5-2010

A Finite Domain Constraint Approach for Placement and Routing of Coarse-Grained Reconfigurable Architectures

Rohit Saraswat
Utah State University

A FINITE DOMAIN CONSTRAINT APPROACH FOR PLACEMENT AND ROUTING OF COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

by

Rohit Saraswat

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Electrical Engineering

Approved:

Dr. Brandon Eames Major Professor	Dr. Aravind Dasu Committee Member
Dr. Koushik Chakraborty Committee Member	Dr. Sanghamitra Roy Committee Member
Dr. Scott Budge Committee Member	Dr. Steve Allan Committee Member

Dr. Byron R. Burnham
Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010

Abstract

A Finite Domain Constraint Approach for Placement and Routing of Coarse-Grained Reconfigurable Architectures

by

Rohit Saraswat, Doctor of Philosophy

Utah State University, 2010

Major Professor: Dr. Brandon Eames
Department: Electrical and Computer Engineering

Scheduling, placement, and routing are important steps in Very Large Scale Integration (VLSI) design. Researchers have developed numerous techniques to solve placement and routing problems. As the complexity of Application Specific Integrated Circuits (ASICs) increased over the past decades, so did the demand for improved place and route techniques. The primary objective of these place and route approaches has typically been wirelength minimization due to its impact on signal delay and design performance. With the advent of Field Programmable Gate Arrays (FPGAs), the same place and route techniques were applied to FPGA-based design. However, traditional place and route techniques may not work for Coarse-Grained Reconfigurable Architectures (CGRAs), which are reconfigurable devices offering wider path widths than FPGAs and more flexibility than ASICs, due to the differences in architecture and routing network. Further, the routing network of several types of CGRAs, including the Field Programmable Object Array (FPOA), has deterministic timing as compared to the routing fabric of most ASICs and FPGAs reported in the literature. This necessitates a fresh look at alternative approaches to place and route designs. This dissertation presents a finite domain constraint-based, delay-aware placement

and routing methodology targeting an FPOA. The proposed methodology takes advantage of the deterministic routing network of CGRAs to perform a delay aware placement.

(190 pages)

To the two most important ladies in my life,
my mother Sushil and my wife Netra.

Acknowledgments

I will be eternally grateful to several people who directly or indirectly contributed to the success of this dissertation. First and foremost, I would like to thank my family for their support and encouragement. I am grateful to my parents who themselves set the standards so high that not pursuing a Ph.D. was never an option. Special thanks to my mom, who in spite of being on the other side of the planet, has always been close enough to support me through my struggles as a graduate student. I thank my wife, Netra, for her unconditional love, constant support, patience, and enthusiasm which kept my morale high and motivated me to keep going through to the end, even when I spent more time with my computer than with her. Honestly, it would take another dissertation to express my appreciation, love, and devotion to her. I also thank my sister, Jyoti, and my wife's parents for their support.

I express my gratitute and thanks to my advisor, Dr. Brandon Eames, for his vision, guidance, and patience through the last four years of graduate school. I thank him for making me aware of the pitfalls along the research path, but more so for letting me shoot myself in the foot for a first-hand experience. He is one the best teachers and researchers that I have come across in my student life. Working with him made me feel more like a fellow researcher than a student. He is truly a friend, philosopher, and guide (the last two not because he has a Ph.D. but because he is my advisor), from whom I have learned a lot in the classroom and beyond.

The other members of my committee have also been instrumental in enriching my graduate experience. I am grateful to Dr. Aravind Dasu who helped me achieve my goals through his encouragement and constructive criticisms. I am also thankful to Dr. Koushik Chakraborty and Dr. Sanghamitra Roy for providing valuable inputs that helped me think beyond my world of deterministic timing and for being as much friends as committee members. I also thank Dr. Steve Allan and Dr. Scott Budge for their influence on my graduate career.

I am also thankful to my friends and colleagues, especially Arvind, Shantanu, Prasad, Atul, and Sravanthi, who made my stay in Logan enjoyable. I would also like to acknowledge Mathstar and the ECE Department, which partially provided financial support for this research.

Rohit Saraswat

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

ACM	Association for Computing Machinery
ADRES	Architecture for Dynamically Reconfigurable Embedded Systems
ALAP	As Late As Possible
ALB	Arithmetic Logic Block
ALU	Arithmetic Logic Unit
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
BLB	Basic Logic Blocks
BPU	Basic Functional Unit
CFB	Configurable Functional Block
CGHRA	Coarse Grained and Hybrid Reconfigurable Architecture
CGRA	Coarse-Grained Reconfigurable Architectures
CLB	Configurable Logic Block
CLP	Constraint Logic Programming
CPE	Configurable Processing Element
DFG	Data Flow Graph
DFT	Discrete Fourier Transform
DMA	Direct Memory Access
DP	Data Ports
DP-FPGA	Data Path Field Programmable Gate Array
DPU	Data Processing Unit
DSP	Digital Signal Processing
DWT	Discrete Wavelet Transform
ECA	Elemental Computing Array
FB	Frame Buffer

FD	Finite Domain
FIFO	First In First Out
FIR	Finite Impulse Response
FPCA	Field Programmable Computing Array
FPGA	Field Programmable Gate Array
FPOA	Field Programmable Object Array
FPR	Field Programmable Gate Array Place and Route
FSS	Five Step Search
GA	Genetic Algorithm
GNU	GNU is Not Unix
GSL	GNU Scientific Library
HPWL	Half Perimeter Wire Length
IEEE	Institute of Electrical and Electronics Engineers
IFU	Interconnect Functional Unit
ILP	Instruction Level Parallelism
LL	Launch/Land
MAC	Multiply Accumulate
MB	Mega Bytes
MDCT	Modified Discrete Cosine Transform
MI	Memory Interface
MILP	Mixed Integer Linear Programming
MM	Matrix Multiplication
MP3	Moving Picture Experts Group Layer-3 Audio
MTAP	Multi Threaded Array Processor
MSMSM	Multi-Source Multi-Sink Maze
MWS	MP3 Window Subband
NN	Nearest Neighbor
NoC	Network-on-Chip

P&R	Place and Route
PE	Processing Element
PL	Party Line
POWV	Possible Optimal Wire length Vector
RAM	Random Access Memory
RAP	Reconfigurable Arithmetic Processing
RAW	Reconfigurable Architecture Workstation
RBI	Routing Based Interleaving
RC	Reconfigurable Cells
REMARC	Reconfigurable Multimedia Array Coprocessor
RF	Register File
RPU	Reconfigurable Processing Unit
RSA	Rectilinear Steiner Arborescence
SA	Schedule Analyzer
SAP	Simulated Annealing-Based Placement
SATD	Sum of Absolute Transformed Difference
SDP	Spreading Data Path
SIMD	Single Instruction Multiple Data
TDFG	Timed Data Flow Graph
TIERS	Topology Independent Pipelined Routing and Scheduling
VLIW	Very Large Instruction Word
VLSI	Very Large Scale Integration
VPR	Versatile Place and Route
WDM	Wiring Distribution Maps

Chapter 1

Introduction

Over the last few decades, computing needs have outgrown what general-purpose microprocessor devices can offer. Higher transistor density, smaller feature size, and increased clock frequencies have made microprocessors far more advanced than their predecessors. In spite of these improvements in computing technology, modern applications are becoming increasingly complex, necessitating the growth of high performance computing architectures. In order to target specific computing needs, the concept of Application Specific Integrated Circuits (ASICs) was introduced. ASICs are integrated circuits intended for specialized applications and they typically consolidate multiple functions into a single, high-speed device. However, despite their advantages, ASICs are limited to a specific target application and have limited flexibility and usability in other applications.

Field Programmable Gate Arrays (FPGAs) are at the opposite end of the Very Large Scale Integration (VLSI) spectrum, offering bit-level configurability. FPGAs are devices that consist of Configurable Logic Blocks (CLBs) and switch blocks. CLBs can be programmed to implement logic functions and multiple CLBs can be connected through configurable switch blocks and routing channels. The functionality of a FPGA can be modified at a later stage by reconfiguring the device. Bit-level granularity and reconfigurability makes FPGAs flexible for design prototyping and for applications that require in-field functional modifications. However, the flexibility of FPGAs comes at the price of large routing area overhead.

Coarse-Grained Reconfigurable Architectures (CGRAs) are reconfigurable devices which offer wider pathwidths than FPGAs and more flexibility than ASICs. In the VLSI spectrum, CGRAs are placed between ASICs and FPGAs due to their wider data paths, efficient coarse-grained Configurable Functional Blocks (CFBs), and fewer routing resources. Due

to these features, CGRAs offer ASIC-like computing power along with FPGA-like configurability. Figure 1.1 offers a relative comparison of flexibility and performance of various computational architectures.

In spite of these differences, a design must be placed and routed on the chip prior to use. Placement is the process of determining exact locations of circuit elements inside a chip's area. Once placement is complete, the circuit components must be connected. The procedure of establishing interconnections among the placed circuit components is called routing. Placement and routing are therefore common steps in the design flow of ASICs, FPGAs, and CGRAs.

The quality of placement and routing has a significant impact on the performance of a design. A poor placement can render a design unroutable or may lead to violations of performance requirements during routing. Even with a good placement, inefficient routing can impair the performance of an implemented design. Place and Route (P&R) techniques have received much attention in the past few decades and several methods have been proposed, a sampling of which is discussed in Chapter 2. Though some of these techniques can be applied to CGRAs, the objectives of placement and routing are different for CGRAs as

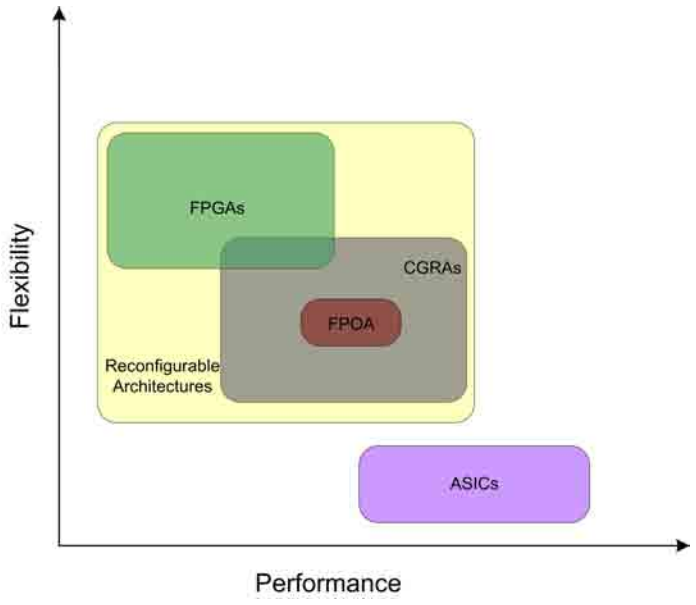


Fig. 1.1: Flexibility vs. performance for different types of architectures.

compared to the other two architectures.

1.1 Motivation of this Research

The performance of a design on an ASIC or FPGA is dependent on the path delay, which in turn is dependent on the wirelength of the critical path. Placement tools for ASIC/FPGA try to place the design such that the routing wirelength is minimized. This phase is important because a bad placement may prohibit the router from finding short paths, or in the worst case, may yield an unroutable design. Hence, significant emphasis is given on finding a good placement. However, the placement tool must be able to obtain some routability estimate to generate a good placement. Wirelength estimation is often used as the metric for guiding the placement process since it directly impacts the delay and is used in a variety of tools [1–3]. The same is not the case with CGRAs, such as the Field Programmable Object Array (FPOA). The interconnect network of an FPOA has deterministic timing with a predefined relation between wire segments and delay. Unlike ASICs and FPGAs, in CGRAs such as the FPOA, it is not the wirelength that decides the delay, but it is the delay that determines the wirelength. Hence, wirelength minimization is not the correct approach to solve the FPOA P&R problem.

Figures 1.2 and 1.3 illustrate the difference between the routing objectives of ASIC/ FPGA and FPOA. In fig. 1.2, each logic block must be placed such that the length of a route

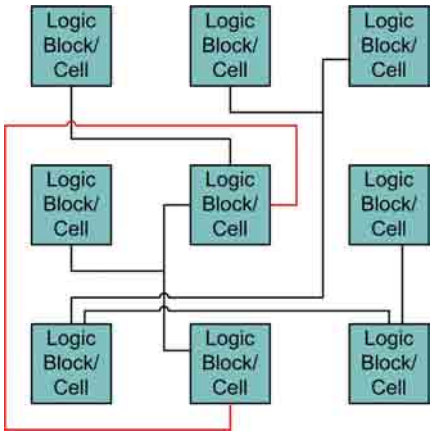


Fig. 1.2: ASIC/FPGA placement and routing objective.

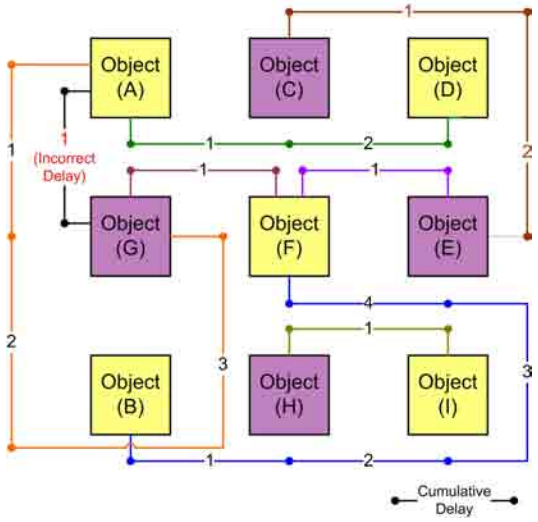


Fig. 1.3: FPOA placement and routing objective.

connecting any two logic blocks can be minimized, and routing complements placement by favoring the shortest candidate paths. In contrast, delay satisfaction takes precedence over wirelength minimization in FPOA P&R. Figure 1.3 shows a post P&R design on an FPOA with possible routes between various objects. Routes between object pairs (G, F), (F, E), and (H, I) have a unit delay which is satisfied by the shortest connections between the respective object pairs. However, if the input specification mandates a delay of three units between object pair (A, G), then the shortest connection offers an incorrect delay of one unit and results in an invalid route. Instead, a correct solution must use a longer route with a delay of three units between A and G.

Further, the source in both cases is different. A Steiner tree [4], or a routing graph, is the starting point for most P&R tools. Steiner trees offer the advantage of minimum total wirelength, which is a desirable goal in standard P&R. Moreover, in FPOA-based P&R, the timing of the input graph has already been established and is provided in the form of a scheduled Data Flow Graphs (DFG). Each node in a scheduled DFG is associated with a processing element or resource, and every edge is annotated with the required communication delay along that edge. Unlike ASIC/FPGA, where interconnect wirelength is minimized and the critical edge decides how fast the design operates, the length of an interconnect

in an FPOA is constrained to fall within an interval $[min\_length, max\_length]$, determined by the delay requirement imposed by the schedule. The problem is no longer to find the shortest path, but to find a path with length ℓ, such that $min\_length \leq \ell \leq max\_length$. Any value of wirlength that falls within these bounds is acceptable, since all such paths will support the required delay.

The P&R problem is further complicated by the heterogeneity of an FPOA. Unlike FPGAs, where all CLBs are identical, the processing elements in an FPOA perform different logical functions. The placement must not only conform to the routing delay, but must also assign the operations to the correct processing element. The placement problem can be interpreted as a one-on-one assignment of a finite set of operations in a DFG to a finite set of processing elements, where the assigned processing element is capable of performing the operation assigned to it. Similarly, the routing problem translates into a search for a set of n switch blocks, where n is the path delay, such that the n switch blocks establish a route between source and sink processing elements. The distance between any two switch blocks is further restricted by the maximum distance necessary for executing the design at a predetermined clock rate.

Furthermore, the type of decisions required for placing and routing a design on an FPOA is different than for FPGAs. For example, a Xilinx Virtex II Pro XC2VP100 FPGA chip has 11280 CLBS (= 44096 slices = 88192 LUTs and 88192 FFs) and approximately 11067 switch boxes [5]. Even if the problem is simplified to a one-on-one assignment of operations to CLBs, there are 11280! ways in which the design can be placed. Considering a channel width $W = 4$ bits, 398412 transistors must be configured during the routing phase. But, for an FPOA, a placement tool deals with 400 coarse-grained objects, and a total of 4000 multiplexers, 2000 launch/land registers, and 1600 nearest neighbor registers must be configured to route a design. The FPOA's heterogeneity further reduces the number of assignment permutations during place and route phase. However, the place and route decisions for FPOAs are driven by design-dictated communication delay. Thus, even though the cardinality of the FPOA resource set is typically lower than an FPGA resource set, the

resource allocation in CGRAs must adhere to the design's timing schedule, requiring a time-aware approach that is not offered by traditional P&R. The temporal nature of FPOA P&R warrants a method that explicitly deals with time.

