support hierarchical design as well as reusing previously synthesized subgraphs. However, module selection becomes more challenging because it has to identify the optimal subgraph which can be implemented by the coarse-grain circuit module. The resource sharing algorithm also becomes more complicated because sharing does not only occur between operations, but between subgraphs as well.

Future work should improve the area and timing estimation algorithm for the circuit modules and resource sharing overhead. Accurate area and timing estimation is very important yet challenging for high level synthesis. The regular structure of FPGA devices makes the estimation even more difficult because operations can be mapped, merged and placed differently by downstream tools under different design context. Some early work [64, 65, 66] has proposed ways to improve the estimation accuracy by considering more effects from RTL synthesis, placement and layout tools. However, more FPGA specific research needs to be conducted in this area to improve the quality of high level synthesis for FPGA designs.

7.2 Summary

Hardware synthesis has become indispensible in the past and current design of electronic circuits and systems. However, with the ever increasing design complexity combined with decreasing design time requirements, hardware synthesis tools have to be improved in several ways. First, they must support synthesis from high-level specification to significantly improve the design team's productivity. Second, they must perform extensive design space exploration to meet critical design goals such as area, timing and power consumption. Third, they must support various technologies such as ASICs and FPGAs. Initial research in these areas has shown quite promising results. This work has proposed a good pipeline synthesis algorithm that transforms untimed high-level specification into area optimized FPGA implementation. It also provides a good framework for the above proposed future work. With the improvements in front-end, circuit module library and estimation algorithms, this work will provide invaluable techniques and algorithms for future high-level synthesis research as well as tool development.

Bibliography

[1] R. Hartenstein, “A decade of reconfigurable computing: a visionary retrospective,” in *DATE ’01: Proceedings of the conference on Design, automation and test in Europe.* Piscataway, NJ, USA: IEEE Press, 2001, pp. 642–649. xxiii, 2

[2] A. Allan, D. Edenfeld, J. William H. Joyner, A. B. Kahng, M. Rodgers, and Y. Zorian, “2001 technology roadmap for semiconductors,” *Computer*, vol. 35, no. 1, pp. 42–53, 2002. xxiii, 4

[3] R. Porter, J. Frigo, M. Gokhale, C. Wolinski, F. Charot, and C. Wagner, “A programmable, maximal throughput architecture for neighborhood image processing,” in *FCCM ’06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’06).* Washington, DC, USA: IEEE Computer Society, 2006, pp. 279–280. 1

[4] Altera, “Stratix-II device handbook,” May 2007. 1

[5] Xilinx, “Virtex-II complete data sheet,” March 2005. 1

[6] K. U. Järvinen, M. T. Tommiska, and J. O. Skyttä, “A fully pipelined memoryless 17.8 gbps AES-128 encryptor,” in *FPGA ’03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays.* New York, NY, USA: ACM Press, 2003, pp. 207–215. 1

[7] Xilinx, “Virtex-E complete data sheet,” January 2006. 1

[8] J. Singaraju, L. Bu, and J. A. Chandy, “A signature match processor architecture for network intrusion detection,” in *FCCM ’05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’05).* Washington, DC, USA: IEEE Computer Society, 2005, pp. 235–242. 1

[9] Y.-L. Lin, “Recent developments in high-level synthesis,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 2, no. 1, pp. 2–21, 1997. 4

[10] G. D. Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, 1994. 5, 12, 14, 42, 47, 64

[11] G. D. Micheli, D. Ku, F. Mailhot, and T. Truong, “The olympus synthesis system,” *IEEE Des. Test*, vol. 7, no. 5, pp. 37–53, 1990. 6, 32

[12] R. Brayton, R. Camposano, G. D. Micheli, R. Otten, and J. van Eijndhoven, "The Yorktown silicon compiler system," IBM Research Report RC 12500, Tech. Rep., February 1986. 6

[13] S. Note, J. V. Meerbergen, F. Catthoor, and H. D. Man, "Automated synthesis of a high-speed cordic algorithm with the CATHEDRAL-III compilation system," in *Proceedings of ISCAS'88*, June 1988, pp. 581–584. 6

[14] R. Woudsma, F. Beenker, J. van Meerbergen, and C. Niessen, "PIRAMID: an architecture-driven silicon compiler for complex DSP applications," in *Proc. of IEEE International Symposium on Circuits and Systems*, 1990, pp. 2696–2700. 6

[15] P. G. Paulin, J. P. Knight, and E. F. Girczyc, "HAL: a multi-paradigm approach to automatic data path synthesis," in *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation.* Piscataway, NJ, USA: IEEE Press, 1986, pp. 263–270. 6, 12, 16

[16] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays.* New York, NY, USA: ACM Press, 2006, pp. 21–30. 11

[17] M. Xu and F. J. Kurdahi, "Layout-driven high level synthesis for FPGA based architectures." in *1998 Design, Automation and Test in Europe (DATE '98).* IEEE Computer Society, Febuary 1998, pp. 446–450. 12

[18] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for FPGAs," in *DATE '02: Proceedings of the conference on Design, automation and test in Europe.* Washington, DC, USA: IEEE Computer Society, 2002, p. 862. 12

[19] D. Chen, J. Cong, and Y. Fan, "Low-power high-level synthesis for FPGA architectures," in *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design.* New York, NY, USA: ACM Press, 2003, pp. 134–139. 12

[20] D. Ku and G. DeMicheli, "HardwareC – a language for hardware design (version 2.0)," Stanford, CA, USA, Tech. Rep., 1990. 12

[21] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J. R. Uribe, "Overview of a compiler for synthesizing MATLAB programs onto FPGAs," *IEEE Trans. VLSI Syst.*, vol. 12, no. 3, pp. 312–324, 2004. 12

[22] R. Camposano and W. Rosentiel, "Synthesizing circuits from behavioral descriptions," *IEEE Trans. on CAD*, vol. 8, no. 2, pp. 171–180, February 1989. 12

[23] V. K. Raj, "DAGAR: An automatic pipelined microarchitecture synthesis system," in *Proc. of ICCD'89*, October 1989, pp. 428–431. 12

[24] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987. 13

[25] E. A. Lee and A. Sangiovanni-Vincentelli, "Comparing models of computation," in *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design.* Washington, DC, USA: IEEE Computer Society, 1996, pp. 234–241. 13

[26] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation, special issue on "Simulation Software Development"*, vol. 4, pp. 155–182, April 1994. 13

[27] M. Barbacci, "Automatic exploration of the design space for register transfer (RT) systems," Ph.D. dissertation, Carnegie-Mellon University, November 1973, department of Computer Science. 15

[28] S. Davidson, D. Landskov, B. Shriver, and P. W. Mallett, "Some experiments in local microcode compaction for horizontal machines." *IEEE Trans. Computers*, vol. 30, no. 7, pp. 460–477, 1981. 15

[29] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "Complexity of scheduling in high level synthesis," *VLSI Design*, vol. 7, no. 4, pp. 337–346, 1998. 15, 18

[30] R. J. Cloutier and D. E. Thomas, "The combination of scheduling, allocation, and mapping in a single algorithm," in *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation.* New York, NY, USA: ACM Press, 1990, pp. 71–76. 16

[31] S. Raje and R. Bergamaschi, "Generalized resource sharing," in *Digest of Technical Papers, IEEE/ACM International Conference on Computer-Aided Design.* IBM Thomas J. Watson Res. Center, Yorktown Heights, NY, USA, November 1997, pp. 326–332. 16, 18, 47, 48, 49

[32] N. Park and A. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 3, pp. 356–370, March 1988. 17, 21, 33

[33] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *MICRO 27: Proceedings of the 27th annual International Symposium on Microarchitecture.* New York, NY, USA: ACM Press, 1994, pp. 63–74. 21, 30, 34, 56, 86, 147

[34] P. M. Kogge, *The Architecture of Pipelined Computers.* McGraw-Hill Book, 1981. 22

[35] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1974. 32

[36] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 6, pp. 661–679, June 1989. 33

[37] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. L. van Meerbergen, and A. van der Werf, "Improved force-directed scheduling in high-throughput digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, pp. 945–960, August 1995. 33

[38] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, "PLS: a scheduler for pipeline synthesis." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 12, no. 9, pp. 1279–1286, 1993. 34

[39] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation.* New York, NY, USA: ACM Press, 1988, pp. 318–328. 34

[40] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," *IEEE Transactions on Computer*, vol. C-21, no. 2, pp. 137–146, Febuary 1972. 35, 130

[41] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974. 35, 130

[42] D. L. Springer and D. E. Thomas, "Exploiting the special structure of conflict and compatibility graphs in high-level synthesis." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 13, no. 7, pp. 843–856, 1994. 47

[43] D. C. Ku and G. D. Micheli, *High level synthesis of ASICs under timing and synchronization constraints.* Norwell, MA, USA: Kluwer Academic Publishers, 1992. 48

[44] S. Mondal and S. O. Memik, "Resource sharing in pipelined CDFG synthesis," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation.* New York, NY, USA: ACM Press, 2005, pp. 795–798. 48

[45] Xilinx, "Xilinx Coregen reference manual," 2005. 61, 124

[46] J. Abke, E. Barke, and J. Stohmann, "A universal module generator for LUT-based FPGAs," *rsp*, vol. 00, p. 230, 1999. 61

[47] J. P. Singh, A. Kumar, and S. Kumar, "A multiplier generator for Xilinx FPGA's," in *9th International Conference on VLSI Design*, January 1996, pp. 322–323. 61

[48] Z. Shen and C. Jong, "Exploring module selection space for architectural synthesis of low power designs," in *Proceedings of the 1997 IEEE International Symposium on Circuits and Systems*, June 1997, pp. 1532–1535. 63, 69

[49] R. Jain, A. Parker, and N. Park, "Module selection for pipelined synthesis," in *Proceedings of the 25th Annual ACM/IEEE Design Automation Conference*, June 1988, pp. 542–547. 63, 69

[50] I. G. Harris and A. Orailoglu, "Intertwined scheduling, module selection and allocation in time-and-area constrained synthesis," in *IEEE International Symposium on Circuits and Systems*, May 1993, pp. 1682 – 1685. 64

[51] I. Ahmad and M. K. Dhodhi, "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis," *Proceedings of the IEE Computers and Digital Techniques*, vol. 142, no. 1, pp. 65–71, January 1995. 64

[52] D. Thomas and G. Leive, "Automating technology relative logic synthesis and module selection," *IEEE Transactions on CAD/ICAS*, vol. CAD-2, no. 2, pp. 94–105, April 1983. 68

[53] R. Jain, "MOSP: Module selection for pipeliend designs with multi-cycle operations," in *Proceedings of the 1990 IEEE International Conference on Computer-Aided Design – ICCAD'90*, November 1990, pp. 212–215. 69

[54] K. Ito, L. Lucke, and K. Parhi, "ILP-based cost-optimal DSP synthesis with module selection and data format conversion," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 6, no. 5, pp. 582–594, December 1998. 69

[55] S. Bakshi and D. D. Gajski, "Component selection for high-performance pipelines," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, vol. 4, no. 2, pp. 181–194, June 1996. 69

[56] S. Bakshi, D. Gajski, and H. Juan, "Component selection in resource shared and pipelined DSP applications," in *EURO-DAC '96/EURO-VHDL '96: Proceedings of the Conference on European Design Automation.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1996, pp. 370–375. 69

[57] L. Hafer and A. C. Parker, "A formal method for the specification, analysis, and design of register-transfer level digital logic," in *DAC '81: Proceedings of the 18th conference on Design automation.* Piscataway, NJ, USA: IEEE Press, 1981, pp. 846–853. 69

[58] W. Sun, M. J. Wirthlin, and S. Neuendorffer, “Combining module selection and resource sharing for efficient FPGA pipeline synthesis,” in *FPGA ’06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays.* New York, NY, USA: ACM, 2006, pp. 179–188. 86

[59] ——, “FPGA pipeline synthesis design exploration using module selection and resource sharing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, Febuary 2007. 86

[60] D. E. Knuth, “Big omicron and big omega and big theta,” *SIGACT News*, vol. 8, no. 2, pp. 18–24, 1976. 100

[61] S. S. Muchnick, *Advanced compiler design and implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. 110

[62] G. Snider, B. Shackleford, and R. J. Carter, “Attacking the semantic gap between application programming languages and configurable hardware,” in *FPGA ’01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays.* New York, NY, USA: ACM, 2001, pp. 115–124. 110

[63] G. Snider, “Performance-constrained pipelining of software loops onto reconfigurable hardware,” in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, 2002, pp. 177 – 186. 110

[64] P. K. Jha and N. D. Dutt, “Rapid estimation for parameterized components in high-level synthesis,” *IEEE Trans. on CAD*, vol. 1, no. 3, pp. 296–303, September 1993. 111

[65] S. Y. Ohm, F. J. Kurdahi, N. Dutt, and M. Xu, “A comprehensive estimation technique for high-level synthesis,” in *ISSS ’95: Proceedings of the 8th international symposium on System synthesis.* New York, NY, USA: ACM, 1995, pp. 122–127. 111

[66] B. So, P. C. Diniz, and M. W. Hall, “Using estimates from behavioral synthesis tools in compiler-directed design space exploration,” in *DAC ’03: Proceedings of the 40th conference on Design automation.* New York, NY, USA: ACM, 2003, pp. 514–519. 111

[67] P. Bellows and B. Hutchings, “JHDL - an HDL for reconfigurable systems,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM ’98)*, K. L. Pocek and J. M. Arnold, Eds., IEEE Computer Society. IEEE Computer Society Press, April 1998, pp. 175–184. 124

[68] C. Leiserson and J. Saxe, “Retiming synchronous systems,” *Algorithmica*, vol. 6, no. 1, 1991. 125

[69] M. C. Papaefthymiou, “Understanding retiming through maximum average-weight cycles,” in *SPAA ’91: Proceedings of the third annual ACM symposium*

on Parallel algorithms and architectures. New York, NY, USA: ACM Press, 1991, pp. 338–348. 127

[70] C. Leiserson and J. Saxe, "Optimizing synchronous systems," *Journal of VLSI and Computer Systems*, vol. 1, no. 1, 1983. 127

Appendices

Appendix A

Multi-cycle Pipelined Circuit Modules

A.1 Circuit Module Characterization and Description

A sample list of FPGA-specific circuit modules are characterized and used for this thesis. The characterization process for these modules is illustrated in Figure A.1. The following circuit modules are characterized:

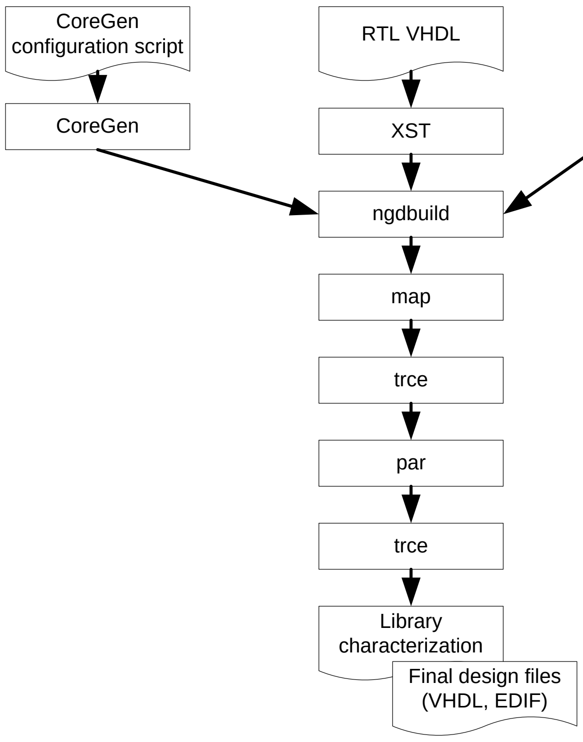


Figure A.1: Circuit module characterization flow

Array Multiplier This parameterizable multiplier was created in RTL VHDL. It employs the ripple carry adder to make use of the fast FPGA carry chain architecture feature. The parameterizable pipeline stages divide partial product adders evenly to increase the throughput at a cost of latency and area. Although array multiplier 2 and 3 are dominated by CoreGen parallel multiplier 1 and 2, this multiplier provides more pipelining capabilities than the CoreGen versions.

Booth Multiplier This multiplier was generated from a JHDL [67] module generator optimized for the Virtex architecture. The use of booth recoding reduces the number of partial products and therefore improves the combinational latency of the operation. This decrease in latency is obtained at a cost of increased area. This multiplier also provides different pipelining capabilities.

CoreGen Parallel This multiplier was created by CoreGen[45], a module generator tool supported by Xilinx. Although two variations of pipelining are supported by this multiplier, the internal multiplier architecture is hidden from the user.

CoreGen Sequential This multiplier was also created by CoreGen using the "2 bits/cycle" option for sequential execution. This multiplier generates the partial product by performing two bits of the computation each cycle. The reported area of this module includes the control and other miscellaneous area overhead.

Shift & Add Multiplier This multiplier was also created by CoreGen using the "1 bit/cycle" option for sequential multiplier.

Bit-Serial Multiplier This multiplier was created by an RTL VHDL code. The primary area cost in this design is the parallel to serial and serial to parallel converters. The small combinational delay allows this module to operate at a much higher clock rate than the parallel multipliers.

Dedicated Multiplier These multipliers were created by CoreGen and exploit the dedicated hardware multiplier in the Virtex4 architecture. The dedicated multiplier is a custom resource that does not use logic slices. In order to use this resource during module selection, a strategy for comparing the area cost of this module against the slice-based modules must be introduced.

Ripple Carry Adder This is a conventional ripple-carry adder that exploits the fast carry logic of the FPGA architecture. Ripple carry adder 2 is a pipelined version of ripple carry adder 1.