The apparent differences between P&R for FPOA vs. ASIC/FPGA necessitates a fresh perspective towards FPOA placement and routing. New objective functions are needed to drive the search for finding appropriate solutions. This research explores and identifies such search objectives, and proposes a finite domain constraint satisfaction-based placement and routing approach for FPOAs. While the FPOA placement and routing problems are conceptually different than operation scheduling, they both can be formulated as problems, similar to scheduling.

Finite domain constraints have been applied to solve scheduling problems but their application to solving a P&R problem remains an unexplored area. Following is the theme of the work presented in this dissertation:

It is possible to develop a finite domain constraint satisfaction methodology to schedule, place, and route a design on an FPOA that minimizes schedule length, communication delay, area, and routing resources.

1.2 Research Contributions

The FPOA belongs to the category of 2-dimensional (2D) mesh CGRAs and its intricate details are described later in Chapters 2 and 3. This dissertation discusses the development of a P&R tool for the FPOA. The proposed FPOA tool flow is shown in fig. 1.4. During the initial phase, a design specification in the form of a DFG is translated into a Timed Data Flow Graph (TDFG) through a resource allocation and operation scheduling procedure. A high-level architecture specification of an FPOA is used as the target platform for placing the design represented by this TDFG. Next, the design is placed and routed using finite domain constraint-based placement and routing algorithms. It is possible that a placed design is not routable, in which case the placement step is repeated. If new placements are also not routable, reallocation and rescheduling of the input DFG generates a new RCG for place and route.

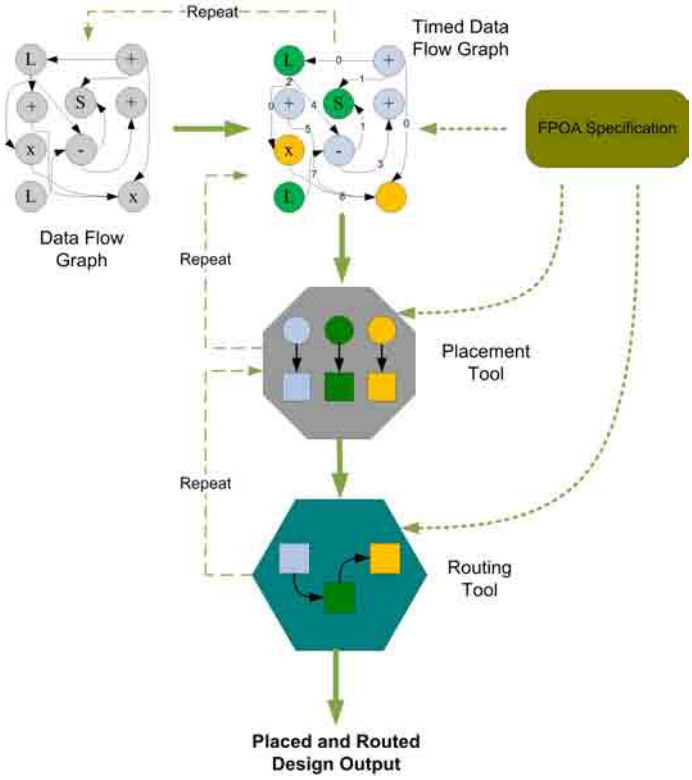


Fig. 1.4: Proposed FPOA tool flow.

The contributions of this research include:

- A simultaneous resource allocation and operation scheduling of a DFG using finite domain constraint satisfaction;
- Development of a constraint satisfaction methodology for placement of design on an FPOA;
- Development of a routing algorithm for FPOA using finite domain constraints;
- Evaluation of placement and routing methodology using scientific applications, multimedia, and signal processing benchmarks.

1.3 Overview of This Document

This dissertation discusses the development of multiple design tools to facilitate the implementation of a design on an FPOA. Chapter 2 outlines various coarse-grained recon-

figurable mesh architectures reported in surveyed literature along with a brief overview of search techniques. It also includes a list of various placement and routing techniques developed for ASICs, FPGAs, and CGRA design flows. Chapter 3 reviews the fundamental concepts used in this research, such as FPOA architecture, finite domain constraint satisfaction, and the Oz/Mozart tool. The methodology for allocating resources and scheduling a DFG to generate an RCG is described in Chapter 4. Chapter 5 delves into the description of a finite domain constraint solution for FPOA placement, elaborating on a mathematical model from which the placement solution is derived. The constraint satisfaction-based routing methodology is discussed in Chapter 6. Chapter 7 presents an overview of the test cases used for evaluating and demonstrating this research effort. The same chapter also presents the results for all the test cases after allocation, scheduling, placement, and routing phases. Chapter 8 concludes the dissertation and discusses future directions of research as related to this topic.

Chapter 2

Related Work

Place and route steps are vital for VLSI design. Existing literature reports a plethora of placement and routing techniques that have been developed for different architectures. In the case of ASIC design, the P&R step decides the final architecture, but the scenario is different for FPGAs and CGRAS where the architecture is fixed. In both of these cases, a search technique forms the core of the P&R algorithm. This chapter surveys several mesh-based CGRAs, search techniques, and provides an overview of the various techniques that have been reported for P&R of applications targeting ASICs, FPGAs, and CGRAs.

2.1 2D Mesh Coarse-Grained Reconfigurable Architectures

2D mesh architectures are characterized by a Manhattan arrangement of Processing Elements (PEs) that communicate using horizontal and vertical connections. Typically, two types of interconnect mechanisms are used: nearest-neighbor links and long-distance links. The nearest-neighbor links allow a PE to communicate with eight adjacent PEs corresponding to the eight directions. To connect distant PEs, interconnect segments of various lengths are used with varying interconnect data paths ranging from 1 bit to larger widths depending on the architecture. Various mesh-style coarse-grained architectures are discussed below.

Data Path-FPGA

Cherepacha and Lewis introduced an architecture similar to an FPGA but with coarse-grained features [6]. This new architecture, called Data Path FPGA (DP-FPGA), combines the flexibility of fine-grained programmability with the advantages of data path regularity, and is intended for use in data path intensive designs, such as digital signal processing,

communications, circuit emulation, and special-purpose processor applications. A design is implemented on a DP-FPGA using Chortle [7] and Hill's algorithm [8].

Kress Array

Introduced by Hartenstein and Kress [9], the reconfigurable Data Path Architecture (rDPA), currently known as the KressArray architecture, is a 2D grid of identical reconfigurable Data Path Units (DPUs), that are connected using a mesh of interconnects. Each DPU is composed of an ALU to perform standard arithmetical and logical operations, and a microprogrammable control unit for more complex operations such as division.

D-Fabrix

The D-Fabrix architecture is an array of 4-bit ALUs, multiplexers, registers, and memory, which are connected through a mesh network configurable by using routing switches [10]. The hierarchical architecture consists of tiles that are composed of two demi-tiles, each containing an ALU, three registers, two multiplexers, and one switchbox. One of the salient features of D-Fabrix is that data is kept local to the processing elements to minimize data transfers, thus reducing power consumption.

Colt

Unlike FPGAs, Colt architecture [11,12] is designed to operate on 16-bit words. Primarily targeted towards signal processing applications, Colt supports run-time reconfiguration and uses wormhole routing for communication between functional units. The architecture consists of 16-bit Interconnect Functional Units (IFUs), Data Ports (DP), crossbars, and a multiplier.

MATRIX

MATRIX is an 8-bit architecture and is comprised of identical Basic Functional Units (BFUs) which are connected using a hierarchical interconnect network [13]. MATRIX has a

three-level hierarchical interconnect network - the lowest level with zero clock cycle communication delay, the second level with length of four bypass interconnect, and the top-most level with global lines for long connections.

Garp

Garp [14] is a hybrid architecture that combines the merits of a conventional microprocessor with a reconfigurable computation unit. The main processor in Garp is built around the MIPS-II instruction-set with additional instructions to configure and execute the reconfigurable array.

FIPSOC

A FIPSOC chip [15,16] consists of an 8051 microcontroller core, an FPGA, configurable analog circuit block, on-chip memory, configuration memory, and I/O subsystem. User applications execute on the microcontroller, which interfaces with the FPGA and the analog circuit block to enhance its computing power.

RAW

Reconfigurable Architecture Workstation (RAW) shifts away from trends commonly applied in superscalar processor design [17–19]. The RAW architecture simplifies the hardware by moving many of the tasks to development software. The architecture consists of identical processing elements called tiles, connected as a homogeneous array. Data travels a short distance, from one tile to another, instead of using long communication network, resulting in higher clock-speed and increased scalability of the design.

REMARC

Reconfigurable Multimedia Array Coprocessor (REMARC) [20] consists of 64 processing elements called nanoprocessors arranged in an 88 matrix. The nanoprocessors communicate with their immediate neighbors using dedicated nearest neighbor connectivity and with those in the same row or column using 32-bit horizontal or vertical buses. REMARC

is tightly coupled to a MIPS RISC processor, whose ISA has been extended to configure and execute instructions on REMARC.

IRAM Architecture

The Intelligent RAM or IRAM architecture takes advantage of the low-latency and high bandwidth available on the memory chip [21,22]. The architecture consists of memory chips that act as regular DRAM but includes an array of processors and a floating point unit, making them different from conventional RAM architectures. By having sufficiently large memory, enough data can be loaded onto the memory to perform in-situ computation, decreasing the off-chip data traffic.

CHESS Array

Hewlett Packard laboratories proposed a reconfigurable arithmetic array, called CHESS, consisting of 4-bit ALUs, switchboxes, interconnection buses, and embedded block RAMs [23]. The routing structure consists of 16 segmented 4-bit buses in each row and column of the reconfigurable array. The routing occupies 50% of the array area, which is considerably less than most FPGAs.

MorphoSys

The Morphoing System (MorphoSys) is a reconfigurable processor array which includes an array of 16-bit reconfigurable cells (RCs) that is controlled by a 32-bit TinyRISC processor [24–27]. The RC array consists of 64 identical RCs or processing elements that are arranged symmetrically in an 8×8 matrix. This matrix is further split into four 4×4 quadrants. A hierarchical three-level interconnection network allows the RCs to access data from other RCs. The architecture also incorporates a context memory to save the configuration data, a Frame Buffer (FB), and a direct memory access (DMA) controller.

DReAM Array

Dynamically reconfigurable architecture for mobile systems (DReAM) is comprised of a

2D mesh of 16-bit processing elements called the reconfigurable processing unit (RPU) [28]. The RPU is responsible for all application specific data-flow and control-flow operations. It consists of two 8-bit reconfigurable arithmetic processing (RAP) units, one Spreading Data Path (SDP), one RPU-controller, two dual port RAMs, and a communication protocol controller. RPUs communicate with adjacent RPUs through a 16-bit fast local interconnect and use 16-bit global buses for long distance communication.

MONARCH

MOrphable Networked micro-ARChitecture (MONARCH) is a heterogeneous parallel processor which combines six 32-bit 5-stage pipeline RISC processors with a Field Programmable Compute Array (FPCA) [29, 30]. A high-bandwidth dynamic switched interconnect aids in data transfer among neighboring FPCA elements. The FPCA design is optimized for streaming data and signal processing application, and can be configured to implement a 256-bit wide SIMD engine.

ADRES

Mei et al. [31, 32] have proposed an architecture called Architecture for Dynamically Reconfigurable Embedded System (ADRES). The ADRES architecture has two components: a VLIW processor and a coarse-grained reconfigurable matrix, which are coupled as a processor and a co-processor. The VLIW processor exploits Instruction Level Parallelism (ILP) while the reconfigurable matrix improves the performance by exploiting the parallelism.

Field Programmable Object Array

Mathstars Field Programmable Object Array (FPOA^{TM}) is a coarse-grained heterogeneous reconfigurable computing platform consisting of coarse-grained silicon objects which communicate though a configurable communication network. The Arrix family [33] is the current generation of FPOA and consists of 400+ silicon objects. Chapter 3 describes FPOA architecture in detail.

ClearSpeed CSX Family

ClearSpeeds CSX family of processors is built on top of a multi-threaded array processor (MTAP) core to support parallel data processing [34, 35]. The architecture of an MTAP processor has a highly parallel execution unit instead of single ALU, register file, and I/O device configuration. The execution unit contains a mono execution unit for processing scalar data while the poly execution unit contains a large number of poly execution (PE) cores. Each PE core has an ALU, a floating point unit, a multiple-accumulate unit, a register file, memory, status and enable registers, I/O channels, and paths for inter-PE communication.

Coarse-Grained Hybrid Reconfigurable Architecture

Verma and Akoglu have proposed a Coarse-Grained and Hybrid Reconfigurable Architecture (CGHRA) [36] targeted for applications like variable block size motion estimation used in H.264 video compression standard [37,38]. The architecture contains 16 configurable processing elements (CPEs), four processing elements of type-2 (PE2s), two processing elements of type-3 (PE3s), and a memory interface (MI). A network-on-chip (NoC) provides the communication backbone to these PEs.

Element CXI Elemental Computing Array

Element CXI is an Elemental Computing Array (ECA), consisting of non-homogeneous, pipelined computational engines called elements [39]. Each element has four 16-bit inputs, two 16-bit outputs, a controller called Judge, and an associated context. Due to its hierarchical interconnect and run-time task binding, Element CXI does not fall under the target architecture domain of this dissertation.

TILE™ Multicore Processor Architecture

Tilera® Corporation has developed a family of tile-based multicore processor architectures which include TILE64, TILEPro36, TILEPro64, and most recently the TILEGx [40–42]. These processors feature a multicore architecture ranging from 36 identical

tiles in TILEPro36 to 100 tiles in TILE-Gx. All the tiles communicate through Tileras iMesh^{TM}on-chip network [43]. A multicore development environment facilitates tile processor programming using a C/C++ compiler.

SmartCell

SmartCell is a coarse-grained reconfigurable architecture targeting stream-based applications [44]. The architecture consists of processing elements operating on 16-bit inputs to generate 36-bit outputs, and includes input registers, an arithmetic and logic unit, instruction memories, instruction controllers, and multiplexers. The prototype contains 16 cells which are tiled in a 2D mesh structure and communicate through a three-level layered interconnect network.

Summary of 2D Mesh CGRAs

Table 2.1 summarizes the architectures described earlier in this section. Architecure and granularity columns illustrate the name and the granularity of an architecture. Granularity is defined as the data path width of an architecture and multiple entries under granularity indicate a multi-granular architecture. The third column displays the nature of the routing network: static and/or dynamic, channel-width specifies the number of parallel wires in the interconnect and can differ from an architectures granularity, and processing element denotes the type of fundamental processing unit(s) used in the architectures. A Y in column six indicates if any design tools for an architecture have been reported, while an N denotes its absence. An asterisk (*) indicates that the tool is a proprietary too. Finally, the last column displays the year in which an architecture was reported, and Table 2.2 lists any design tools reported for architectures mentioned in Table 2.1.

2.2 Search Techniques

Several search techniques have been reported in the literature which include integer linear programming, evolutionary algorithms, simulated annealing, and constraint satisfaction.

Table 2.1: Summary of coarse-grained reconfigurable architectures. A * indicates proprietary tools.