Bit-Serial Adder This module performs addition using bit-serial arithmetic at one bit per clock cycle. The area of this circuit includes the data format converters and other control overhead.

A.2 Multi-cycle, Pipelined Circuit Module and C-Slow Retiming

Retiming [68] is a transformation on synchronous circuits [68]. It addresses the problem of minimizing the cycle-time or the area of synchronous circuits by changing the position of the registers. Because the system clock is bounded by the critical path delay in the combinational component of a synchronous circuit, i.e., by the longest combinational path between a pair of registers. Hence retiming aims at moving and placing the registers in appropriate positions, so that the critical paths are as short as possible.

An example of retiming is shown in Figure A.2. If an operation has registers on all of its inputs, those registers can be moved to the output of the operation without changing its functionality. Operation “a” meets this requirement, so its input registers “r1” and “r2” can be moved to the position “r4”. Similarly, if the output of an operation drives a register (and nothing else), that register may be moved backwards to the inputs. In this example, “r3” on the output of operation “d” may be moved back to create registers “r5” and “r6”. If all operations have a propagation delay of 1, the top circuit has a minimum clock period of 2 (which can be seen from the path (r1, a, d, r3)), while the retimed circuit has a minimum clock period of 1.

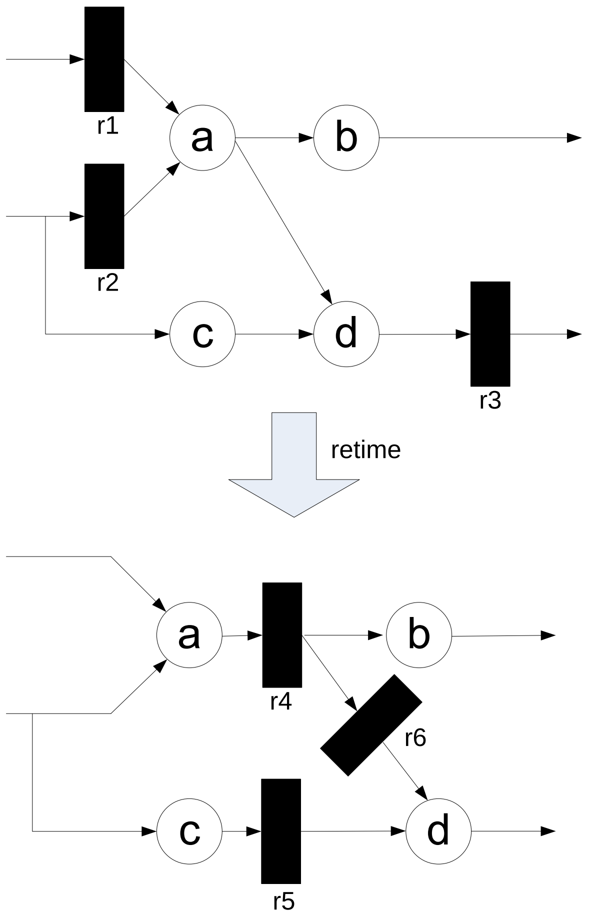


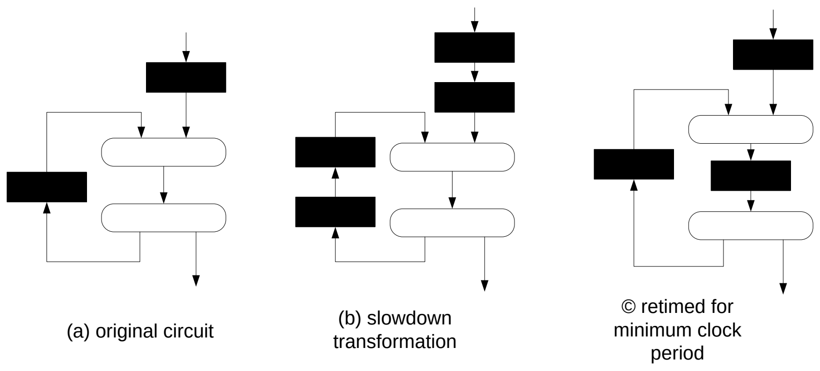
Figure A.2: Retiming a circuit. Rectangles represent registers and circles represent operations. Registers r1 and r2 in the top circuit are moved *forward* to create r4 in the bottom circuit; register r3 in the top circuit is moved *backward* to create r5 and r6

Feedback loops, or cycles, within the circuit are usually the limiting factor in retiming. Leiserson proved that the number of registers around a cycle cannot be changed without changing either the circuit's basic structure or behavior. The "average weight" of a cycle is defined to be the sum of the propagation delays through all operations on that cycle, divided by the number of registers on the cycle. Papaefthymiou proved that the clock period of a retimed circuit could not be less than the maximum average-weight of any cycle within the original circuit [69]. Suppose that you have a circuit where all operations have the same unit propagation delay, and that, for some reason, you require a clock period equal to that propagation delay. But if the original circuit contains a cycle with an average weight greater than one, it would appear that achieving this objective through retiming would be impossible since the minimum clock period cannot be less than the average weight cycle.

This problem can be circumvented (at a price) through a synchronous circuit transformation called slowdown [70] . In slowdown, each register in the original circuit is replaced by a sequence of c registers, producing what is known as a c-slow circuit. The resulting circuit is then retimed to distribute the registers and minimize the clock period (Figure A.3). As long as the maximum average weight cycle of the c-slow circuit is less than or equal to 1, the retimed, c-slow circuit will be able to execute with the desired clock period of 1. The penalty of slowdown, though, is suggested by its very name: if the original circuit were able to accept input values on every cycle, the retimed, c-slow circuit would only able to accept inputs and produce outputs every c cycles.

Since a c-slowed circuit can accept new input data every c cycles, it is equivalent to a pipeline schedule with a data introduction interval of c. However, slowdown technique can only be applied to a scheduled circuit. Pipeline scheduling, on the other hand, constructs such a c-slowed schedule from an unscheduled, functional dependence graph.

Module selection between circuit modules of the same type but with different levels of pipelining is very similar to the retiming technique discussed previously. By different levels of pipelining a circuit module, it is equal to moving the registers out



(a) original circuit
(b) slowdown transformation
© retimed for minimum clock period

Figure A.3: Slowdown and retiming. In the original circuit (a), each operation (white rectangle) has a delay of 1, so the minimum clock period between the registers (black rectangles) is 2. After a 2-slowdown transformation (b) followed by a retiming (c) the minimum clock period has been reduced to 1

of the circuit module into the circuit module. This reduces the combinational delay of the circuit module and potentially increases the system frequency.

Pipeline scheduling with combined module selection between differently pipelined circuit modules are functionally the same as a c-slow retiming. C-slow and retiming are very useful in increasing the system performance. However, they can only be applied to a scheduled or even implemented circuit. Pipeline scheduling and module selection achieve the same effects by constructing such an implementation from a higher level. Thus, pipeline scheduling and module selection are very important to generate high quality implementations, and the resulting circuit is predictable due to the well-characterized circuit module library.

Appendix B

Supporting Functions in HOIMS

B.1 *MinDist* Matrix Calculation

Algorithm B.1 illustrates the details to compute the *minDist* matrix and check the pipeline schedule validity of the input system data introduction interval for the input SCC. The algorithm begins by initializing *minDist[i,j]* with the minimum permissible time interval between i and j considering only the edges from i to j. If there is no such edge, *minDist[i,j]* is initialized to be $-\infty$ (see line 2). If $e_{(i,j)}$ is an edge from i to j, *minDist[i,j]* is initialized to be the effective delay from i to j (see line 1), refer to Equation 3.8 for computing the effective delay). Once the *minDist* matrix is initialized, the longest path between all nodes in the directed graph is used to compute the *minDist* matrix for the candidate δ (see line 3, 4 and 5). If *minDist* has a positive entry along the diagonal, the candidate δ is invalid (see line 6) and a larger δ should be tried.

The minimum permissible system data introduction interval can then be obtained by iteratively checking candidate δ using Algorithm B.1 for all SCCs in the dependence graph. Starting from $\delta = 1$, if any diagonal entry of the *minDist* for any SCC is positive, δ is incremented by 1 and Algorithm B.1 is run with new δ. This process is repeated until all diagonal entries are not positive and the current δ is the minimum system data introduction interval.

Algorithm B.1: minDist calculation and δ validity check algorithm

```
  input  : $\delta$, $SCC(V, E)$
  output: $true$ if $\delta$ is valid for this $SCC$,
          $false$ if $\delta$ is invalid for this $SCC$

  for $i \leftarrow 1$ to $|SCC|$ do
      for $j \leftarrow 1$ to $|SCC|$ do
          if $e_{(i,j)} \in E$ then
1             $minDist[i, j] \leftarrow \lambda_{eff}(i, j)$;
          else
2             $minDist[i, j] \leftarrow -\infty$;

  for $k \leftarrow 1$ to $|SCC|$ do
      for $i \leftarrow 1$ to $|SCC|$ do
          for $j \leftarrow 1$ to $|SCC|$ do
3             $dist \leftarrow minDist[i, k] + minDist[k, j]$;
4             if $dist > minDist[i, j]$ then
5                 $minDist[i, j] \leftarrow dist$;
6                 if $(i = j)$ and $(dist > 0)$ then
                      return $false$;

  return $true$;
```

B.2 Scheduling Priority Calculation in Pipeline Scheduling

The IMS algorithm uses a priority function that is a direct extension of the height-based priority [40] that is popular in acyclic list scheduling [41]. For acyclic list scheduling, the height-based priority of an operation P, Height(P), is defined as:

$$\text{Height(P)} = \begin{cases} 0, \text{ if P is the STOP pseudo-op} \\ \max_{\text{Q} \in \text{Succ(P)}} (\text{Height(Q)} + \text{Delay(P,Q)}), \text{ otherwise.} \end{cases} \tag{B.1}$$

This priority function has two important properties. First, since it computes the longest path from P to the end of the graph (the STOP pseudo-operation), the larger Height(P) is, the smaller is the amount of slack available to schedule operation P. It is well known that giving priority to operations on the critical path is important to achieving a good schedule, and the height-based priority function does so.

From the viewpoint of efficiency, it is preferable to schedule operations in some topological sort order so that each operation is scheduled before any of its successors. The second nice property of the height-based priority function is that it defines a topological sort. A predecessor operation will have a larger height-based priority than every one of its successors.

Extending the height-based priority function for use in IMS requires that we take into account the inter-iteration dependences. Consider a successor Q of operation P with a dependence edge from P to Q having a distance of D. Assume that the operation Q that is in the same iteration as P (the current iteration) has a height-based priority of H. Since P's successor Q is actually D iterations later, its height-based priority, relative to the current iteration, is effectively H-δ*D. Then the priority function used for iterative modulo scheduling, HeighR(), is given by

$$\mathrm{HeightR(P)} = \begin{cases} 0, \text{ if P is the STOP pseudo-op} \\ \max_{\mathrm{Q} \in \mathrm{Succ(P)}} \mathrm{(HeightR(Q) + EffDelay(P,Q))}, \text{ otherwise.} \end{cases} \tag{B.2}$$

Appendix C

Resource Sharing Overhead for FPGAs

Resource sharing is less common in architectural synthesis approaches that target FPGAs, due to the higher cost of multiplexing and interconnecting resources. Although the sharing costs within FPGAs is higher than custom circuits such as ASICs, resource sharing can be used on FPGAs when the resources saved by sharing is larger than the sharing overhead. For example, resource sharing can be justified for large circuit elements such as multipliers, large memories, and other circuits that consume a large amount of expensive FPGA resources.

Resource sharing overhead includes both area and time. Area overhead characterization assures the resource sharing algorithm that the resources saved by sharing is larger than the overhead area itself. The timing of sharing overhead must also be characterized so that the extra hardware for resource sharing does not become the critical path of the implementation.

To steer the input data to the shared circuit module, multiplexers and control logic must be employed to guide the correct input data into the shared module at correct time. Figure 2.3 illustrates an example of sharing two multiplier operations with one multiplier circuit module. As Figure 2.3 shows, the area overhead for resource sharing is made up of two parts: the multiplexer and the controller. This section will discuss the area and timing overhead of these two parts in detail.

C.1 Multiplexer Area

The primary cost of resource sharing in FPGAs are the multiplexers needed to steer data to the shared resource. Although multiplexers are relatively expensive in FPGAs, they are still much cheaper than large modules such as multipliers, RAMs etc. Further, small multiplexers can often be merged into underutilized LUT resources. The small relative size of multiplexers and the ability to merge multiplexing suggests that resources sharing is feasible for FPGA circuits.

The multiplexer area cost in FPGAs is carefully characterized in this work. Multiplexers of different input port number (N) and port width (W) are automatically generated and run through FPGA implementation tools. The selection line width (S) of the multiplexer equals $\lceil log_2(N) \rceil$. The BCD (Binary Coded Decimal) value of selection line is used to directly select from the input ports as the output. For example, if there are seven input ports (N = 7), three selection lines will be needed (S = 3). When S="101", input port 5 will be selected as the output. Figure C.1 plots the area of 1-bit multiplexers for a range of input port numbers. The X axis represents the number of input ports (excluding the select line) of the multiplexer, starting from 2 to 256. The Y axis represents the FPGA LUT count. Figure C.2 plots the relationship between the input port width and the multiplexer area for several multiplexers with different input port number. The X axis represents the bit-width of input ports and the Y axis represents the FPGA LUT count.

The multiplexer area in FPGAs is linear to the port width, but not linear to the number of input ports. As Figure C.2 shows, for most N-input multiplexer, the relationship between the input port width and the area cost is linear. Although a few multiplexers, such as for N=9,10,11, are not in accordance with this linear relationship, the relative deviation is very small. The average deviation from a linear estimation and the actual area cost, for multiplexers with input port number from 2 to 256 and bit width from 2 to 32, is only 1.74%. Thus, a general equation for

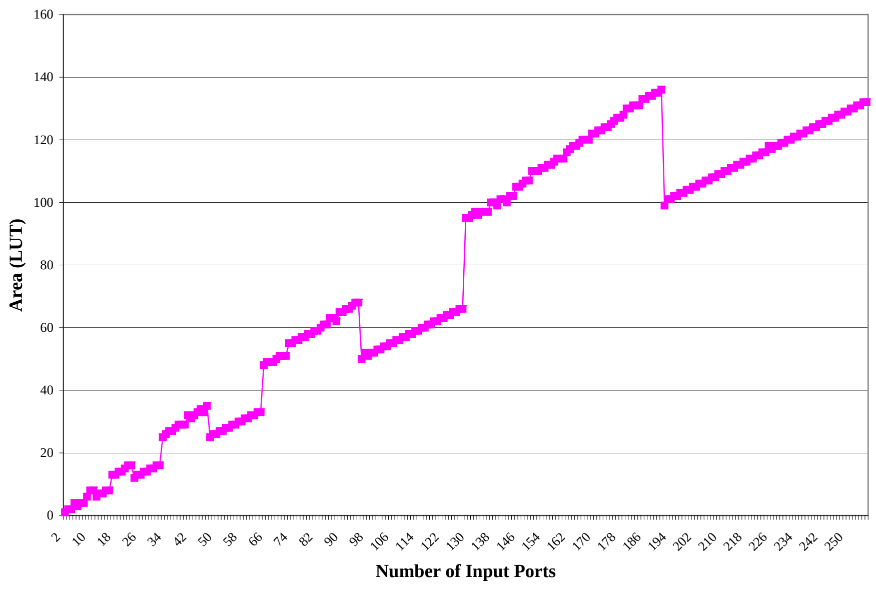


Figure C.1: Relationship between the number of input ports (N) and area cost in FPGA LUT count for N-input 1-bit multiplexers

computing the area cost for an N-input, W-bit multiplexer from an N-input, 1-bit multiplexer can be linear:

$$Area_{MUX_N(W)} = W * Area_{MUX_N(1)} \tag{C.1}$$

where $Area_{MUX_N(W)}$ represents the area cost of an N-input multiplexer whose bit-width is W. However, the linear equation does not hold for the relationship between input port numbers and area cost for a 1-bit multiplexer though, as Figure C.1 shows. This is caused by the specific architecture feature of Xilinx FPGAs: the 4 input LUT and the internal multiplexers within each slice.

Multiplexers in FPGA are as expensive as an adder, but much cheaper than large circuit elements. For example, a 2-input 16-bit multiplexer costs 16 LUTs, which is equal to 8 slices. This is as expensive as a 16-bit two-input ripple carry adder. For sharing an M-input circuit module, M multiplexers are needed. This makes sharing an adder twice as expensive as using two dedicated adders. However, multiplexers

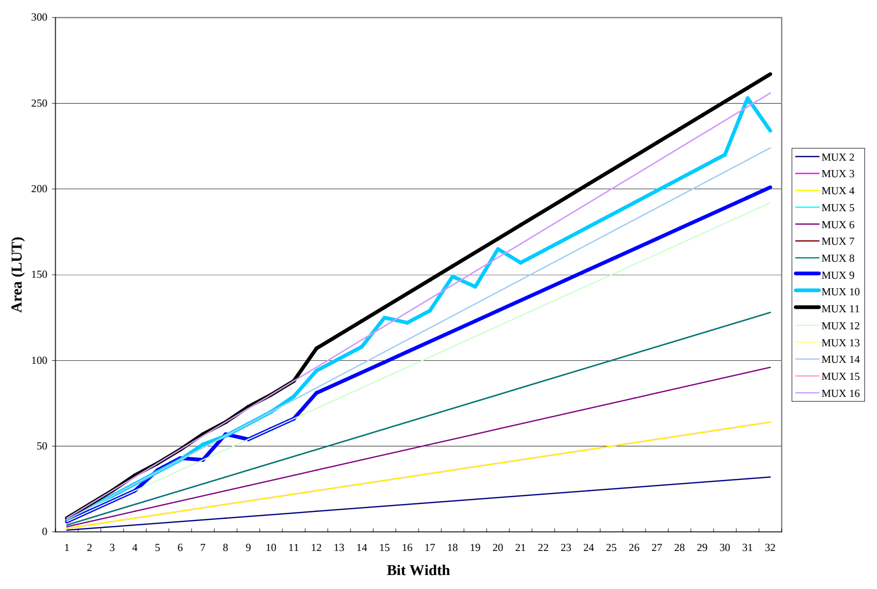


Figure C.2: Relationship between the bit width (W) and area cost in FPGA LUT count for several N-input multiplexers

are much smaller than the multiplier circuits in FPGA. For example, sharing a 16-bit *Array Multiplier 1* requires two 16-bit multiplexers, which cost 16 slices. This is less than 10% of allocating a new dedicated multiplier. Thus, sharing large circuit modules such as multiplier, block RAMs, etc in FPGA can greatly reduce the overall area cost.