Architecture	**Granularity**	**Routing Network**	**Channel Width**	**Processing Element**	**Design Tools**	**Year**
DP-FPGA	4-bit	Static	4-bit	LUT	Y	1994
Kress Array	32-bit	Static Dynamic	32-bit	rDPU/ALU	Y	1995
D-Fabrix	4-bit	Static Dynamic	4-bit	ALU	Y*	1995
Colt	16-bit	Dynamic	16-bit	IFU/ALU	N	1996
MATRIX	8-bit	Static Dynamic	8-bit	BFU/ALU Control Logic	N	1996
Garp	2-bit	Static Dynamic	2-bit	LUT Clusters	Y	1997
FIPSOC	4-bit 8-bit 9-bit	Static	4-bit	Microprocessor Digital Cells Analog Cells	Y	1997
RAW	32-bit	Static	32-bit	RISC processor	Y	1997
REMARC	16-bit	Static	16-bit 32-bit	Nanoprocessors	Y	1998
IRAM	8-bit	Static	N/A	ALU RISC processor	N	1998
CHESS	4-bit	Static	4-bit	ALU	N	1999
MorphoSYs	16-bit	Static	16-bit	Multiplier-ALU	Y	1999
DReAM	8-bit	Static	16-bit	RPU	N	2000
MONARCH	32-bit	Dynamic	32-bit	RISC processor Multiplier-ALU	Y*	2002
ADRES	32-bit	Static	32-bit	ALU	Y	2003
FPOA	16-bit	Static Dynamic	21-bit	ALU, MAC	Y*	2003
ClearSpeed	64-bit	Static	64-bit	ALU	Y*	2006
CGHRA	8-bit	Dynamic	32-bit	Subtractors Adders	N	2008
SmartCell	16-bit	Static Dynamic	16-bit 36-bit	ALU	Y	2009

Dantzig developed a technique called the Simplex method [45, 46] to solve linear programs. The concept of linear programming has been extended to Mixed Integer Linear Programming (MILP) [47], which formulates an optimization problem as the minimization or maximization of a cost function, subject to a set of constraints. MILP suffers from limited scalability and has limited expressiveness.

Table 2.2: Design tools for CGRAs.

Architecture	**Design Tools Reported in Literature**
DP-FPGA	Uses Chortle and Hill's algorithm
Kress Array	Data path synthesis system based on Simulated Annealing
D-Fabrix	Proprietary tools
Garp	Garp Configurator, C compiler-based design flow, Garp simulator
FIPSOC	Proprietary tools
RAW	RAW compiler
REMARC	REMARC configuration Environment
MorphoSYs	Morphosim VHDL simulator
MONARCH	Proprietary tools
ADRES	VHDL synthesis and simulation
FPOA	Proprietary tools
ClearSpeed	Proprietary tools
SmartCell	Smart_C prototype compiler

Evolutionary algorithms and simulated annealing are combinatorial search techniques and have been used in embedded system design, electronic system level design tools, and ASIC/FPGA placement and routing. Evolutionary algorithms or genetic algorithms model the search space as a population of potential solutions or chromosomes. New generations are created by combining parent chromosomes or by randomly changing the genes within the chromosome. A fitness function evaluates the quality of the population after creating a new generation and discards inferior offsprings. The population grows and improves with each iteration of the process and terminates after a pre-defined number of generations have been created. An evolutionary search problem can be divided into two sub-components: the problem definition and the search algorithm. PISA [48] is an interface specification that allows a problem definition and an evolutionary multi-objective search algorithm to be implemented as separate communicating processes. The PISA framework establishes a formal model that dictates the control flow and data exchange between these two processes. All the communication takes place through text-files, which further allows the two processes to be located on different machines running different operating systems. However, both the processes are required to be PISA compliant. Even though the goal of PISA is multi-objective optimization, PISA is not a search tool but instead, it is an interface specification that enables the user to combine a search problem with an evolutionary search technique

to address an optimization problem.

Simulated annealing [49], on the other hand, is a concept that has been borrowed from metallurgy. The problem is modeled as a random and possibly invalid solution state. Random perturbations produce new states, where the frequency and the degree of perturbation is controlled by a parameter called temperature such that the number of random changes decrease as the temperature cools. A new state is accepted if it is better than the existing state, as evaluated using a cost function. The algorithm terminates once the temperature falls below a threshold. For both the above search techniques, the random changes avoid being stuck in local minima. However, neither guarantees a good solution nor ensures full coverage of search space.

Constraint Logic Programming (CLP) is a combination of constraint satisfaction and logic programming [50]. A CLP problem involves a constraint domain, variables, and set of constraints defined over the variables. The constraint domain defines the domain of computation and the set of permissible operations on the values represented by the domain. A typical program for CLP specifies a domain, a set of constraints, and a goal. The constraint solver attempts to prove the goal while satisfying the constraints by assigning values to the variables, where the values lie within the constraint domain. The simplest form of a constraint consisting of a single operation and corresponding number of arguments is called a basic constraint. Complex constraints can be expressed as a conjunction of multiple basic constraints.

Finite Domain (FD) constraints are a field of CLP such that the constraint domain is restricted to a finite set of values. The constraint domain is restricted to non-negative integers and permits integral as well as Boolean operations. FD constraints facilitate the representation of, and solution for, discrete search problems [51]. FD constraint programming has been applied to high-level synthesis [52], static scheduling of real-time systems [53], and design space exploration [54].

2.3 Placement and Routing Techniques

Several placement and routing techniques have been proposed in the literature. These

techniques have been implemented using various search algorithms [55] and target different architectures. Even for the same architecture, the objectives of two P&R methods can be different. The following sections provide an overview of the place and route methodologies used in design flow of ASICs, FPGAs, and CGRAs.

2.3.1 P&R for ASICs

P&R is an essential step in the design of an integrated circuit and consists of two phases: Placement and Routing. During placement, the VLSI cells are laid out on the chip layout, such that the cells do not overlap and can be connected through wires as per the design netlist. Once the design is placed, routing phase establishes the actual interconnects among cells. Routing is dependent on placement and a bad placement can result in a non-routable design. The quality of placement is measured in terms of the cost of placement, which can be the total wirelength, number of cuts, congestion, timing, etc. This cost is calculated using cost functions. The goal of placement is to place the cells while minimizing one or more cost functions.

Minimizing wirelength reduces the congestion and signal delay, while also decreasing the chip area required for routing the wires. Almost all placement tools use some form of wirelength minimization criterion. For minimizing the wirelength during placement, the tool must ideally know the exact routing of wires. Since the placement phase precedes routing, actual routing information is not available. Instead, techniques are used for estimating wirelength for guiding the placement. Typically, a cost function is defined based on some design criteria: such as minimal total-wirelength, penalty for wires longer than a given threshold, etc.

In a 2D mesh network, the shortest interconnect among various nodes is a Steiner tree [55] connection. Measurement and minimization of the rectilinear Steiner tree wirelength is a good objective for placement, but unfortunately it is computationally expensive. During placement, the cost function needs to be evaluated often, because an expensive function results in a slow and unacceptable placement procedure. Since an exact wirelength is not required during placement, alternative techniques for estimating the wirelength may be

used. Instead of Steiner tree, other wiring schemes such as minimal spanning tree, chain connections [55], and multi-source multi-sink [56] connections are used. Bounding box or Half Perimeter Wire Length (HPWL) estimation is a popular method for wirelength approximation [57]. For lower degree nets, HPWL gives the exact minimal rectilinear Steiner wirelength.

FLUTE [58] is another approach that uses lookup tables for estimating wirelength for networks with small number of pins. The lookup table consists of possible optimal wirelength vectors (POWVs), where a POWV represents a potential optimal route for a net along Hanan's grid [4]. Possible routing topologies are generated by removing redundant or non-optimal routes from the set to obtain a POWV for each net. For larger degree nets, a divide and conquer approach is used and the nets are subdivided into several nets of smaller degrees.

Dragon2000 [59] is a 2-phase hierarchical iterative placement tool developed for large circuits. In the first/global phase, the area is divided into four bins and the cells are distributed to each bin, followed by subsequent subdivisions. Cell overlap is permitted during this phase with the goal of minimizing both wirelength and min-cut. A greedy algorithm is used for legalizing the placement and reducing wirelength during the second/detailed placement.

Roy and Markov propose that Steiner Tree Wire Length (StWL) correlates better with the routed wirelength than the commonly used half perimeter wirelength [60]. However, building Steiner trees and estimating StWL for each net is computationally expensive and becomes a bottleneck when used in an iterative placement approach. Fast Steiner evaluators such as FastSteiner [61] and FLUTE exist but may still not have reasonable runtimes for large problems. The authors have developed a placement algorithm called ROOSTER [60] based on net weighted StWL estimation. In order to evaluate StWL in a reasonable time, the authors propose the use of an efficient data structure which reduces the execution time for Steiner evaluators.

Min-cut is a partitioning-based placement method where a design is subdivided into

smaller partitions, with the goal that the number of interconnects between partitions is minimized. It is a top-down approach that initially emphasizes global placement. Breuer [62] proposed the first min-cut placement algorithm for physical implementation of electrical circuits: Quadrature placement procedure and Slice/bi-section placement procedure. In the former algorithm, the cut lines alternate between vertical and horizontal. In the latter case, the area is iteratively divided into horizontal slices and elements are assigned to the bottom row at each iteration. Finally, the cells are assigned to the columns using vertical bi-section. Lauther [63] suggested cell rotation, squeezing, and reflection to reduce whitespace and improve min-cut placement.

NTUplace [64] is a hierarchical, partitioning-based placement algorithm which uses weighted-nets, ratio partitioning, and look-ahead bi-partitioning to place mixed size cells. The partitioning problem is formulated as a hypergraph partitioning problem with weights assigned to the edges in the hypergraph. The chip area is divided into regions, with blocks assigned to a region, such that the HPWL of the nets is minimized. To determine appropriate cut-size for the regions, higher weights are assigned to larger cells and the resulting weight determines the cut-size.

Capo [65] is an academic min-cut hypergraph partitioning-based placement tool which performs recursive bi-section placement. Capo generates wirelengths comparable to commercial placers and the placed design is usually routable. Kahng and Reda [66] have proposed an iterative placement feedback technique for improving terminal propagation during placement. The idea is to perform a placement step, undo it, and feed the results back into the current step to drive the terminal propagation. The authors have implemented their technique in Capo and have reported HPWL reductions on standard benchmarks.

Feng Shui [67,68] is another partitioning-based placement tool which aims at optimizing the placement from a global perspective by minimizing the wirelength. The tool performs recursive bi-section, but unlike traditional approaches, it performs multi-level partitioning in which multiple regions are bisected simultaneously. Recent work on dynamic programming for cut sequences [69] and fractional cuts [70] has been reported for improving Feng Shui's

performance.

TimberWolf [71] is a simulated annealing-based [72] set of placement and routing tools for VLSI designs. TimberWolf consists of programs that can place standard cells, custom cells, and gate arrays. It also has a router program for global routing of standard cells. The simulated annealing-based placement algorithm estimates the total wirelength and also uses a penalty function to compute overlap penalties. The estimated wirelength calculates the HPWL for various nets to determine the total wirelength. The penalty function is employed to minimize overlap that arises from interchange or displacement of cells.

The router program in Timberwolf package operates in two stages. In the first stage, the program finds the net segments to minimize interconnection distance. In the second stage, a simulated annealing algorithm determines the routing.

Fast Place [2,73,74] is a placement tool based on the quadratic placement approach [75]. It uses a hybrid net model for capturing the placement problem and to speed-up the quadratic solver. A 3-stage algorithm first performs a coarse global placement by spreading the cells in the placement region while minimizing the wirelength. After the coarse placement, a local refinement technique is employed for another global optimization and cell shifting step. Since the primary objective is wirelength minimization, a higher weight is attached to the wirelength component than the bin utilization. In the final stage, a detailed placement step legalizes the placement from the previous stage by removing overlap among cells.

FastRoute [76] is an attempt towards developing a fast router which can estimate interconnect congestion and delay during placement with an acceptable computational burden. Most traditional routers have runtimes which restrict their usage in iterative placement algorithms in which the router is executed frequently for estimating the interconnect. Pan and Chu proposed the FastRoute router which generates a congestion map, constructs a corresponding Steiner tree, and employs pattern and maze routing. Further improvements have been made in the recent versions of FastRoute, such as replacement of pattern routing with a monotonic routing scheme, Multi-Source Multi-Sink Maze (MSMSM) routing [56]

and via minimization [77]. In an effort to use the same technique for routing and congestion estimation, the authors propose IPR tool [78, 79] which integrates routing during the placement process.

Gao et al. borrowed various concepts from FastRoute and PathFinder [80] to develop an iterative router called NTHU-Route [3,81]. Initially, a Steiner tree topology is generated using FLUTE and edge shifting. Once the congested areas are identified, the edge cost is computed by a history-based cost function. A monotonic routing approach with rip-up and re-route of congested edges approach is employed to allocate routes to less congested edges. An adaptive multi-source multi-sink maze routing approach is applied to route the remaining unrouted nets.

Placement and routing methods used in ASIC design are summarized in Table 2.3. The name of the method or place and route tool is listed under Methods/Tool column. The next column displays the name of algorithm(s) used for place and/or route. Evaluation metric column shows the parameter used for measuring and optimizing the quality of place and route. The year in which the method was first reported is mentioned under the Year column. Finally, the validation or performance evaluation method is listed in the last column. As can be observed, most researchers have compared the performance of their tool with other available tools using benchmark suites such as ISPD [82], MCNC [83], or Digital Signal Processing (DSP) algorithms.

2.3.2 P&R for FPGAs

Rent's rule formulates the relationship between the number of external interconnects of a logic block and the number of logic gates inside the logic block. Donath [84] proposed that Rent's rule can be used for estimating wirelength in a logic design prior to placement and routing. Rent's rule has been used for estimating the interconnect wirelength [85] and congestion estimation [86]. MVPR [87] is a simulated annealing-based tool for FPGAs that uses Rent's exponent in its cost function to drive the placement in order to minimize the design area.

Cost effective routing algorithm (CeRA) [88] is a routing approach based on exact

Table 2.3: Placement and routing methods for ASICs.

Method/ Tool	**Algorithm**	**Evaluation Metric**	**Year**	**Validation/Evaluation method**
Breuer's method	Quadrature placement, bisection	Interconnect-minimization	1977	Results compared with manual placement
TimberWolf	Simulated annealing	Wirelength	1985	Compared to other tools using benchmarks
Dragon2000	Min-cut, iterative	Wirelength	2000	Compared to other tools using benchmarks
Capo	Recusrsive bisection	Wirelength	2000	Evaluated using a set of benchmarks
FengShui	Recusrive bisection	Wirelength	2001	Compared to other tools using benchmarks
FastPlace	Quadratic placement	Wirelength Bin utilization	2005	Compared to other tools using benchmarks
NTUPlace	Hypergraph partitioning	Wirelength	2005	Evaluated using ISPD benchmarks
FastRoute	Monotonic & MSMSM routing	Congestion	2006	Compared to other tools using benchmarks
IPR	FastPlace, FastRoute	Wirelength Congestion	2007	Compared to other tools using benchmarks
ROOSTER	Min-cut placement	Wirelength	2007	Compared to other tools using benchmarks
NTHURoute	Monotonic & MSMSM routing, rip-up reroute	Wirelength, Via mini-mization	2008	Compared to other tools using benchmarks
MaizeRouter	Edge shifting, Maze routing	Wirelength	2008	Compared to other tools using benchmarks

calculation of routing density in symmetrical FPGAs. CeRA not only considers the routing density at connection blocks, but also takes into account the track utilization at switch blocks. The algorithm consists of a global routing phase and a detailed routing phase, both of which depend on the exact routing density of nets.

The criteria for a good routability estimator are speed and accuracy, FPGA independence, and ability to be incorporated in most applications requiring estimation. Typically, the routing demand for each routing resource is computed and considered by the placement or routing tools to generate solutions that avoid using the resources that are in high

demand, yielding routable solutions. Kannan et al. [89] compare the following routability estimation methods that are used in FPGA placement tools: fGREP [90], RISA [91], Lou's method [92]. fGREP takes into account the demand imposed by each terminal of a net on the routing resources in the net's bounding box. The runtime of fGREP is proportional to the set of routing resources in a bounding box and the number of terminals, which can be large with increases in circuit size. RISA takes an empirical approach and generates wiring distribution maps (WDMs) for a large set of randomly generated nets to obtain a mean net-weight for different pin counts. This net weight is used for computing the resource demand in the net bounding box, where a net bounding box is the smallest rectangular region that spans all the points in the net. Lou et al. [92] use a probabilistic model for computing the congestion. In Lou's method, after placing the net lists, the chip area is divided into a grid of regions. The demand for a routing resource within each region is calculated as the ratio of number of routes using the particular resource and the total number of routes possible. This ratio is the probability of usage of the resource.