In summary, multiplexers are essential parts for resource sharing. Although it is too expensive to share an adder and other simple logic functions in FPGAs, it is very cheap to share large circuit modules such as multipliers, RAM blocks etc. The relationship between the input port number and the multiplexer area cost is hard to be represented with a simple mathematical function, due to the regular architecture of FPGA based on look up tables. The relationship between the port width and area for fixed input port number is very close to linear. Thus, 1-bit multiplexer area is characterized and used as the basis for estimating arbitrary bit width multiplexer area cost.

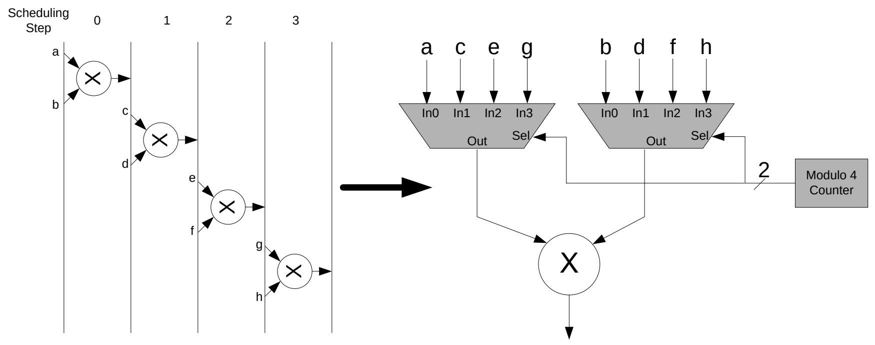


Figure C.3: The simplest controller to share 4 operations with 1 circuit module

C.2 Controller Area

A multiplexer requires an external controller to choose the appropriated input port at each clock cycle. This is implemented by using the controller to generate the correct value at every clock cycle, and driving the multiplexer's selection line with these values. Under the context of pipeline scheduling, the modulo start time of each operation is within 0 and system data introduction interval minus 1 (i.e. $0 \leq t_s \leq \delta - 1$). So the controller only needs to have δ states, and for each state, it generates the multiplexer selection line value according to the scheduling information in each state.

In the simplest form, a modulo-δ counter can be used as the controller. The output of the counter is connected directly to the multiplexer selection line input. This is illustrated with an example in Figure C.3. In this example, circuit module m is shared between four operations. The system data introduction interval is 4. The scheduling for operation *op1*, *op2*, *op3* and *op4* is 0, 1, 2 and 3 respectively. As the figure shows, the modulo-4 counter output is directly connected to the 4-1 multiplexor's selection line, which selects the corresponding input at every pipelining scheduling step. For 4-input LUT based FPGA architecture, a modulo M counter costs $\lceil \lceil log_2(M) \rceil /2 \rceil$ slices.

If the circuit module is not reused in every pipeline phase, the input port number of its corresponding multiplexer need not be as large as the pipeline stage length (i.e. the system data introduction interval). According to Figure C.1 and Figure C.2, the reduction in the input port number can save considerable amount of hardware area. For example, if δ is 8, and the circuit module is only shared by 4 16-bit operations. Reducing the input port number from 8 to 4 can save 64 LUTs in total. The cost for this area reduction is an encoder. The input width of the encoder is the same as the output width of the counter ($w_\delta = \lceil log_2(\delta) \rceil$). The output width of the encoder is $w_s = \lceil log_2(N_s) \rceil$, where N_s is the number of operations sharing the circuit module. For FPGAs with a 4 input LUT architecture, a w_δ to w_s encoder costs:

$$2^{max((w_\delta - 4),0)} * w_s(\text{LUTs}). \tag{C.2}$$

For the previous example, a 3 to 2 encoder is needed, which only costs only 2 LUTs but saves 64 LUTs.

C.3 Timing Overhead

Resource sharing overhead comes not only in the form of extra hardware area, but extra time also. To make resource sharing working correctly and meeting the system frequency requirement, the timing of the multiplexer and controller deserves careful study.

The combinational delay grows with the increment of the number of multiplexer input ports. Figure C.4 shows the relationship between the number of multiplexer input ports N and its corresponding combinational delay. As the figure shows, the multiplexer combinational delay increases in general with the number of input ports because more stages of logic are needed. However, due to the logarithm relationship between the number of logic stages and the number of input ports, and the special architectural structure of FPGAs, even a very wide multiplexer is very fast.

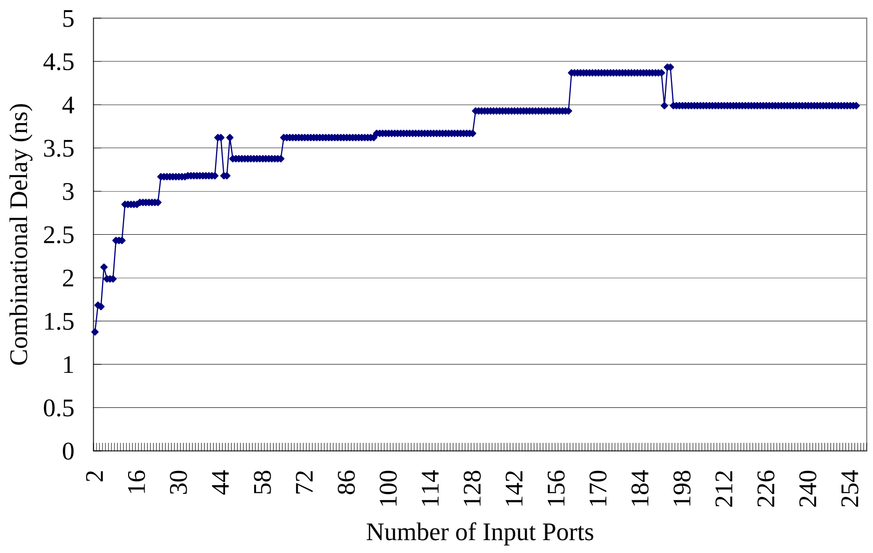


Figure C.4: Relationship between the number of input ports (N) and combinational delay for N-input multiplexers

Although not shown in the figure, the characterization shows that the bit width of the multiplexer input ports does not affect the timing of the multiplexer with the same number of input ports.

The timing of the controller should include the modulo δ counter and the δ inputs encoder. The critical path of a modulo δ counter is a $\lceil log_2(\delta)\rceil$ bit adder. Even if the δ is very large, for example 256, only a 8-bit adder is needed. It is clear that the modulo δ counter won't become the critical path of the circuit. Since the counter output width is very small, the level of LUT tree for the encoder is very limited, so the encoder won't become the critical path of the circuit either.

Appendix D

ASAP Exploration Algorithm

The first algorithm tried for combining scheduling, module selection, and resource sharing is a recursive branch and bound algorithm based on pipeline scheduling with ASAP priority. This scheduler is a relatively simple adaptation of a non-pipelined ASAP list scheduler to incorporate modulo resource constraints. The objective of this approach is to identify the lowest area cost architectural solution that meets the constraints of a given (f, δ) pair. This algorithm, summarized in Algorithm D.1, can be run multiple times to identify solutions with different (f, δ) pairs for a fixed throughput constraint.

D.1 Algorithm Summary

The outer loop of this algorithm uses pipeline scheduling with ASAP priority. Each operation n will be scheduled as soon as possible when all its predecessors are scheduled (line 3 and 4, S is the set of unscheduled operations). After the operation has been scheduled, it is bound to a circuit module before proceeding to another operation in the dependence graph. The binding of an operation first tries to share any previously allocated modules (lines 5–9). After exhausting all resources sharing possibilities, a new circuit module is allocated for the operation (lines 10–16). *Every* potential circuit module that is compatible with this operation is explored. This process proceeds recursively for all remaining operations (lines 9 and 16).