Jariwala and Lillis [93] have experimented with the reliability of routing congestion prediction for driving placement. Their experiments reveal that the estimated congestion is not accurate, but may be helpful if it reliably identifies the areas of congestion. To improve routability during detailed placement, the authors propose an exact approach for estimating congestion called Routing-Based Interleaving (RBI). RBI attempts to reduce maximum routing density. When applied to island style FPGAs, it takes into account number of channels at maximum density and the wirelength.

SEGment Allocator (SEGA) [94] is a routing tool for FPGA architectures featuring a mix of short and long interconnect channels. Initially, all the nets are globally routed to generate a coarse route, which effectively is a sequence of point-to-point connections joined together to form a path from the source to the sink. In order to establish a physical path, the length of a connection is matched to the length of an interconnect channel to generate a set of possible physical paths for the connection. SEGA completes the detailed routing by selecting one physical path from each set using a cost function that discourages the use of

long channels for short connections, minimizes the number of segments used, and prioritizes critical routes.

PROXI [95] is a timing-driven placement and routing approach for island style FPGAs. It uses a simulated annealing algorithm to simultaneously place and route designs. A placement perturbation is followed by a rip-up of the affected routes, which are re-routed incrementally, avoiding an otherwise heavy penalty for a full re-route. The worst case delay of critical paths is also included in the cost function to bias the selection of shorter paths. PROXI gives good results in terms of routing density and performance but is computationally intensive, making it infeasible for large designs.

Lee and Wu [96] proposed TRACER-fpga_PR router for RAM-based FPGA's. TRACER-fpga_PR attempts to minimize the routing channel density as well as overall path delay. Nets are routed depending on their criticality to obtain an initial routing which is then ripped-up and re-routed to resolve any resource conflicts. To remove timing violations, a simulated evolutionary algorithm performs further rip-up and re-route iterations to ensure that the timing constraints are met, otherwise the tool reports failure.

FPGA placement and routing (FPR) [97, 98] tool uses a divide and conquer approach by subdividing the FPGA into $m \times n$ regions. The logic blocks in each net are assigned to a region while minimizing the wirelength and congestion. Using simulated annealing, the logic blocks are moved between regions to refine the placement. Global routing for the design is performed by constructing Rectilinear Steiner Arborescences (RSAs) [99, 100] for each net. A greedy heuristic then reduces the congestion and recursively assigns the RSA edges to switch blocks until at most one logic block remains in each region. Finally, a Steiner tree graph is constructed for capturing the routing structure of the FPGA, and detailed routing is performed using a greedy iterative heuristic.

Pathfinder [80] is an iterative routing algorithm for FPGAs and attempts to balance the congestion delay trade-off. A timing-critical net must use a minimum delay path even if it leads to an increase in congestion, whereas non-timing-critical nets can be routed through alternate uncongested paths. Unlike obstacle avoidance routers which do not allow

any resource overuse to avoid congestion, Pathfinder initially routes nets even if it leads to congested or overused illegal routes. Competing nets negotiate for a shared path which is assigned to the most timing-critical net. To eliminate non-timing-critical nets, the congestion cost of a path is gradually increased depending on the demand and the congestion history of the path. As the cost of a path increases, candidate nets are forced to look for less congested and possibly longer paths. Eventually, the path is assigned to the most critical net amongst the competing set.

Versatile Place and Route (VPR) [1,101] is the state-of-the-art in placement and routing tools for FPGA. It can be used for island-style and row-based FPGAs. The primary components of VPR include VPACK, placement algorithm, and a routing algorithm. VPACK is a logic block packing algorithm which assigns LUTs to basic logic blocks (BLEs) and the resulting BLEs to logic clusters depending on user defined parameters. Once the logic clusters are determined, the next step is to place them on the FPGA fabric. VPR takes a simulated annealing approach for placement. The cost function employed penalizes those placements which place higher demand on lower capacity channel.

The routing algorithm used in VPR is based on Pathfinder [80]. Both Pathfinder and VPR's router use maze routing algorithms [102] to find nets. Maze routing performs a breadth first search to find the sinks. Once a sink is found, the search begins again by considering the original source and the newly found sink as new sources. Thus, a new breadth first search is initiated every time a sink has to be found. VPR's router avoids performing a search from scratch and resumes from where the last sink was found, thereby reducing the computation time.

Independence [103] is a routability-driven, architecture independent, FPGA placement tool. Independence uses simulated annealing to minimize wirelength and routing congestion. To speedup the placement, only an incremental rip-up and re-route is done after every simulated annealing move. Independence produces placement solutions for island style FPGAs which are within 5% of the state-of-the-art VPR tool [101].

FPGA placement and routing methods are summarized in Table 2.4. This table has

Table 2.4: Placement and routing methods for FPGAs.

Method/ Tool	**Algorithm**	**Evaluation Metric**	**Year**	**Validation/Evaluation method**
SEGA	Exhaustive search, segment length matching	Channel segment minimization	1993	Evaluated using MCNC benchmark
PROXI	Simulated annealing, Maze routing	Routability, delay	1995	Compared to other tools using benchmarks
FPR	Partitioning, Simulated annealing, Greedy heuristic	Wirelength, congestion	1995	Compared to other tools benchmarks
Pathfinder	Iterative congestion negotiation	Congestion	1995	Evaluated on existing FPGA architectures
TRACER-fpga_PR	Rip-up reroute	Channel density, Wirelength	1997	Compared to other tools using benchmarks
VPR	Simulated annealing, pathfinder	Congestion	1997	Compared to other tools using benchmarks
MVPR	Simulated annealing	Device area utilization	2001	Compared to other tools using benchmarks
CeRA	Iterative maze routing, net decomposition	Routing density	2004	Evaluated using MCNC benchmark
Indepen-dence	Simulated annealing	Wirelength congestion	2005	Adaptability to different architectures, Comparison with VPR
KPF	Simulated annealing	Wirelength	2009	Evaluated using MCNC benchmark

the same format as Table 2.3. Majority of validation approaches are based on performance comparison of the reported place and route tool with other existing tools using a benchmark suite.

2.3.3 P&R for CGRAs

Silva et al. [104] propose a genetic placement algorithm for data-driven coarse grained reconfigurable array architectures. The algorithm supports a variety of array architecture topologies. They model the array architecture as a chromosome by representing each array cell as an element of the chromosome. The placement is denoted by assigning a node of a DFG to an element. The fitness function favors shorter paths while penalizing longer paths.

Ferreira et al. extended the work presented by Silva et al. [104] and use graphs to represent the architecture topologies [105]. Their placement algorithm performs a simultaneous depth first traversal of both the DFG and the architecture graphs to find a placement. Using random depth first searches, a set of placement solutions is obtained. Each of these solutions is evaluated for its routing cost and the solution with the least routing cost is selected.

Hartenstein and Kress [9] proposed a simulated annealing algorithm for rDPA placement. The cost function considers the chip boundaries, routing resources, and penalizes routing via the rDPA bus. Lai et al. [106] developed a recursive placement algorithm for rDPA. The algorithm transforms a DFG into a path-structure graph which captures the paths in decreasing order of path lengths, starting from the longest path in the DFG. Each node in the path-structure graph denotes a unique path in the DFG. By placing a child in vertical (or horizontal) direction and its immediate parent in horizontal (or vertical) direction, the placement algorithm attempts to place connected operations in the DFG on adjacent DPUs. If adjacent placements are not possible, additional DPUs acting as pass-through connections are used to avoid overlap.

RAW processor's placement algorithm is based on VLSI cell-placement approach [107] and performs a one-on-one mapping of software threads on to a physical tile, while minimizing latency and bandwidth [17]. Routing is done using a greedy Topology Independent Pipelined Routing and Scheduling (TIERS) algorithm which determines the paths for inter-tile communication [108]. The wire delay is measured as number of network hops, where one network hop corresponds to one cycle wire delay [109]. The programmer and RAW compiler are exposed to the network delay through RAW ISA and can program the static router to ensure ordered data flow.

Fung et al. [110] have taken a genetic algorithm (GA)-based approach for placing designs on heterogeneous coarse grained reconfigurable arrays. A chromosome encodes a net as an individual and assigns a position value to each logic element in the net. The position value denotes the relative proximity of logic elements in the net. The fitness function consists of three parameters: total wirelength, number of routing switches needed,

and the criticality of the path. The GA-based placement engine performs tournament selection to select the best placement from a random pair of placements in the population. The process is repeated and the fitness is evaluated to find a good placement solution. Even though the algorithm is targeted towards CGRAs, the authors chose to compare the results with VPR using benchmarks for FPGAs. VPR outperforms the GA engine in almost all the test cases.

Methods reported for placing and routing CGRAs are summarized in Table 2.5. This table has the same format as Table 2.3. Researchers have validated/evaluated their tools by using DSP algorithms as benchmarks or by comparing the performance with another place and route tool such as VPR. Once again most of the validation approaches are based on performance comparison of the reported tool with other existing tools using a benchmark suite.

Table 2.5: Placement and routing methods for CGRAs.

Method/ Tool	**Algorithm**	**Evaluation Metric**	**Year**	**Validation/Evaluation method**
rDPA placement	Simulated annealing	Routing resources	1995	Not reported
RAW compiler	Simulated annealing, TIERS	Latency, bandwidth	1997	FPGA-based logic emulation
Fung's method	Genetic Algorithm	Wirelength, routing resources, path criticality	2006	Compared to VPR using MCNC benchmarks
Silva's method	Genetic Algorithm	Wirelength	2006	Evaluated using DSP benchmarks
Ferreira's algorithm	Depth first graph traversal	Interconnections	2007	Evaluated using DSP benchmarks

Chapter 3

Background

The primary objective of this research is to develop a finite domain constraint satisfaction-based scheduling, placement, and routing approach for FPOA architectures. Section 3.1 of this chapter discusses silicon objects and the communication framework comprising an FPOA architecture. An overview of finite domain constraint satisfaction methodology is presented in sec. 3.2.

3.1 FPOA Architecture

An FPOA is a field-programmable silicon device that offers a higher level of abstraction than FPGAs. Instead of using fine grained building blocks (gates), the FPOA employs coarse-grained processing elements called silicon objects. These objects interact with each other through a configurable communication network and perform high-level functions. The Arrix family of FPOA offers 400 such objects, arranged in a 20×20 grid, which includes 256 Arithmetic Logic Unit (ALU), 64 Multiply Accumulate (MAC), and 80 Register File (RF) objects as shown in fig. 3.1. ALU, MAC, RF, and the interconnect framework are described below.

3.1.1 ALU Object

The ALU object is one of the most complex programmable objects in the FPOA. It consists of a 16-bit Arithmetic Logic Block (ALB), a configurable instruction state machine, four general-purpose truth functions, and a truth function for the arithmetic logic block. The ALB supports general purpose arithmetic and logic operations on four inputs which include two 16-bit operands, a 16-bit mask, and a 1-bit carry input. Selection of actual inputs consumed by an instruction is decided by the instruction state machine. During

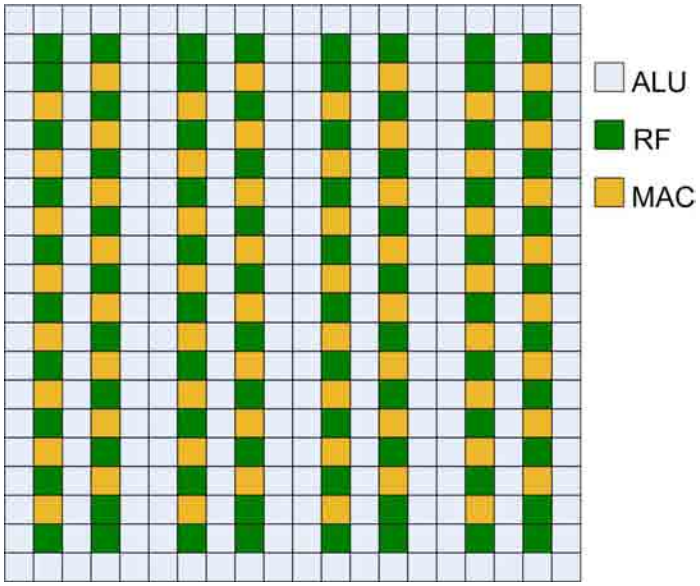


Fig. 3.1: FPOA arrix architecture.

initialization, the instruction state machine is always reset to state 0.

3.1.2 MAC Object

The FPOA provides MAC objects to perform multiply and accumulate functions. Internally, a MAC is comprised of a multiplier function and an accumulator function. The multiplier function operates on two 16-bit operands to generate a 32-bit product. The primary purpose of the accumulator function is to add a 32-bit input to the existing number within the accumulator block. The accumulator performs three basic functions: preload a number into the accumulator block, load the output of the multiplier into the accumulator block, or add multipliers output to the existing number in the accumulator block. The accumulator generates a 40-bit output which can be dynamically mapped onto two 16-bit output registers by selecting any two of [39:24], [33:18], [31:16], or [15:0] bit fields of the 40-bit output.

3.1.3 RF Object

An RF object is a storage element that supports simultaneous read and write on every clock cycle. Each RF object contains 64 20-bit memory location that can hold 16 bits of

data, plus four control bits. The register file can be configured to operate in single width or double width read write mode to provide 64 × 16-bit or 32 × 32-bit output, respectively. The RF object operates in either of these three modes: Random Access Memory (RAM), First In First Out (FIFO), and Read sequence. In RAM mode, the RF object resembles a dual port random access memory and supports simultaneous read/write to/from the same address, while the FIFO mode configures a register file into a fixed 64-word circular buffer. The read sequence mode combines the RAM and the FIFO modes to perform random address writes but sequential read from a circular buffer.

3.1.4 Interconnect Framework

All the objects on an FPOA communicate using a configurable mesh of interconnects. The interconnect framework provides two types of connections: Nearest Neighbor (NN) and Party Line (PL). Nearest Neighbor connection allows adjacent objects to transfer data without any latency while Party Line connection is used by non-adjacent objects. Figure 3.2 shows an object and its communication channels. A communication channel is comprised of 16 data bits, four control bits, and one valid bit. An object can have a maximum of eight NN channels and 10 PL channels.

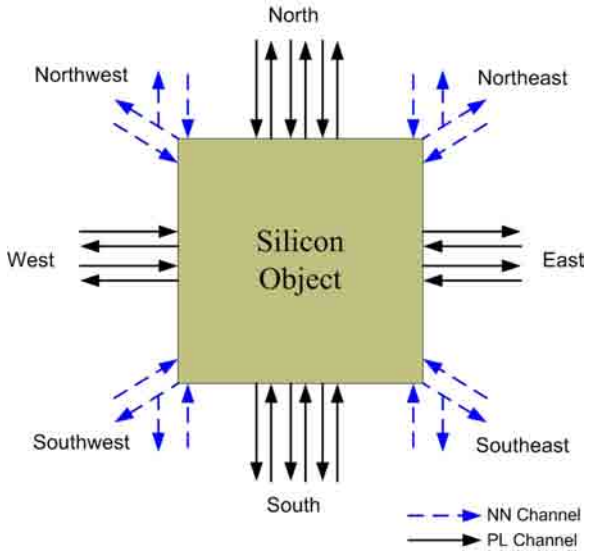
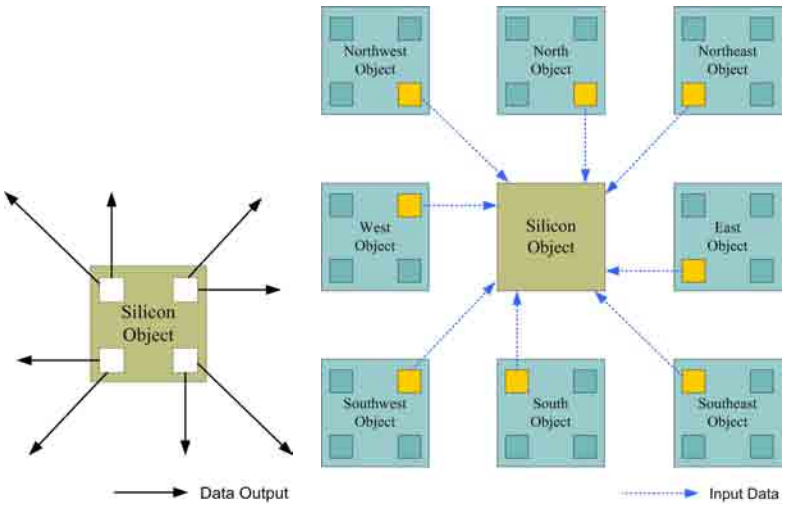


Fig. 3.2: Communication channels.

Nearest Neighbor Communication

Nearest Neighbor communication is required when the data produced must be consumed in the next clock cycle. Each silicon object has four special registers called NN registers. Figure 3.3(a) shows that each NN register can output data to two adjacent silicon objects. These registers are identified by their output direction: North/Northwest (NNW), East/Northeast (ENE), South/Southeast (SSE), West/Southwest(WSW). A silicon object can also read data from the eight NN registers of its adjacent objects as shown in fig. 3.3(b). Each input register is denoted by the orientation of its parent object: Northwest(NW), North(N), Northeast(NE), East(E), Southeast(SE), South(S), Southwest (SW), and West(W).

NN channels provide zero-latency communication but require a producer and a consumer to be physically adjacent. Except for periphery objects, all other objects have eight adjacent neighbors. It is observed in fig. 3.1 that no two MAC objects can be adjacent to each other, hence no two MAC objects can communicate using NN channels. Similarly, only eight RF objects are adjacent to each other. In contrast, a large number of ALU objects are adjacent to other ALU objects, supporting ALU to ALU zero-latency communication.



(a) (b)

Fig. 3.3: Nearest neighbor registers (a) Local NN registers for data output, and (b) Adjacent NN registers for data input.

Party Line Communication

PL connectivity offers data communication over longer distances but has a non-zero latency. PL channels are divided into three groups: Group 1, Group 2, and Group 3. Group 1 and 2 have four channels, one each in North, South, East, and West directions. Group 3 contains only two channels, one each in North and South directions.

Special registers called Launch/Land (LL) registers are used for party line communication. One LL register is shared by two channels moving in opposite directions within the same group. The directions of the channel pair are used for naming the shared LL register. For example, in Group 1, a North/South LL register is shared by the North and South channels. Each silicon object contains a total of five LL registers. LL registers are used for sending data from a source object, receiving data at the destination object, and for registering data at the end of each clock cycle during transmission.

When sending data, a silicon object places data in its local NN registers or in a LL register. Data is launched through a launch Multiplexer (Mux), and travels from one object to another through party line channels. Figure 3.4 shows how data is launched in the East direction from an East/West LL register. A change in direction is allowed as long as data remains within the same PL group. However, a U-turn is not allowed without first landing

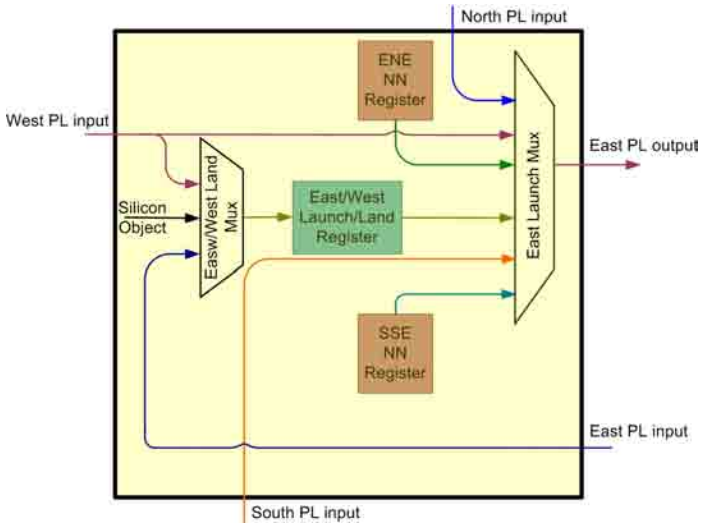


Fig. 3.4: Party line launch and land register.

on a an LL register. In order to switch groups, a silicon object must be programmed to move data between LL registers of two different groups. If the data has reached the destination silicon object, or when the maximum number of hops have been spanned, the data must land on an LL register. Thereafter the data is either relaunched or is consumed by the destination object. For example, as shown in fig 3.4, incoming data from East or West PL channel lands on the East/West LL register, after which either the data is used by the silicon object or it is relaunched through the East launch multiplexer.

3.2 Finite Domain Constraints

Finite domain (FD) constraints are a field of constraint logic programming, which facilitate the representation of, and solution for, discrete search problems [51]. A variable X is constrained to take a value from a set D, where D is referred to as the domain of X. When all variables in a constraint satisfaction problem are bound to domains of finite cardinality, the problem is said to be a Finite Domain Constraint problem. The constraint store CS is a repository of constraints and domains of variables. The finite domain solver applies these constraints to reduce the cardinality of the domains of variables resident in the constraint store. Let is consider two variables X and Y, where $X \in D_1$, $X \in D_2$, and $Y \in D_3$ are constraints posted to the constraint store, restricting the domains of X and Y. In this case, the constraint store becomes $CS_{XY} = (X \in D_1) \wedge (X \in D_2) \wedge (Y \in D_3)$. The conjunction of two basic constraints involving X allows the solver to infer that the domain of X must be equal to $D_1 \cap D_2$. A constraint store is said to be consistent when for every variable M in the store, $M \notin \emptyset$, or in other words, the solver has not determined that no value can be bound to M which satisfies the set of constraints imposed on M. A finite domain constraint problem is comprised of a set of FD variables and a set of constraint defined over those variables. A constraint solver attempts to discover a solution to the constraint satisfaction problem through the repeated application of basic constraints to the set of variables captured in the constraint store, with the goal of reducing the cardinality of the domains associated with each variable. A solution to the problem is found when each variable has been grounded and the constraint store is consistent.

The Mozart programming system and constraint solver is a development platform for constraint programming in the Oz language [111]. Oz is a concurrent programming language with features of functional, logic, and constraint programming. Furthermore, Oz supports constraint-based problem solving using finite domain constraints. The Mozart constraint satisfaction engine employs three steps in determining a solution to a constraint satisfaction problem: Propagation, Distribution, and Search.

3.2.1 Propagation

A propagator is a thread of execution which is assigned the responsibility of enforcing the consistency of a single constraint. A constraint solver imposes these non-basic constraints on the variables contained within the constraint store to reduce the cardinality of their domains. If a reduction in a variable's domain is detected, the constraint store is updated to reflect the change. The solver then determines if any other propagators depend on the variable corresponding to the modified domain and invokes them to further shrink the domains of all associated variables. Thus, a change in one variable's domain is propagated to other variables through the constraint store. This process of imposing constraints, sharing information, and shrinking domains is called propagation.

For example, $x, y, z \in \{1, 2, ..10\}$ are three finite domain variables with a constraint $x+y = z$ defined over them. The constraint store contains a conjunction of basic constraints $x \in \{1, 2, ...10\} \wedge y \in \{1, 2, ...10\} \wedge z \in \{1, 2, ...10\}$, but it does not have sufficient information to narrow down any of these domains. The constraint $x+y = z$ realizes a propagator which eliminates the value 10 from the domains of x and y, and the value 1 from the domain of z, since these values cannot satisfy the constraint. Propagator $x + y = z$ is said to constrain the variables x, y, and z. Introducing another propagator $2x = y$ strengthens the store to $x \in \{1, 2, ...4\}$, $y \in \{2, 3, ..8\}$, and $z \in \{3, 4, ...10\}$. A new constraint $x = 1$ is now sufficient to ground y to the value 2 through propagator $2x = y$. The updated values for x and y cause propagator $x + y = z$ to bind z to the value 3.

Propagators affect the domain of a variable depending on the propagation scheme implemented: domain or interval. Domain propagation has the maximum impact on the

domain of a variable since it attempts to eliminate all infeasible values to shrink the domain. On the other hand, interval propagation only narrows down the lower and upper bounds of a variable's domain. Interval propagation is computationally less expensive and is usually preferred over domain propagation. The above example demonstrates interval propagation since the value 3 is not removed from y's domain after the introduction of $2x = y$ constraint. However, interval propagation does not yield incorrect solutions because $y = 3$ assignment causes a constraint violation which eventually removes the value 3 from y's domain.

While propagation is an important feature for solving constraint satisfaction problem, typically it is not sufficient to find a solution. If no new information can be discerned from the current state of the constraint store and the set of constraints, the propagation is suspended. Therefore, the solver must resort to distribution in order to facilitate propagation.

3.2.2 Distribution

Distribution is used to add new information to the constraint store when the solver reaches a point where it cannot proceed further toward a solution. Distribution creates two threads of contradictory sub-problems by cloning the search space and by inserting contradictory constraints in each clone, which are independently solved. A distribution point is introduced whenever propagation stalls. If propagation stalls in any of the newly created sub-problems, the distributor is invoked again. During this process, if the solver discovers a constraint violation, the solver backtracks to a previously cloned, but yet unexplored space in order to continue the search. Backtracking indicates that search proceeded along incorrect paths until the right path was selected, and is an important aspect of constraint solvers as it allows performing a quick test to determine if the current sub-problem could eventually lead to a valid solution or not. Mozart provides built-in distributors but also allows development of custom distribution strategies, which can benefit from knowledge of the problem domain.

For the example presented in sec. 3.2.1, propagation stalls in the absence of constraint $x = 1$. It is obvious that multiple values of x, y, and z satisfy $x + y = z$, but not enough information can be gathered to find a single solution. Distribution can resume propagation

by selecting one of the un-grounded variables and setting it equal to one value in one sub-problem and setting it not equal to that value in the other sub-problem. Typically, the variable with minimal domain size is selected. In the above example, selection of x is preferred because it has the smallest domain size and a choice point is created by setting $x = 1$, and $x \neq 1$. Figure 3.5 shows the distribution tree along with the domains of variables at each distribution step for finding possible solutions to this problem. Three solutions are found for $x = 1, 2$, and 3. However, when x is bound to the value 4, no value can be found for y which satisfies the remaining constraints in the store. Conequently the space with that assignmnt of $x = 4$ fails. In case of $x = 4$, $y \in \emptyset$ causes violation of $2x = y$ constraint, resulting in failure.

For problems representing large design spaces, even a small number of distribution steps can impose severe memory requirements, which may be impossible to satisfy causing the search to terminate prematurely. To address this issue, memory requirements for large search problems must be minimized.

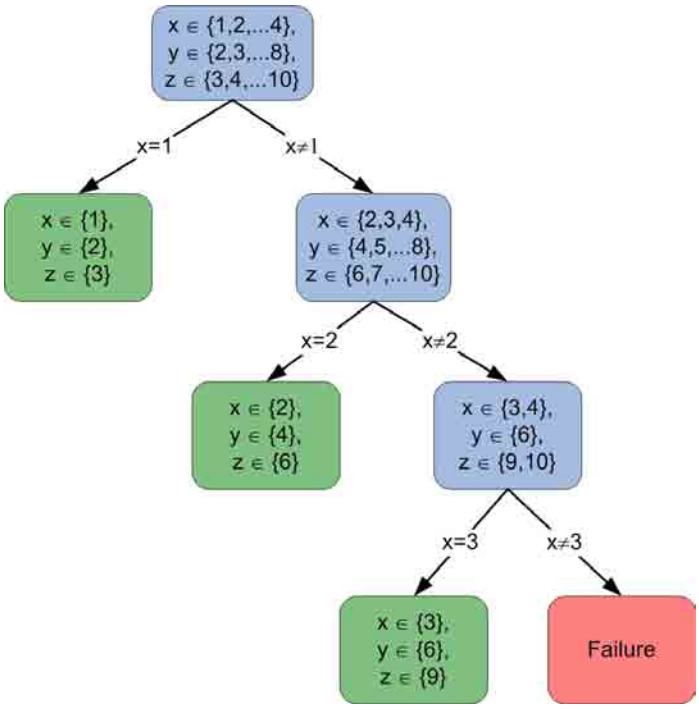


Fig. 3.5: Distribution steps for $x + y = z$.

Mozart allows a trade-off between memory and convergence time through a feature called re-computation. Typically, after each distribution step, the space state is saved so that it can be restored in the case of a backtrack. However, instead of saving space state after each distribution step, if the state is saved only after every S_{RC} steps, the memory requirement can be reduced by a factor of S_{RC}, where S_{RC} is a user defined value. In the event of a backtrack, all the intermediate states are re-computed from a previously saved state, increasing computation burden and search convergence time. Figure 3.6 shows a distribution tree with step size $S_{RC} = 3$. The search space is cloned in distribution steps 1, 4, and 7. After step 6, the left search path fails and the previous stable state, which happens to be the state at step 6, must be restored. However, the space was cloned only at step 4 requiring re-computation of the space state at steps 5 and 6, which increases computation burden and search time while reducing memory requirements.

The order in which variables are selected for distribution also plays an important role during distribution. All distributor implementations used in this research use a variable

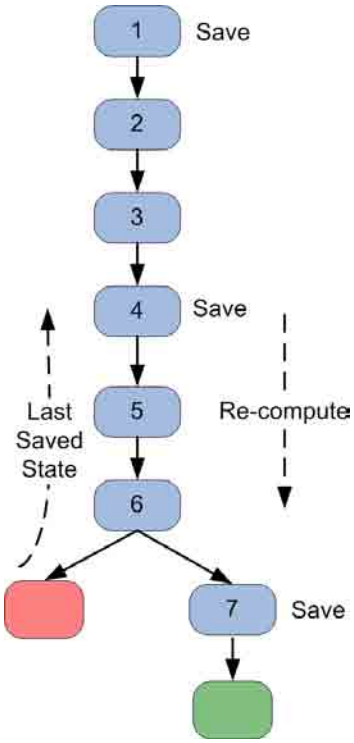


Fig. 3.6: Recomputation with step size $S_{RC} = 3$.

selection strategy called first fail. In first fail an ungrounded variable with the smallest domain is selected for distribution. Even though first fail guarantees that the selected variable has the smallest domain size, it is possible that a large number of values in its domain do not yield a valid solution, resulting in extensive backtracking. If multiple variables have the smallest domain size then any of these variables can be selected and the choice of variables selected first can make a significant difference in the search convergence time. A direct implication of variable selection is that a small sized problem may not always converge faster using lesser memory as compared to a large sized problem. However, search performance can still be improved through efficient backtracking and propagation.

Once a variable is selected, a value from its domain can be selected in various ways. This selection impacts the size of the search tree. In this research, the following two first fail strategies are used. The first one is the standard first fail which selects the least possible value in a variable's domain. The second one is a first fail split domain strategy which splits the domain of a variable and uses either the lower or the upper part of the domain. The effectiveness of these strategies in reducing the size of the search tree depends on the nature of the problem and neither one of them is necessarily superior to the other.

3.2.3 Search

Any constraint problem can have zero or more solutions. The path to a solution depends on how the distribution tree is traversed and it impacts the search time as well as the memory requirement. The search for a solution can explore the tree depth first, breadth first, or may use a heuristic search technique. Depth first exploration of a tree is considered to be more efficient in terms of memory requirement and is typically preferred over breadth-first search.

Chapter 4

Resource Allocation and Scheduling

The FPOA has a deterministic timing network and relies upon the input design specification to provide communication delays between connected components. Typical Data Flow Graphs (DFGs) do not capture this information and require additional processing to determine communication delays. This chapter describes the methodology used in this research to allocate computational resources and schedule a DFG to generate a delay annotated data flow graph, henceforth referred to as Timed Data Flow Graph (TDFG).

Section 4.1 describes the need for transforming a DFG into a TDFG and is followed by sec. 4.2, which explains the concept of FPOA resources. Section 4.3 explains and formalizes the resource allocation and scehduling problem from an FPOA perspective, and algorithms to solve these problems are proposed in sec. 4.4. In sec. 4.5, an approach for improving the placeability of a schedule is discussed. Finally, sec. 4.6 presents the proposed Oz-based constraint satisfaction approach for resource allocation and scheduling.

4.1 Data Flow Graph

A DFG is a directed graph that captures the data dependency among various operations. A DFG is represented by

$$G = (V, E), \tag{4.1}$$

where V is a set of vertices or nodes representing an operation and E $\in V \times V$ is a set of directed edges. Each node, v $\in$ V, has zero or more inputs and outputs. An edge connects the output of one node to the input of another and models a data dependence relationship between two nodes. The function represented by a node can only be executed when all its inputs are available. Upon execution, a node consumes input data and produces output data according to its functional behavior.