To limit the search space, four bounding functions are used to prevent the search from pursuing unnecessary paths. The first bound eliminates the paths when

Algorithm D.1: ASAP Exploration

```
1  Data: $(f,\delta)$, $\lambda_{max}$
   ASAP_EXPLORE($S$,$C_{curr}$,$C_{best}$) begin
2      if $S \neq \phi$ then
3          Select a schedulable node $n \in S$;
4          Modulo Schedule $n$ ASAP and remove $n$ from $S$;
5          forall $p \in \mathcal{Q}(n)$ do
6              if not SHARABLE($p$) then continue;
7              Reschedule $n$ if necessary;
8              $\mathcal{M}(n) = p$;
9              ASAP_EXPLORE($S$,$C_{curr}$,$C_{best}$);
               $\mathcal{M}(n) = \phi$;
               Restore schedule $n$;
           end
10         forall $m \in \mathcal{T}(n)$ do
11             $\mathcal{M}(n) = m$;
12             if $f_m < f$ then continue;
13             if $\delta_m > \delta$ then continue;
14             if ESTIMATE_COST()$> C_{best}$ then continue;
15             if ESTIMATE_LATENCY()$> \lambda_{max}$ then continue;
               $C_{curr}$+=COST($m$);
16             ASAP_EXPLORE($S$,$C_{curr}$,$C_{best}$);
               $\mathcal{M}(n) = \phi$, $C_{curr}-$ =COST($m$);
           end
           unschedule $n$ and add $n$ back to $S$;
       else
17         if $C_{curr} < C_{best}$ then $C_{best} = C_{curr}$;
       end
   end
```

$f_m < f$ (line 12). The second eliminates the paths when $\delta_m > \delta$ (line 13). These two bounding functions reflect the constraints of pipeline scheduling design space on the module selection. The third estimates a lower bound on the area cost of the current branch and eliminates the paths when it is larger than the current best area cost (line 14). ESTIMATE_COST() assumes maximal resource sharing or lowest cost module selection for all unscheduled operations, whichever is smaller. The fourth eliminates the branches when the current branch violates the latency constraint (line

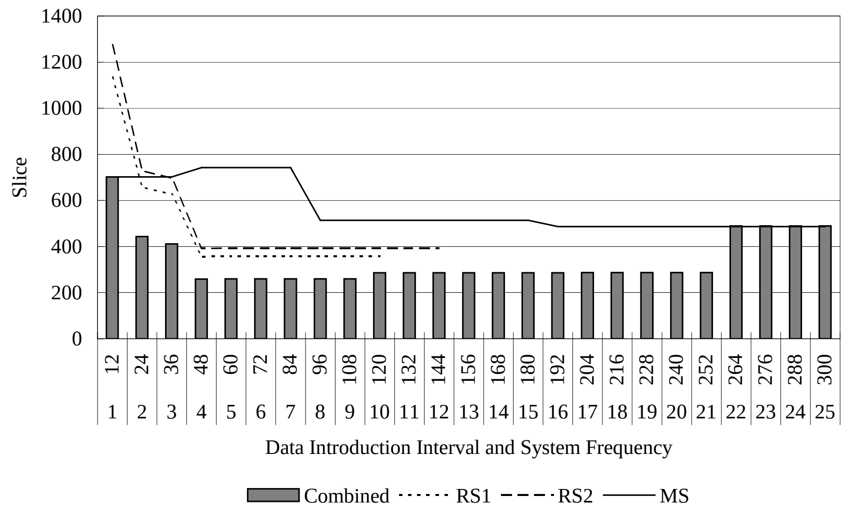


Figure D.1: *ASAP Exploration* area result with different synthesis techniques for the "Color Space Conversion" example under a 12 MSamples/Sec throughput constraint

15). ESTIMATE_LATENCY() assumes ASAP scheduling with minimum latency circuit modules for all unscheduled operations.

D.2 ASAP Exploration Results

A number of signal processing kernels were tested using this ASAP exploration algorithm. The circuit modules listed in Table 5.1 were used for module selection and resource sharing. To demonstrate the importance of combining module selection with resource sharing, three synthesis methods were tested as described in Section 6.1. For resource sharing only method, *RS1* uses *Array Multiplier 3* and *RS2* uses *Booth Multiplier 3* as the fixed module selection. Each method explores all possible (f, δ) pairs that meet the throughput constraint of 12 MSamples/Sec. For each (f, δ), the best pipeline schedule was created and the area result is listed.

Figure D.1 to Figure D.5 shows the area results for five kernels by the four different exploration strategies. These algorithms are color space conversion, FFT transformation, FIR filter, linear interpolator and the IDCT transformation. In these figures, the area of the combined resource sharing and module selection is shown as

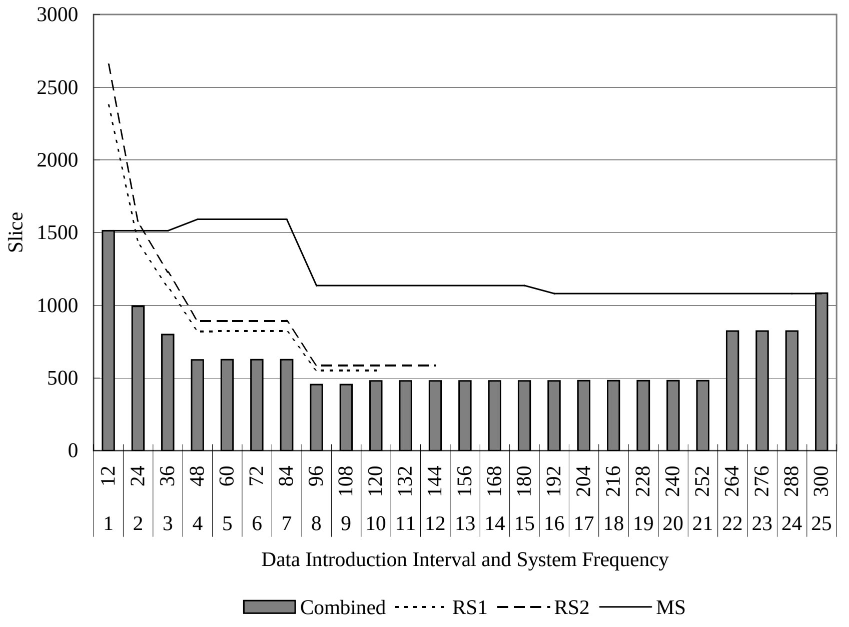


Figure D.2: *ASAP Exploration* area result with different synthesis techniques for the FFT example under a 12 MSamples/Sec throughput constraint

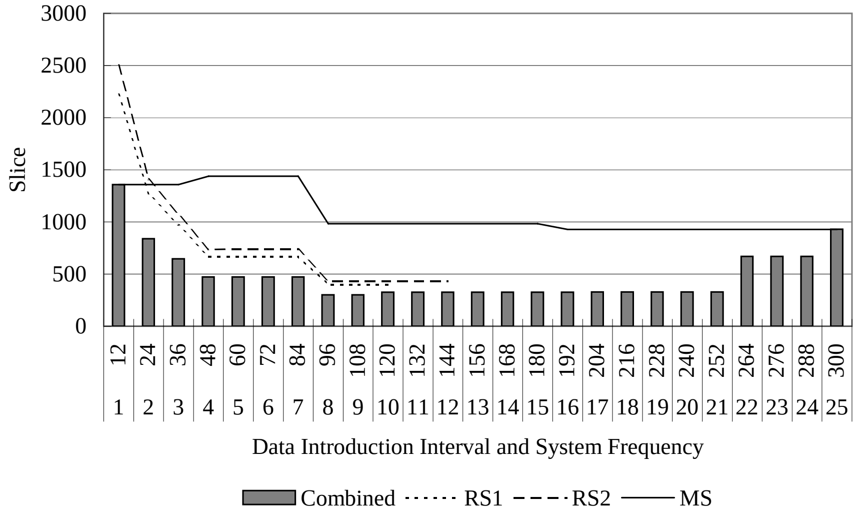


Figure D.3: *ASAP Exploration* area result with different synthesis techniques for the FIR example under a 12 MSamples/Sec throughput constraint

gray bars. The solid line shows the area cost with module selection only method. The two dotted lines represent the area cost of two different resource sharing only methods.