DFGs are frequently used for modeling system behavior. However, a DFG does not provide specific information on the type of physical resource, execution latency, communication delay, or the order of execution of different operations. This information is obtained by allocating resources in an architecture to nodes of a DFG and analyzing the data dependence relations among various nodes to establish the order of execution of operations. The output of this process is a TDFG which is depicted as G_T in eq. (4.2),

$$G_T = (V_T, E_T, D_e), \tag{4.2}$$

where V_T is a set of vertices or nodes representing resources, $E_T = V_T \times V_T$ is a set of directed edges, and $D_e \subset N$ is the set of communication delays along edges E_T, where N is the set of natural numbers. Each edge $e = (v_{src}, v_{dst}) \in E_T$ connects a pair of nodes, where the communication delay along edge e is given by $Delay : E_T \rightarrow D_e$. The term "resource" refers to one of many physical objects present in an architecture and not to a specific physical object as described in sec. 4.2.

4.2 Resources in an FPOA

The FPOA Arrix architecture consists of 256 ALU, 64 MAC, and 80 RF objects. An ALU object can perform arithmetic and logical operations, whereas MAC and RF objects are used for multiply-accumulate and load/store operations, respectively. All these objects collectively represent the computational resources available on an FPOA. In order to separate the placement process from allocation and binding, a set of virtual resources, $\Re_{FPOA}$, is defined in eq. (4.3).

$$\Re_{FPOA} = \{ALU_0..ALU_{255}, MAC_0..MAC_{63}, RF_0..RF_{79}\} \tag{4.3}$$

The resource set $\Re_{FPOA}$ contains 256 ALU resources, 64 MAC resources, and 80 RF resources and represents the pool of resources to which operations can be allocated. In the above discussion, placement refers to the process of associating each virtual resource with a

physical object on the chip and is described in Chapter 5. Allocation is the determination of the number of virtual resources that a particular scheduled DFG requires, while binding is the process of associating each DFG operation with a specific virtual resource.

In addition to the arithmetic and logic processing elements, each ALU contains an eight state instruction state machine which can execute one instruction in each state as shown in fig. 4.1. Set $S = \{0..7\}$ represents the eight states of an ALU's state machine. The state machine is always initialized to state 0 which is executed first by default. After executing the instruction in state n, control is automatically transferred to state $n + 1$, unless the current instruction causes a branch to a different state.

The execution latency of a resource $r \in \Re_{FPOA}$ is defined as $Latency : \Re_{FPOA} \rightarrow Z+$ and is equal to the number of clock cycles required to execute an instruction by a physical object corresponding to the resource r, where Z is the set of non-negative integers. Table 4.1 shows the latency of ALU, MAC, and RF objects. ALU latency is one clock cycle in each state in the instruction sequence. MAC has a two clock cycle latency but a single cycle throughput due to its pipelined implementation. Latency of an RF object can be one or two clock cycles depending on the mode of operation.

4.3 Resource Allocation and Scheduling

The process of implementing a design on an FPOA begins with an unschduled DFG. Resources are allocated to nodes of a DFG and each node is assigned a start time to

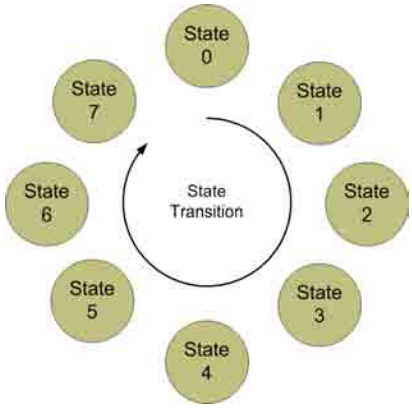


Fig. 4.1: Instruction state machine of an ALU.

Table 4.1: Latency of FPOA objects.

Object	**Latency (Clock cycles)**
ALU	1
MAC	2
RF	1 or 2

generate a TDFG. The following provides an in-depth description of the resource allocation and scheduling methods employed in this research.

4.3.1 Resource Allocation

In the context of an FPOA, allocation refers to the process of assigning a compatible virtual resource to a node in a DFG. Equation (4.4) defines a function $Alloc : V \rightarrow \Re_{FPOA}$ which assigns a virtual resource to a node v.

$$Alloc(v) = r \text{ , only if } r \in \Re_{FPOA} \ \wedge \ NodeType(v) = ResType(r), \tag{4.4}$$

where $v \in$ V is a node in the DFG and $\Re_{FPOA}$ is the virtual resource set of an FPOA. $NodeType : V \rightarrow OprType$ is a function that indicates the type of operation performed by v, where $OprType = \{ALU, MAC, RF\}$ represents the set of all arithmetic, logical, and load/store type of operations supported by an FPOA. Similarly, function $ResType : \Re_{FPOA} \rightarrow OprType$ gives the type of operation supported by a resource $r \in \Re_{FPOA}$. $Alloc()$ assigns a resource to a node only if the resource is capable of executing the operation represented by that node.

Consider node v_3 in fig. 4.2, which represents an addition operation that must be executed on an ALU because only ALUs support addition operations. Thus, the node type of v_3 is ALU. Resources r_1 and r_2 are both ALU instances and either of them can be allocated to v_3. In fig. 4.2, r_1 is allocated to v_3, while r_2 remains available for assignment to another compatible node in the DFG.

The allocation performed by $Alloc()$ assumes unlimited resources which is not a valid assumption. In a real-world scenario, the number of resources is finite and a good allo-

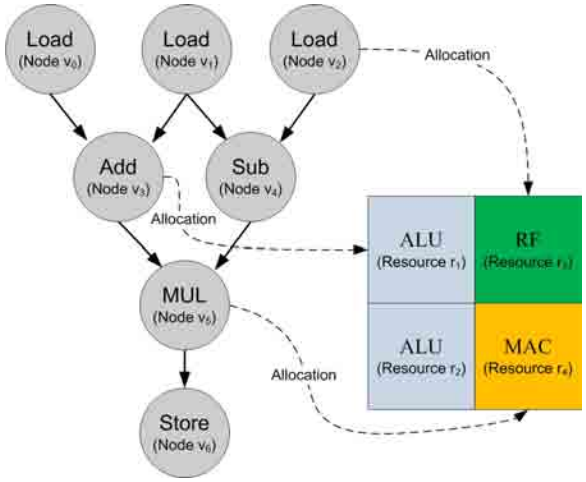


Fig. 4.2: Allocating resources to nodes of a DFG.

cation scheme must take into account the availability of resources as well as the cost of each assignment. Moreover, the above approach needs to be refined to include instruction states when allocating ALUs. While it is straightforward to allocate MACs and RFs to multiplication and load/store operations, respectively, the eight state ALU complicates the assignment because up to eight DFG nodes can be assigned to the same ALU resource. A simple solution to this problem is to treat each state in an ALU as a separate ALU resource and assign a node to an individual state. Thus, instead of 256 ALUs, we now have 256×8 ALU states for allocating arithmetic and logic operations. Equation (4.4) only maps a node to a physical resource and is no longer sufficient for capturing allocation involving ALU resources. The allocation of a node to an ALU must address the internal ALU instruction state machine as well.

Equation (4.5a) refines eq. (4.4) to capture the particular state $s \in S$ within an ALU and defines a new allocation function.

$$Alloc_{FPOA}(v) = (r, s), \text{ only if } r \in \Re_{FPOA} \ \wedge \ NodeType(v) = ResType(r) \ \wedge \ s \in \{0...7\} \tag{4.5a}$$

$$\forall (r, s), ResType(r) \in \{MAC, RF\} \Rightarrow \ s = 0 \tag{4.5b}$$

$Alloc_{FPOA} : V \rightarrow \Re_{FPOA} \times S$ is a one-to-one map from a node to the resourse set and instruction state set. However, MAC and RF resources do not have an instruction state machine. Without loss of generality, it is assumed that MAC and RF resources have a single-instrution state machine, and always execute instruction in state 0 as shown in eq. (4.5b). It should be noted that eq. (4.5a) permits up to eight operations to be co-located on the same ALU, but to different instruction sequencer states. The order in which instruction states are assigned to co-located ALU operations impacts the start times of these operations, as described in the next section.

4.3.2 Scheduling

The process of establishing the order of execution and start times of each node is called Scheduling. A schedule can be obtained by using straightforward scheduling methods such as As Soon As Possible (ASAP) or As Late As Possible (ALAP); ASAP scheduling attempts to execute a node at the earliest possible time step while ALAP procrastinates the execution unless it is absolutely warranted. Both ASAP and ALAP assume unlimited physical resources; for cases with limited set of resources, methods such as list scheduling are used. List scheduling attempts to minimize execution time subject to resource constraints.

Figure 4.3 shows a possible allocation for the DFG in fig. 4.2. The execution latency of each operation is equivalent to the latency of the corresponding physical resource. Given the execution latency of each node in a DFG, it is sufficient to find the execution start times $St(v_i)$ for each node $v_i \in V$ to determine a schedule, where $St : V \rightarrow Z$ represents the time at which v_i commences operation. The start time of a node depends on the start times of its preceding nodes. Consider edge $e_k = (v_s, v_d) \in E$, shown in fig. 4.4. The data dependence modeled by e_k mandates that the source of the edge must complete its operation prior to the commencement of the destination operation. Equation (4.6) specifies the relationship between the start times of v_s and v_d,

$$St(v_d) > St(v_s) + Latency(r_s), \tag{4.6}$$

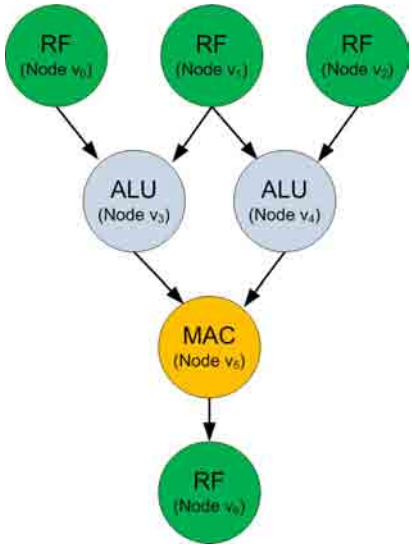


Fig. 4.3: An allocated DFG.

where $AllocFPOA(v_s)$ allocates the resource r_s to node v_s. By iteratively considering the edges in the input DFG, precedence relations between the start times of all connected nodes can be established.

ALU instruction state machine and architectural restrictions introduce additional interdependence between resource allocation and start times. Consider the allocation of two ALU type nodes v_i and v_j. If these nodes are assigned to the same ALU, they must belong to different states, say s_i and s_j, respectively. The instruction state machine takes exactly $|s_i - s_j|$ clock cycles to transition sequentially from the lower state to the higher state. Thus, the difference between the start times of these nodes, $|St(v_i) - St(v_j)|$, must be equal to $|s_i - s_j|$. This property of the state machine helps in deriving eq. (4.7), which defines a relationship between the start times of any two nodes that share an ALU.

$$St(v_i) = St(v_j) - (s_j - s_i) \tag{4.7}$$

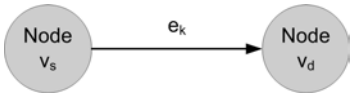


Fig. 4.4: Two nodes connected by edge e_k. Node v_s precedes node v_d.

The architecture of an FPOA has no two MAC resources that are nearest neighbors. Similarly, only eight RF objects are positioned on the FPOA to be pairwise nearest neighbors. The first restriction implies that for an edge $e_{mm} = (v_{ms}, v_{md}) \in E$, which begins and ends with two distinct MAC type nodes, $Delay(e_{mm})$ must be greater than 0. Equation (4.8) gives the relationship between the start times of nodes v_{ms} and v_{md}. The second restriction permits only eight edges in E, which connect two distinct RF type nodes, to have a zero delay.

$$St(v_{md}) > St(v_{ms}) + 1 \tag{4.8}$$

After resource allocation and scheduling, delay along an edge $e = (v_{src}, v_{dst}) \in E$ in a DFG is calculated using start times $St(v_{src})$ and $St(v_{dst})$, and the execution latency of the resource r_{src} assigned to v_{src}. Section 4.1 defines $Delay()$ to be a function which specifies the time between the production of a value until its consumption. Based on the above discussion, a formal definition of $Delay()$ is provided in eq. (4.9).

$$Delay(e) = St(v_{dst}) - St(v_{src}) - Latency(r_{src}) \tag{4.9}$$

There are circumstances when $Delay(e) = 0$, implying that the data produced by a source operation is consumed in the next clock cycle. A zero delay communication requires that the source and destination operations must either be located on FPOA objects which share a nearest neighbor connection, or be co-located on the same object. The latter case arises often when both source and destination are ALU type operations.

Even though Allocation and Scheduling appear to be mutually exclusive, they are actually interdependent. Unless we allocate resources, the execution latency of a set of independent nodes is unknown. However, for resource allocation, we need to know the number of physical resources available which can be determined only after finding a valid schedule. The above scenario presents a cyclic dependency between Allocation and Scheduling which must be resolved. Section 4.6 describes a simultaneous Allocation and Scheduling methodology to address this problem.

4.4 Resource Allocation and Scheduling Algorithms

The equations discussed in the previous sections provide the foundation for developing allocation and scheduling algorithms. These equations can be grouped into three categories: allocation, scheduling, and architectural constraints. Algorithms for each of these categories are presented in this section.

Algorithm 4.1 presents a methodology to allocate resources from an FPOA to the nodes of an input DFG. The algorithm begins with a set of unallocated nodes in line (4), where each node must be assigned a suitable FPOA resource and an available state in the instruction state machine of the resource. Initially, all resources in $\Re_{FPOA}$ are available for allocation, as shown in line (5). Line (6) initially assumes that all resources in $\Re_{FPOA}$ have an instruction state machine, and assigns a set of eight available states to each resource. As

Algorithm 4.1 Allocation algorithm

```
// Allocation
(1)  input  : DFG = (V,E)
(2)  input  : Resource set ℜ_FPOA
(3)
(4)  unallocated_node_set = V
(5)  available_resource_set = ℜ_FPOA
(6)  ∀r ∈ ℜ_FPOA, available_state_set_r = S
(7)  IsALU : V → {True, False}, where IsALU(v) = True iff NodeType(v) == ALU
(8)
(9)  while (unallocated_node_set ≠ ∅) do
(10)   select v_i ∈ unallocated_node_set
(11)   select r ∈ available_resource_set, such that NodeType(v_i) == ResType(r)
(12)   if (IsALU(v_i)) then
(13)     select s ∈ available_state_set_r
(14)     available_state_set_r = available_state_set_r − s
(15)     if (available_state_set_r == ∅) then
(16)       available_resource_set = available_resource_set − r
(17)     end
(18)   else
(19)     s = 0
(20)     available_resource_set = available_resource_set − r
(21)   end
(22)   Allocate (r, s) to v_i
(23)   unallocated_node_set = unallocated_node_set − v_i
(24) end
```

described earlier, MAC and RF resources are assigned a single-instruction state machine for the purpose of resource alloction, and are always assigned state $s = 0$, as described later in the algorithm. Line (7) defines a function $IsALU()$ which returns $True$ when applied to an ALU type node. Line (9) marks the beginning of an iteration. In lines (10) and (11), an unallocated node v_i is selected from the *unallocated_node_set* and a suitable resource $r \in \Re_{FPOA}$ is selected from the pool of available resources, such that $NodeType(v_i)$ is equal to $ResType(r)$. The condition in line (11) is necessary to ensure that r is capable of executing the operation represented by v_i. Resource allocation of ALU type nodes is handled differently than for MAC and RF type nodes. To this effect, line (12) determines the operation type of v_i.

If $NodeType(v_i) = ALU$, an available state s in ALU r is selected and is subsequently removed from the available pool of states in lines (13) and (14), respectively. The availability of r for further allocation depends on the number of available states in $available\_state\_set_r$. However, if s is the last available state in $available\_state\_set_r$, then lines (15) and (16) remove the fully utilized resource r from *available_resource_set*. On the contrary, if $NodeType(v_i) \neq ALU$, state s is set to zero in line (19), satisfying eq. (4.5b). Since r is of type MAC or RF, it cannot be allocated to any other node, and is removed from *available_resource_set* in line (20).

The pair (r, s) is assigned to v_i in line (22), completing v_i's resource allocation. Line (23) removes v_i from the *unallocated_node_set*, and the above process is repeated by selecting a new unallocated node. The algorithm terminates when a resource has been allocated to all nodes in the DFG, or $unallocated\_node\_set = \emptyset$. The algorithm requires $|V|$ iterations to perform allocation and has a time complexity $O(|V|)$, where $|V|$ is the number of nodes in the input DFG.

As mentioned at the beginning of this chapter, the purpose of resource allocation and scheduling is to transform a DFG into a TDFG. Allocation helps in determining the amount of time required to perform a specific operation, but it does not establish the order of execution of operations in the DFG. Algorithm 4.2 describes the procedure to schedule

Algorithm 4.2 Scheduling algorithm

```
// Scheduling
(1)  input  : DFG = (V,E)
(2)  input  : Resource set ℜ_FPOA
(3)  IsALU : V → {True, False}, where IsALU(v) = True iff NodeType(v) == ALU
(4)  V_ALU ={v ∈ V|IsALU(v) = True}
(5)
(6)   forall e = (v_src, v_dst) ∈ E do
(7)     let (r_src, s_src) = Alloc_FPOA(v_src)
(8)     if (St(v_src) ≤ St(v_dst) − Latency(r_src)) then
(9)       Delay(e) = St(v_dst) − St(v_src) − Latency(r_src)
(10)    else
(11)      goto line (7)
(12)    end
(13)  end
(14)
(15)  forall (v_i, v_j) ∈ V_ALU × V_ALU do
(16)    let (r_i, s_i) = Alloc_FPOA(v_i)
(17)    let (r_j, s_j) = Alloc_FPOA(v_j)
(18)    if ((r_i == r_j) && (v_i ≠ v_j)) then
(19)      if (St(v_i) ≠ St(v_j) − (s_i − s_j)) then
(20)        goto line (16)
(21)      end
(22)    end
(23)  end
```

operations in a DFG.

The initial step in scheduling is to build a set of precedence constraints to model data dependence among the nodes in the DFG. For each edge $e = (v_{src}, v_{dst}) \in E$, line (7) states that resource r_{src} is allocated to node v_{src} and $Latency(r_{src})$ denotes the number of clock cycles required by v_{src} to execute an operation. Line (8) posts a precedence constraint, shown in eq. (4.6), enforcing the data dependence relationship imposed by the edge. If the precedence constraint is satisfied, then the delay along the edge is calculated in line (9). Otherwise, control is transferred to line (7) and a new resource is allocated. Lines (6) to (13) iteratively establish precedence constraints and calculate communication delay for all the edges in the DFG.

Precedence constraints are necessary, but not sufficient, to determine start times of nodes assigned to ALUs. As mentioned in sec. 4.3.2, additional constraints apply to co-

located ALU type node pairs. For each node pair $(v_i, v_j) \in V_{ALU} \times V_{ALU}$, where v_i and v_j share the same ALU, lines (15) to (23) post the constraint specified in eq. (4.7).

The loop in lines (6) to (13) executes $|E|$ times, where $|E|$ denotes the number of edges in the input DFG. However, the condition in line (8) is data dependent and may require an unknown number of repetitions of line (7) to (11) during each iteration of the loop. Hence, the algorithm has a non-deterministic polynomial complexity. The similar behavior is observed in the second loop shown in lines (15) to (23). While the loop will execute 256×256 in the worst case, the condition in line (19) makes the run-time of the algorithm non-deterministic.

While the above algorithms can allocate and schedule a DFG, they can yield solutions that are not placeable due to the restrictions imposed by an FPOA's architecture on MAC and RF objects as discussed in sec. 4.3.2. An algorithm is proposed for constraining the start times of certain nodes in the DFG which correspond to MAC operations, such that invalid start times are removed from consideration. Algorithm 4.3 prohibits the scheduling of two adjacent data flow MAC operations such that NN communication is required. Lines (4) to (7) iterate over all the edges $e \in E$ in the DFG. For every edge connecting two MAC operations, line (6) requires that the start time of the destination operation is at least two clock cycles later than the source operation. The time complexity of Algorithm 4.3 is $O(|E|)$.

Unlike MAC operations, up to eight distinct RF node pairs are allowed to be nearest neighbors. The correct approach for removing invalid start times corresponding to RF op-

Algorithm 4.3 Disallow two MACs as nearest neighbors

```
// Constraints MAC MAC NN
(1)  input  : DFG = (V,E)
(2)  IsMAC : V → {True, False}, where IsMAC(v) = True iff NodeType(v) == MAC
(3)
(4)  forall e = (v_src, v_dst) ∈ E do
(5)    if (IsMAC(v_src) && IsMAC(v_dst)) then
(6)      St(v_dst) − St(v_src) > 1
(7)    end
(8)  end
```

erations is to determine the number of RF node pairs that need NN communication and to ensure that no more than eight such pairs are permitted as NNs. However, the exact number of RF node pairs requiring NN communication is not known until after the DFG has been scheduled, necessitating rescheduling if the number of zero delay edges between RF node pairs exceeds eight. A proactive sequential approach must either favor non-NN communication between RF node pairs, or it should initially be biased towards NN communication. Both these scenarios are inefficient because they do not consider the need for having a NN communication between RF node pairs, which may arise as scheduling proceeds. Section 4.6.2 describes a concurrent approach which determines the RF node pairs that require NN communication and allows up to eight RF operations to communicate with zero delay.

It can be argued that ALU objects are also subject to architectural limitations when considering nearest neighbor communication. However, the number of nearest neighbor ALU objects is far more than the corresponding numbers for MAC and RF objects, decreasing the odds of obtaining unplaceable schedules involving ALU nearest neighbors. Every ALU object has at least two nearest neighbor ALU objects, meaning nearest neighbor connectivity is much more readily available for inter-ALU communication. Even though algorithms can be developed to remove invalid start times resulting from excessive NN communication between ALU operations, the benefits may not outweigh the increase in computation burden. Hence, the current approach does not consider architectural limitations imposed on ALU objects. Instead, if an unplaceable schedule is obtained due to the unavailability of two nearest neighbor ALU objects, the DFG is rescheduled.

4.5 Schedule Relaxation

The scheduler implemented in this research tries aggressively to schedule adjacent operations to have zero interceding communication delay, implying the need for nearest neighbor placement. Maximizing zero delay connections typically results in shorter schedule lengths, decreases dependence on PL communication, and minimizes routing resource usage. A direct consequence of increasing the NN connections is the additional computational burden

on the placement tool, because it handles all the NN connections. A less obvious impact is on the placeability of the generated schedule. Algorithm 4.3 avoids some of the pitfalls that arise during placement, due to the attempted placement of an infeasible schedule for MAC operations. In spite of such algorithms, removing all possible invalid cases within the scheduler is computationally expensive as not all invalid cases are as straightforward as the MAC NN connections. For example, the architecture permits up to 16 ALUs to form a pipeline such that each ALU has a MAC nearest neighbor and every connected ALU pair is a nearest neighbor. To avoid pipelines longer than 16 ALUs, the scheduler must constantly monitor the start times of all the objects forming this pipeline, incurring computational overhead. Considering several invalid cases, not all of which affect a single design, incorporating unplaceable schedule avoidance in the scheduler can unnecessarily complicate the implementation and severely affect the performance by increasing the computational burden. Moreover, as described later in sec. 4.6, the resource allocation and scheduling algorithms are interdependent problems which benefit from a concurrent implementation. However, a concurrent implementation does not guarantee a specific order for assigning start times to objects, resulting in possible late detection of invalid cases and extensive rescheduling. Thus, early detection and avoidance of invalid cases is not always possible. An alternative solution to fix the unplaceable schedule is to generate an initial schedule, analyze it to identify invalid cases, and adjust the schedule to make it placeable. Since the analysis of the schedule and identification of invalid cases can be performed efficiently using a sequential procedure, it is implemented in C++ as a separate tool called Schedule Analyzer (SA), which is used after an initial schedule is obtained.

The purpose of the SA is to search for unsatisfiable architectural requirements imposed by a schedule. One such example is when two ALUs are connected by zero delay, where each has three MAC nearest neighbors. While many ALUs on the FPOA are nearest neighbors of three MACs, no two such ALUs can themselves be nearest neighbors, rendering the scheduled design unplaceable. The schedule analyzer identifies many of the unsatisfiable architectural requirements and makes suggestions for relaxing the delay along edges to

improve the placability of the schedule. A second round of scheduling takes these suggestions into account and generates a new schedule. If no suggestions were made, then the original schedule itself is used for placement. Additionally, operation merging also creates scenarios with large numbers of nearest neighbors for a particular ALU resource, that are impossible to place on the FPOA. For example, if two ALU operations, each with two MAC nearest neighbors are merged, then the result is a single ALU object with four MAC neighbors, as shown in fig. 4.5. Since no single ALU has four MAC nearest neighbors, such a schedule cannot be placed. In order to fix these issues, the schedule analyzer prevents merging of operations that lead to unplaceable schedules.

4.6 A Finite Domain Model for Allocation and Scheduling

The previous sections present the formulation of the fundamental relationships guiding the allocation and scheduling problem. This section discusses the realization of these relationships as a constraint satisfaction problem using Oz. Both allocation and scheduling are interdependent problems that can benefit from the concurrency model offered by Oz. Both problems are translated into a separate finite domain model, which are then jointly issued to the FD solver. Information is shared between these finite domain models through common variables. The subsequent sections describe the implementation of allocation and scheduling as finite domain constraint satisfaction problems.

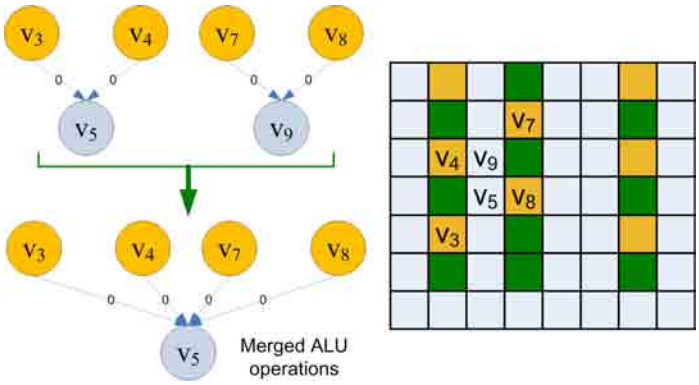


Fig. 4.5: ALU merging resulting in an unplaceable schedule.

4.6.1 A Finite Domain Model for Allocation

Figures 4.6, 4.7, and 4.8 provide the Oz implementation of the finite domain constraint model of Algorithm 4.1. The finite domain constraint model is functionally equivalent to the corresponding algorithm, but is formulated to leverage the concurrent constraint-based structure used by the Oz solver. Figure 4.6 describes the procedure *AllocInitialize*, which operates on the nodes in the input DFG. These nodes are supplied in the variable *DFGNodeList*, which is a list of tuples, one tuple per node, of the form [[$Node\_Id_1$,

```
(1)  proc {AllocInitialize DFGNodeList N ?TDFGNodeTuple}
(2)     ALU = 0
(3)     MAC = 1
(4)     RF = 2
(5)  in
(6)     TDFGNodeTuple = {Tuple.make nodeRecTup N}
(7)     {List.forAll DFGNodeList
(8)        proc {$ NodeElem}
(9)           Node_Num Node_Type                    %Local variables
(10)       in
(11)          Node_Num|Node_Type|_ = NodeElem
(12)          Node_Rec = {FD.record nrec [num type state res_num start_time]
(12a)               0#FD.sup}
(13)          Node_Rec.num = Node_Num
(14)          Node_Rec.type = Node_Type
(15)          if (Node_Rec.type == ALU) then        %ALU
(16)             Node_Rec.state ::: 0#7
(17)             Node_Rec.res_num ::: 1#256
(18)          end
(19)          if (Node_Rec.type == MAC) then        %MAC
(20)             Node_Rec.state =: 0
(21)             Node_Rec.res_num ::: 1#64
(22)          end
(23)          if (Node_Rec.type == RF) then         %RF
(24)             Node_Rec.state =: 0
(25)             Node_Rec.res_num ::: 1#80
(26)          end
(27)       TDFGNodeTuple.NodeNum = Node_Rec
(28)       end
(29)    }
(30) end
```

Fig. 4.6: Oz implementation for initializing allocation problem.

$Node\_Type_1$] [$Node\_Id_2$, $Node\_Type_2$] ...], where $Node\_Id$ and $Node\_Type$ are the id and type of a node in the DFG, respectively. *AllocInitialize* is responsible for creating a set of data structures, one per node in the DFG, which hold allocation information, such as the variable representing the to-be-determined resource number, the instruction state assigned to the node, and the variable holding the to-be-determined start time of the operation. *AllocInitialize* accomplishes this by iterating over all elements of the list *DFGNodeList*, and performing some context-specific initializations, depending on the element's operation type.

Lines (2) to (4) declare symbolic constants for ALU, MAC, and RF resources to facilitate the implementation of *AllocInitialize* by employing the following numerical encoding for resource type in an FPOA: type 0 represents an ALU resource, type 1 represents a MAC resource, and type 2 represents an RF resource. For a DFG with N nodes, line (6) defines a tuple $TDFGNodeTuple$, which is similar to a length-N array, with labels for storing attributes of each node. Line (7) provides the looping construct, which applies the procedure defined at line (8) in turn to each element, *NodeElem*, of *DFGNodeList*. Line (7) separates the pair *NodeElem* into two variables: *Node_Num* and *Node_Type*.

Line (12) defines the data structure $Node\_Rec$, referred to as a finite domain record, for *NodeElem*, which contains several named fields, each of which is bound to a finite domain variable. These fields include *num*, *type*, res_num, *state*, and $start\_time$, which represent a node's id, type, to-be-determined resource allocation, to-be-determined instruction state, and the start time of operation represented by the node. For any given node, a combination of *num*, *type*, res_num, and *state* fields is sufficient to specify resource allocation as described in eq. (4.5). The field, $start\_time$, is reserved for later use during scehduling. Lines (13) to (26) evaluate the type of operation the node represents, and based on that operation type, restrict the domains of the finite domain variables associated with the *state* andres_num fields of $Node\_Rec$.

For example, the FPOA offers 256 ALU resources, each of which is assigned a unique integer id from 1 to 256. The finite domain variable res_num represents the binding of an ALU operation to an as-yet unplaced ALU object on the FPOA and is restricted to take

```
(1)   proc {MakeRFMACDistinct TDFGNodeTuple Res_Type}
(2)      ResList                              %Local variable
(3)   in
(4)      ResList = {Record.foldL TDFGNodeTuple
(5)         proc {$ Accum NodeElem ?Output}
(6)            if (NodeElem.type == Res_Type) then
(7)               Output = {List.append [Node_Rec.res_num] Accum}
(8)            end
(9)         end
(10)     nil}
(11)     {FD.distinct ResList}                %Apply Distinct Constraint
(12)  end
```

Fig. 4.7: Oz implementation for imposing distinct constraints on MAC and RF.

on a value from 1 to 256 in line (17). Multiple ALU operations can be bound to the same ALU resource, since ALUs support state-based sequencing of up to eight operations. Since each ALU operation must be bound to a unique (resource, state) pair, the *state* field can be bound to a value from 0 to 7. Since MAC and RF resources do not support multiple operations, the *state* field for MAC or RF type operations is always bound to a value 0, as shown in lines (20) and (24).

AllocInitialize does not restrict two nodes from being bound to the same resource. The *MakeRFMACDistinct* procedure, given in fig. 4.7 addresses this issue for MAC and RF operations. Lines (4) to (9) of *MakeRFMACDistinct* create a list containing all the *res_num* variables. Line (4) initiates a folding operation which applies the procedure defined in line (5) to each element *NodeElem* of *TDFGNodeTuple*, where the input argument *Accum* is the accumulator in which the result of the previous invocation or the initial value specified in line (10) is passed, and the last argument *Output* is the variable which holds the result of the current invocation. Depending on the value of *Res_Type*, a list containing all the *res_num* variables corresponding to MAC or RF operations is created. Line (11) invokes the FD.distinct constraint, provided as part of the Mozart libraries, which requires all the variables in *ResList* to have unique values. *MakeRFMACDistinct* must be invoked twice, once for MAC operations and once for RF operations.