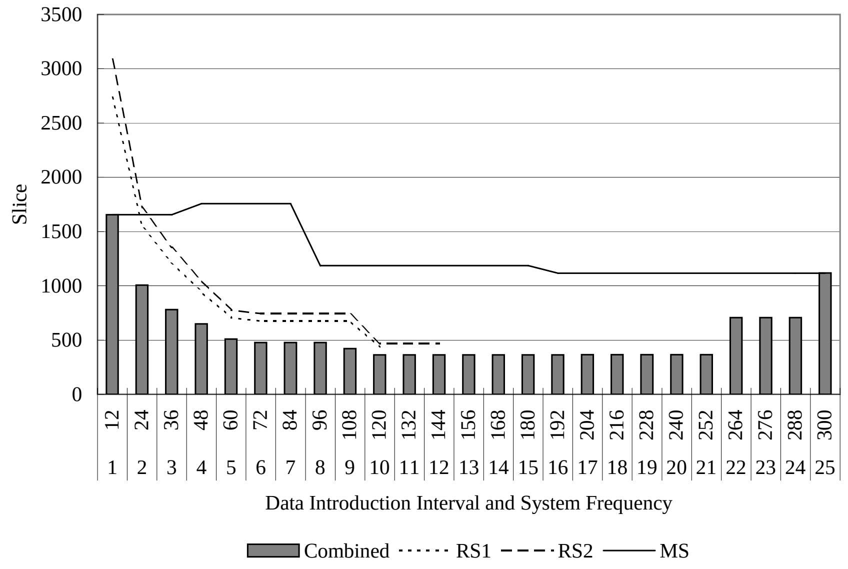


Figure D.4: *ASAP Exploration* area result with different synthesis techniques for the "Linear Interpolator" example under a 12 MSamples/Sec throughput constraint

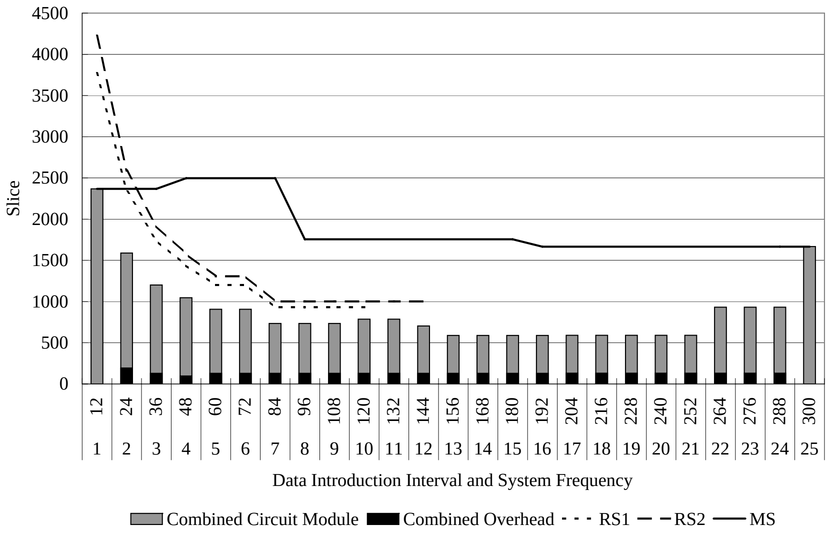


Figure D.5: *ASAP Exploration* area result with different synthesis techniques for the IDCT example under a 12 MSamples/Sec throughput constraint

D.3 Limitations

Although this branch and bound exploration algorithm identifies very efficient design solutions by combining module selection with resource sharing under an ASAP pipeline scheduling context, its computational complexity is prohibitively high. The use of an exhaustive algorithm to explore module selections will typically not be practical for large data-flow models and circuit module libraries. For example, design exploration for the IDCT model took almost six days to complete (see Table 6.4). A second disadvantage of this approach is the simplistic ASAP non-backtracking scheduler. This greedy scheduling algorithm may not be able to generate schedules for resource sharing or module selection alternatives that would otherwise reduce the overall system area cost.

Appendix E

IMS Exploration Algorithm

To address the limitations of the ASAP exploration approach, a second exploration strategy was developed which combines module selection with a backtracking scheduler based on IMS (Iterative Modulo Scheduling) [33]. IMS performs pipeline scheduling and resource sharing simultaneously by employing a modulo reservation table for each circuit module. The algorithm is iterative and allows backtracking (unschedule and reschedule) to find solutions that are otherwise not attainable for non-backtracking scheduling approaches such as the one used in the previous section.

The primary constraint for this algorithm is the minimum throughput T, for the data-flow specification. Same as the exact exploration algorithm, the iterative modulo exploration algorithm searches through a range of data introduction interval values (and the corresponding system frequency) to identify the best solution. This range is bounded by the minimum data introduction interval (δ_{min}) due to the recurrence constraint (see Equation 3.10) and maximum data introduction interval (δ_{max}) due to the maximum feasible system clock frequency of the circuit (see Equation 3.3).

E.1 Algorithm Summary

The outline of iterative modulo exploration algorithm is presented in Algorithm E.1. For a given throughput constraint (T) and a given module library, the algorithm computes the bounds on δ first. Then, the outer *for* loop (line 1) of this algorithm iterates over each feasible value of δ and computes the corresponding minimum operating frequency (line 2). The solutions for each (f, δ) pair is then identified by the inner loop of the algorithm (line 3 to 5) which performs module selection, pipeline scheduling and resource sharing. Finally, the best solution and the corresponding (f, δ) pair are picked (line 6) for the whole pipeline space.

Algorithm E.1: Iterative Modulo Exploration

```
  IMS_EXPLORE($T$) begin
      Calculate the $\delta_{min}$;
      Calculate the $\delta_{max}$;
1     for $\delta \leftarrow \delta_{min}$ to $\delta_{max}$ do
2         $f_\delta \leftarrow \delta * T$;
3         for $i \leftarrow 1$ to $K$ do
4             $ms \leftarrow$ Module_Selection($f_\delta$);
              if ESTIMATE_COST() $> C_{best\_solution}$ then continue; ;
              IMS();
              Update best solution for $(f_\delta, \delta)$ ;
5         Record the current best solution;
6     Compare and pick the best solution;
  end
```

There are two important differences between this exploration strategy and the previous approach. First, this approach performs module selection *before* scheduling (see line 4 of Algorithm E.1). This significantly reduces the search space for scheduling and resource sharing. Second, this approach does not explore the full module selection design space. Instead, a finite number of "good" module selection possibilities are identified. The size of this finite search space can be adjusted by the user to balance execution time with the quality of the result (see line 3, which is detailed next).

E.2 Heuristic Module Selection

The inner loop of this algorithm performs module selection followed by IMS (see Algorithm 3.1) which simultaneously performs pipeline scheduling and resource sharing. As described earlier, performing module selection before resource sharing can greatly limit the resource sharing design space. However, the whole module selection space is too big, so a good coverage of module selection space is very important for a heuristic algorithm. In this approach, an heuristic algorithm is used to identify "good" module selection choices early in the exploration process.

The design space for module selection is significantly reduced by limiting the number of modules that can be considered for each operation in the dependence graph. This heuristic algorithm will select at most two candidate circuit modules for each operation. With only two module possibilities for each operator, the module selection design space is reduced to 2^N where N is the number of operations in the dependence graph. While still exponential in size, it is significantly smaller than the design space described in section 2.5.

To further reduce the module selection space, not all 2^N module selection possibilities are searched. Instead, K unique module selection points will be selected from the more limited 2^N design space (see line 3). The constant K can be selected by the user to control the size of the design space. Experiments have shown that $K = N log_2 N$ is sufficient for finding good results, where N is the number of operations in the dependence graph. Other more targeted heuristic approaches for module selection will be introduced in next section.

Because a smaller module selection design space is searched, it is very important to select "good" subset of modules for each operation. The first step in this selection process is to eliminate all circuit modules for consideration that cannot be used under the system design constraints. These include those modules with an insufficient data introduction interval ($\delta_m > \delta$) or and insufficient maximum operating frequency ($f_m < f_0$).

The next step is to choose two circuit modules from the remaining compatible circuit modules. The first module chosen is the *smallest* circuit module that is compatible with the given operation. This selection limits the sharing of the circuit module with other operations as smaller circuit modules usually have a larger δ_m, or it is very specific to that operation. The second module chosen is the circuit module with the smallest *amortized* cost (see Equation 5.6) and encourages more sharing.

E.3 Scheduling and Resource Sharing

Before proceeding to scheduling, a best case resource estimate will be computed on the current module selection. This is similar to the estimate used in Algorithm D.1. If the best case area estimate of the module selection is larger than the current solution, the module selection is discarded and IMS scheduling is skipped. The module selection process is repeated rather than proceeding to scheduling with an inferior module selection.

Once a candidate module selection has been made, the algorithm proceeds to scheduling. As described earlier, iterative modulo scheduling is then used to schedule the operations of the graph using the modules chosen in the previous step. After scheduling, a more accurate estimate of the circuit area can be made by taking into account both the data-path and resource sharing overhead cost. This area estimate is compared against the best solution for the current value of δ. If the estimate is smaller than the current solution, the best solution is updated.

E.4 *IMS Exploration* Results

The iterative modulo exploration algorithm was applied to two sample data-flow models, a Biquad IIR filter and an 8-tap FIR filter. Both models use 16-bit precision to fully utilize circuit modules in Table 5.1. The throughput constraint for both models is set at 12 MSamples/Sec. A large number of architectures were synthesized under the constraint for each data-flow model. The architectural results obtained from this synthesis flow are shown in Figure E.1 for the FIR filter and in

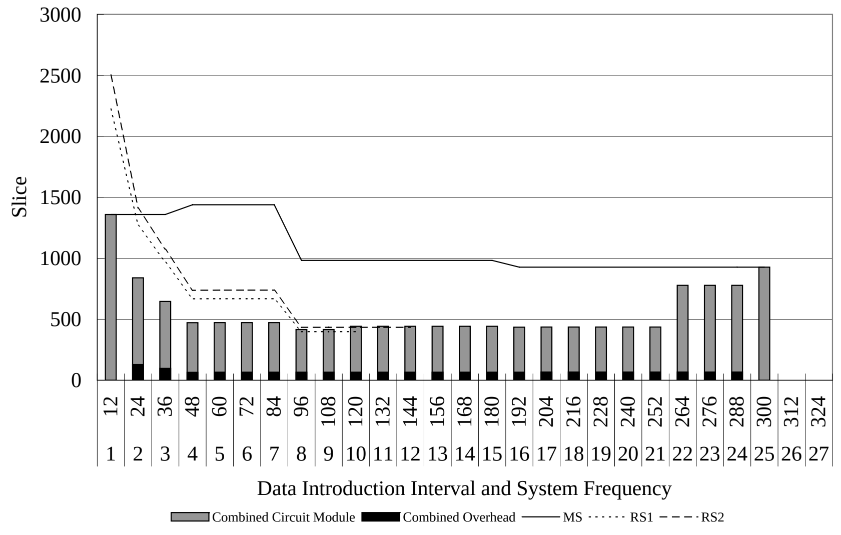


Figure E.1: Area comparison between different techniques for FIR filter with 12M Samples/Sec throughput constraint

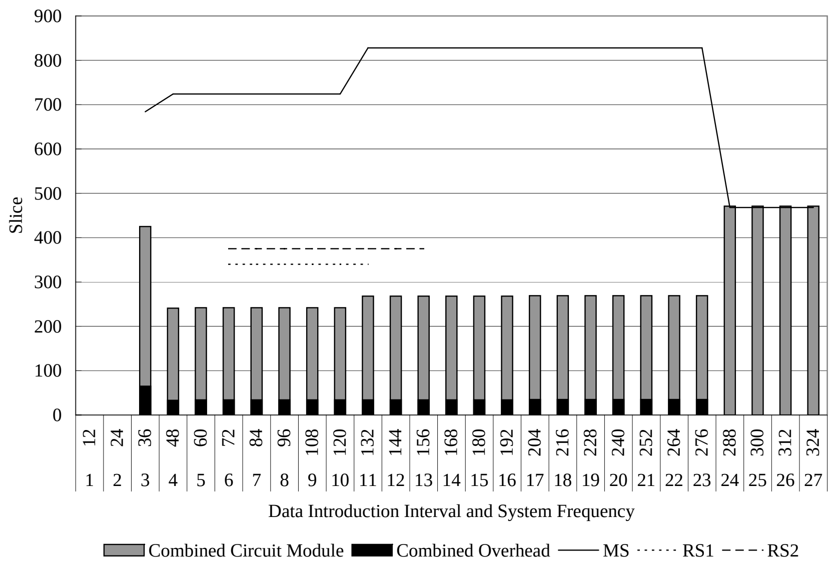


Figure E.2: Area comparison between different techniques for Biquad filter with 12M Samples/Sec throughput constraint

Figure E.2 for the Biquad filter. The meaning of the bars and lines in these figures are the same as in Figure 6.2 and was described in Section D.2.

The minimum feasible δ for the Biquad example is 3 due to the feedback constraint within the model. Without any feedback constraints, the feed-forward

FIR filter may operate at a minimum $\delta = 1$. In Figure E.1, the maximum $\delta = 25$ corresponds to a system frequency of $f = 300$ MHz, no circuit modules operate at $\delta = 26, 27$ with corresponding system frequency of $f = 312, 324$ MHz. Although additional design points exist beyond these limits, they were omitted for brevity.

Both results exhibit the "bathtub" shape of the pipeline synthesis design space described in Section 6.1. The best solution for the Biquad example occurs at ($f = 44$ MHz, $\delta = 4$). At this design point, only one relatively low-area multiplier (*CoreGen Multiplier 1*) is allocated and fully shared. Although such sharing involves multiplexing overhead, the relatively low value of δ requires a simple controller. With larger values of δ, more costly multipliers are required and the complexity of the sequencer increases. At the ($f = 264$ MHz, $\delta = 24$) design point, only the *CoreGen Sequential* will operate. Each operator is allocated a dedicated multiplier due to the resource sharing limitation of the sequential multiplier module.

The best solution for the FIR example occurs at ($f = 96$ MHz, $\delta = 8$). At this point, one multiplier (*CoreGen Multiplier 1*) is shared among the eight operators. Design points with $\delta < 8$ require more multipliers that are clocked slower than possible. At higher values of δ (up to $\delta = 21$), the sequencing overhead area increases slightly with no increase in module or sharing costs. At ($f = 264$ MHz, $\delta = 22$) design point, more expensive higher speed multipliers are required.

The module selection only results (solid line) shows that exploiting module selection only is inferior to the best solutions from the combined approach. The difference in area cost between the module selection only technique and the combined technique is due to the limited resource sharing capability of the circuit modules found with module selection only technique. For example, at δ=8, module selection only will choose the *CoreGen Sequential* multiplier, but the combined technique will choose the *CoreGen Parallel 1* multiplier. Although the parallel multiplier is more expensive than the sequential multiplier, the larger resource sharing capability of the parallel module offsets the increased circuit module cost, and yields a lower overall system area cost.

The resource sharing only (dotted lines) results in these figures show that the design space associated with this technique is significantly limited by the module chosen. For example, RS1 in the Biquad example uses the *Array Multiplier 3*. With a latency of 4, the circuit must operate at a minimum $\delta = 6$ due to the feedback loop. Because the fastest operating frequency of this module is 127 MHz, the maximum δ is limited to 13.

E.5 Comparison Between *ASAP Exploration* and *IMS Exploration*

Figure E.3 plots the ASAP exploration result and the IMS Exploration result for the FIR filter using the same throughput constraint (12 MSamples/Sec). Between δ from 1 to 7, both exploration approaches generate the same result, because the heuristic module selection algorithm for the Iterative Modulo Exploration results in only one circuit module for the multiplier operators (*CoreGen Multiplier 1*), which is the same module selection found with the exact exploration. Between δ equals to 8 and above, the exact Exploration approach generates more area efficient architectural solutions than the Iterative Modulo Exploration approach. With these δ values, there are more than one candidate circuit modules for each multiplier operator. Since the heuristic module selection algorithm doesn't explore all the module selection possibilities, it does not find the same solutions as the exact Exploration approach. For example, at (f = 96 MHz, δ=8), the exact exploration selects *CoreGen Parallel 1* multiplier for all the multiplier operators, while the IMS exploration selects *CoreGen Parallel 1* and *CoreGen Sequential* multipliers, which results in under-utilization of the circuit modules and thus higher overall area cost.

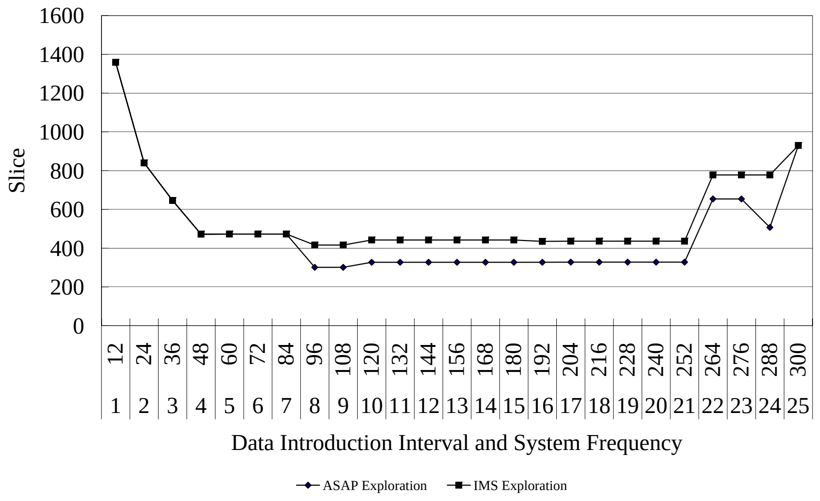


Figure E.3: Area comparison between ASAP and IMS exploration approach for FIR filter with 12 MSamples/Sec throughput constraint

E.6 Limitations

The iterative modulo exploration algorithm explores the combined design space much more effectively than the ASAP exploration algorithm, while yielding similar results. It is an implementation of the separated module selection and resource sharing strategy, while resource sharing is implicitly performed during pipeline scheduling. The major disadvantages of this algorithm are: First, it does not differentiate between different sharing. Although resource sharing overhead is counted, different sharing possibilities are treated as the same. However, different sharing can result in different interconnection architecture and control architecture. Second, the results of resource sharing/pipeline scheduling is not fully utilized by the module selection algorithm for future iteration's refinement. Feeding back the result to module selection can greatly improve the module selection search efficiency. Third, it assumes that all operations in the dependence graph are of the same bit-width, this simplifies module selection and resource sharing, but may results in inefficient module utilization.