Allocating nodes to ALUs is more complicated and requires a different approach than

```
(1)   proc {MakeALUDistinct TDFGNodeTuple ?ALUResLISt}
(2)      ALU = 0
(3)   in
(4)      {Record.forAll TDFGNodeTuple
(5)         proc {$ NodeElem}
(6)            ALUResStateProd              %Local variable
(7)         in
(8)            if (NodeElem.type == ALU) then
(9)               ALUResStateProd = {FD.int 0#2047}
(10)              ALUResStateProd =: (Node_Rec.res_num-1) * 8
(10a)                        + Node_Rec.state
(11)              ALUResList = {List.append ALUResList [ALUResStateProd]}
(12)           end
(13)        end
(14)     }
(15)     {FD.distinct ALUResList}          %Apply Distinct Constraint
(16)  end
```

Fig. 4.8: Oz implementation for imposing distinct constraint on ALU.

MakeRFMACDistinct. As described above, two ALU operations can be bound to the same ALU resource, as long as they are allocated to distinct execution states within that resource. In order to simplify the checking of the requirement that no two ALU operations are allocated to the same (resource, state) pair, the procedure *MakeALUDistinct*, given in fig. 4.8, introduces a new parameter in line (8) called *ALUResStateProd*. This variable is assigned an encoding of the (resource, state) pair as a single number, the result of the product of the bound ALU resource number with the state number. Each unique (resource, state) pair produces a unique integer, allowing the Oz distinct constraint to be imposed to require unique bindings for each ALU operation.

To facilitate stronger propagation, a redundant constraint called *MaxEightALUs*, not shown in the figure, is added to permit a maximum of eight nodes to be assigned to a single ALU instance. This constraint is redundant because *MakeALUDistinct* ensures that no more than eight nodes can share an ALU. However, the *MaxEightALUs* constraint can reduce the domains of the *res_num* finite domain variables earlier in the search process. *MaxEightALUs* utilizes the Global Cardinality Constraint (GCC) [112], which is similar

to FD.distinct but is computationally more expensive and is not included in the Mozart library. The GCC-based C++ implementation of *MaxEightALUs* is shown in Appendix A.

The implementations described above facilitate the mapping of operations in a DFG on to a set of FPOA resources. However, the above discussion does not consider the influece of operation start times on resource allocation. The next section describes the realization of Algorithms 4.2 and 4.3 as constraint satisfaction problems and the impact of scheduling on allocation.

4.6.2 A Finite Domain Model for Scheduling

Algorithm 4.2 consists of two separate loops. The first loop establishes precedence relationships between any two nodes which share an edge. The second loop targets ALU-type nodes and establishes additional precedence relationsips to account for the impact of states on start times of nodes that are assigned to the same ALU instance. The procedure *SchedulingInit*, given in fig. 4.9, implements the first loop of Algorithm 4.2. *SchedulingInit* makes use of the *TDFGNodeTuple* data structure that is generated by procedure *AllocInitialize*. The procedure iterates over all pairs of nodes in the DFG which are connected by an edge, and posts a precedence constraint relating the start times of the nodes. Line (4) provides a loop construct with applies the procedure given in line (5) to elements of *DFGEdgeList*, which is a list of tuples, one tuple per edge, of the form [[Src_Node_1 Dst_Node_1] [Src_Node_1 Dst_Node_1] ...], where Src_Node and Dst_Node represent the node number of the source and destination nodes of an edge in the DFG, respectively. Line (8) separates the source and destination node numbers into two variables: $SrcNode$ and $DstNode$. Depending on the type of source node, the latency Lat, of the source node is assigned a value of 1 or 2. It is assumed that RF resources operate in RAM mode which has a latency of 1. Line (16) declares $Delay$ as a finite domain variable. Line (18) implements the precedence constraint that was described in eq. (4.6) and line (19) computes the communication delay between two nodes accoding to eq. (4.9). The output of *SchedulingInit* is a list of tuples, one tuple per edge, of the form [[Src_Node_1 Dst_Node_1 $Delay_1$] [Src_Node_2 Dst_Node_2 $Delay_2$] ...], where $Delay$

```
(1)  proc {SchedulingInit DFGEdgeList TDFGNodeTuple ?TDFGEdgeList}
(2)     MAC = 1
(3)  in
(4)     TDFGEdgeList={List.foldL DFGEdgeList
(5)        proc {$ Accum EdgeElem ?Output}
(6)           SrcNode DstNode SrcStTime DstStTime Delay Lat     %Local variables
(7)        in
(8)           SrcNode|DstNode|_ = EdgeElem
(9)           SrcStTime = TDFGNodeTuple.SrcNode.start_time
(10)          DstStTime = TDFGNodeTuple.DstNode.start_time
(11)          if (TDFGNodeTuple.SrcNode.type == MAC) then
(12)             Lat = 2                          %Latency of MAC resource
(13)          else
(14)             Lat = 1                          %Latency of ALU or RF resource
(15)          end
(16)          Delay =: {FD.int 0#FD.sup}
(17)
(18)          SrcStTime <: DstStTime - Lat
(19)          Delay =: DstStTime - SrcStTime - Lat
(20)          {NoTwoMacNN SrcNode DstNode TDFGNodeTuple}
(21)          Output = {List.append [[SrcNode DstNode Delay]] Accum}
(22)       end
(23)    nil}
(24) end
```

Fig. 4.9: Oz implementation for initializing scheduling algorithm.

represents the communication delay along the edge (Src_Node_1 Dst_Node_1). Line (20) in *SchedulingInit* invokes procedure *NoTwoMACNN*, which prohibits two MACs from being NNs, and is described later in this section.

The procedure *ALUPrecedenceConstraint*, which is illustrated in fig. 4.10, implements a single iteration of the second loop in Algorithm 4.2 and must be invoked for every node pair $(v_i, v_j) \in V \times V$ of the DFG. This procedure is applicable only to those edges which connect two ALU type nodes. Line (13) of the procedure posts a constraint according to eq. (4.7). This line presents a reified constraint which requires that either nodes V_I and V_J be allocated to different resources, or in the case of co-location, that the communication delay between nodes V_I and V_J be equal to the diference between their allocated states.

The Oz implementation of Algorithm 4.3 is given in fig. 4.11. The procedure *NoTwoM-*

```
(1)  proc {ALUPrecedenceConstraint V_I_Node V_J_Node TDFGNodeTuple}
(2)     V_I_StTime V_J_StTime V_I_ResNum V_J_ResNum V_I_State V_J_State
(3)     ALU = 0
(4)  in
(5)     V_I_StTime = TDFGNodeTuple.V_I_Node.start_time
(6)     V_J_StTime = TDFGNodeTuple.V_J_Node.start_time
(7)     V_I_ResNum = TDFGNodeTuple.V_I_Node.res_num
(8)     V_J_ResNum = TDFGNodeTuple.V_J_Node.res_num
(9)     V_I_State = TDFGNodeTuple.V_I_Node.state
(10)    V_J_State = TDFGNodeTuple.V_J_Node.state
(11)    if (TDFGNodeTuple.V_I_Node.type == ALU) then
(12)       if (TDFGNodeTuple.V_J_Node.type == ALU) then
(13)          ((V_I_ResNum \=: V_J_ResNum) + (V_I_ResNum =: V_J_ResNum) *
(13a)         (V_I_StTime - V_I_State =: V_J_StTime - V_J_State)) =: 1
(14)       end
(15)    end
(16) end
```

Fig. 4.10: Oz implementation for imposing precedence constraints on ALU type nodes.

ACNN applies only to those edges in the DFG, which connect two MAC type operations and must be invoked once for each such edge. Line (10) in *NoTwoMACNN* restricts a pair of adjacent MAC operations from requiring nearest neighbor communication, through the imposition of a constraint on their start times. Even though Algorithm 4.3 contains a loop, *NoTwoMACNN* does not implement a loop because it is repeatedly executed by procedure

```
(1)  proc {NoTwoMACNN SrcNode DstNode TDFGNodeTuple}
(2)     SrcStTime DstStTime
(3)     MAC = 1
(4)  in
(5)     SrcStTime = TDFGNodeTuple.SrcNode.start_time
(6)     DstStTime = TDFGNodeTuple.DstNode.start_time
(7)
(8)     if (TDFGNodeTuple.SrcNode.type == MAC) then
(9)        if (TDFGNodeTuple.DstNode.type == MAC) then
(10)          DstStTime - SrcStTime >: 1
(11)       end
(12)    end
(13) end
```

Fig. 4.11: Oz implementation to prohibit two MACs from being NN.

SchedulingInit, once for each edge in the DFG.

Similarly, fig. 4.12 presents the Oz procedure *LimitRFNN*, which stipulates that no more than eight edges connecting RF operations can require NN communication. Line (6) initiates a folding operation which iteratively applies the procedure in line (7) to each element *EdgeElem* of list *DFGEdgeList*. In line (10), the source and destination node number pair in *EdgeElem* is separated and is assigned to variables *SrcNode* and *DstNode*, respectively. Lines (15) to (20) count the number of zero delay edges, *NumZeroEdges*, using the reified expression of line (17) by evaluating if the difference between the start times of the source and destination operations is 1. Line (23) posts a constraint to limit the value of *NumZeroEdges*, implying that there can be a maximum of eight pairs of adjacent RF operations that may require NN communication.

```
(1)  proc {LimitRFNN DFGEdgeList TDFGNodeTuple}
(2)     NumZeroEdges
(3)     RF = 2
(4)  in
(5)     NumZeroEdges = {FD.int 0#FD.sup}
(6)     NumZeroEdges =: {List.foldL DFGEdgeList
(7)        proc {$ Accum EdgeElem ?Output}
(8)           SrcNode DstNode SrcStTime DstStTime NZEdge
(9)        in
(10)          SrcNode|DstNode|_ = EdgeElem
(11)          SrcStTime = TDFGNodeTuple.SrcNode.start_time
(12)          DstStTime = TDFGNodeTuple.DstNode.start_time
(13)          NZEdge = {FD.int 0#FD.sup}
(14)
(15)          if (TDFGNodeTuple.SrcNode.type == RF) then
(16)             if (TDFGNodeTuple.DstNode.type == RF) then
(17)                NZEdge =: Accum + (DstStTime - SrcStTime =: 1)
(18)                Output = NZEdge
(19)             end
(20)          end
(21)       end
(22)    0}
(23)    NumZeroEdges =<: 8
(24) end
```

Fig. 4.12: Oz implementation for limiting two RFs from being NN.

4.6.3 Distribution Strategy for Allocation and Scheduling

Constraint propagation alone may not be sufficient to implement a complete constraint solver. While propagation helps in shrinking the domains of finite domain variables, it may stall and require more information to proceed. As noted in Chapter 3, constraint distribution introduces choice-points in a constraint problem, resulting in two contradictory sub-problems. The order of selecting variables for distribution significantly impacts the performance of finite domain solver and often benefits from a domain specific custom distribution strategy.

Mozart provides built-in distributers that can be used for implementing the scheduler, but these distributers apply a one-blanket-fits-all approach without taking advantage of the problem domain. A built-in distributer using first fail strategy will always distribute on the FD variable with the smallest domain first, implying that distribution begins with *state* variables, followed by *res_num*, *ALUResStateProd*, and *start_time* variables. Experiments with the built-in distributer indicate that initially all *state* variables are grounded to the least value in their domain and are identical. Consequently, all *res_num* variables are distinct because no two *ALUResStateProd* can be same, implying that no ALU operations are merged. Instead of *state* variable, if either *res_num* or *ALUResStateProd* variable is selected first, then the scheduler aggressively merges ALU operations, which increases the likelihood of generating unplaceable schedules and directly affects the schedule length because collocated ALU operations cannot execute concurrently. Hence, a custom distribution strategy is required which prioritizes the scheduling of operations, followed by aggressive merging of operations on ALU.

The distribution strategy employed for solving the allocation and scheduling problem is derived experimentally and is given in fig. 4.13. The distribution starts with a list of ungrounded *start_time* variables and uses the first fail heuristic for variable selection. The list of variables is sorted according to domain size and a variable with minimal domain size is selected for distribution. Suppose ST is the selected variable. Since ST is not grounded, it has a domain $D_{ST} = [D_{min}...D_{max}]$ associated with it, which represents the values that

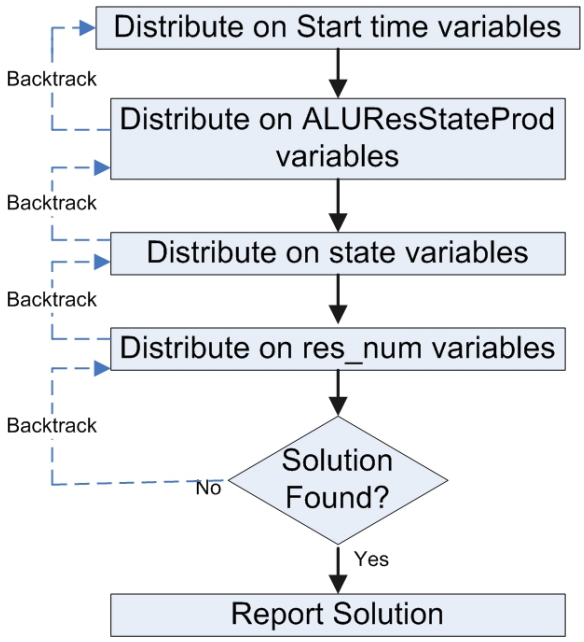


Fig. 4.13: Distribution strategy for scheduling.

can be assigned to it. D_{min} is the lower bound and D_{max} is the upper bound on the values that ST can take. A choice-point is created for ST by introducing two contradictory constraints $ST =: D_{min}$ and $ST \backslash =: D_{min}$. If a choice-point causes a propagator to fail, then the choice-point is discarded and distribution continues by backtracking to the previous stable state. After all *start_time* variables become singleton, distribution continues by repeating the above process for lists of ungrounded $ALUResStateProd$, *state*, and *res_num* variables. A solution is reported when a value has been assigned to all the variables such that all the constraints are satisfied. Occasionally, the Scheduler may not converge due to excessive memory requirements causing the search to terminate. As described in Chapter 3, recomputation can trade-off memory requirements with convergence time to avoid premature termination of the search.

Chapter 5

Delay Aware Placement

Placement is defined as an assignment of a set of operations to compatible physical objects in an FPOA, subject to a set of user-specified design constraints. As explained in Chapter 4, Allocation and Scheduling transform a DFG into a TDFG, assigning suitable resources to each operation. However, the exact locations of these operations or corresponding resources on an FPOA must still be determined.

The placement problem can be phrased as an assignment of a set of nodes in a TDFG to physical objects in an FPOA, subject to a set of user-specified design constraints. Communication delay along the edges in a TDFG impose constraints which must be satisfied in order to successfully implement the design. Since the methodology presented in this chapter attempts to place designs by satisfying communication delays, the approach is termed as "delay aware placement." The chapter is organized as follows. Section 5.1 builds the mathematical foundation for representing the objects on an FPOA. This mathematical model is used for formulating the placement problem as described in sec. 5.2. An algorithm and a constraint satisfaction approach for solving the placement problem are discussed in sec. 5.3 and 5.4, respectively. Section 5.5 provides a summary of the chapter.

5.1 A Formal Model for Objects in an FPOA

As stated previously, the FPOA architecture consists of three types of objects: ALU, MAC, and RF. These objects are arranged in a $XMax \times YMax$ square grid, where $XMax = YMax = 20$. Since these are physical objects on a silicon chip, they are often referred to as Silicon Objects (SO) to disambiguate them from other terms such as "resources" or "nodes" in a TDFG.

The 20×20 grid can be represented using Cartesian coordinates $(x, y) \in X \times Y$, where

$X = \{1, 2, ...XMax\}$ and $Y = \{1, 2, ...YMax\}$. As shown in fig. 5.1, the silicon object with coordinates (1,1) is located in the bottom left corner of the array. Objects on the array are each assigned a unique coordinate pair, where adjacency is marked by unit distance. The bottom right, top left, and top right corners are located at $(XMax, 1)$, $(1, YMax)$, and $(XMax, YMax)$, respectively.

A total of 400 different objects comprise the FPOA grid and are represented by the object set O_{FPOA}, which contains 256 ALU, 64 MAC, and 80 RF silicon objects. In order to refer to a particular object, we can either use its location or else assign it a unique id. The location is frequently used for calculating Manhattan distances between pair of objects. Since a location requires a pair of numbers, it can be cumbersome if used as a primary key for frequently accessing an object. However, since each object has a unique location, it is possible to define a formula for computing a numerical identifier based on an object's coordinates. This technique allows us to obtain the location of an object from its id and vice versa, without the need for a lookup table, and is explained below.

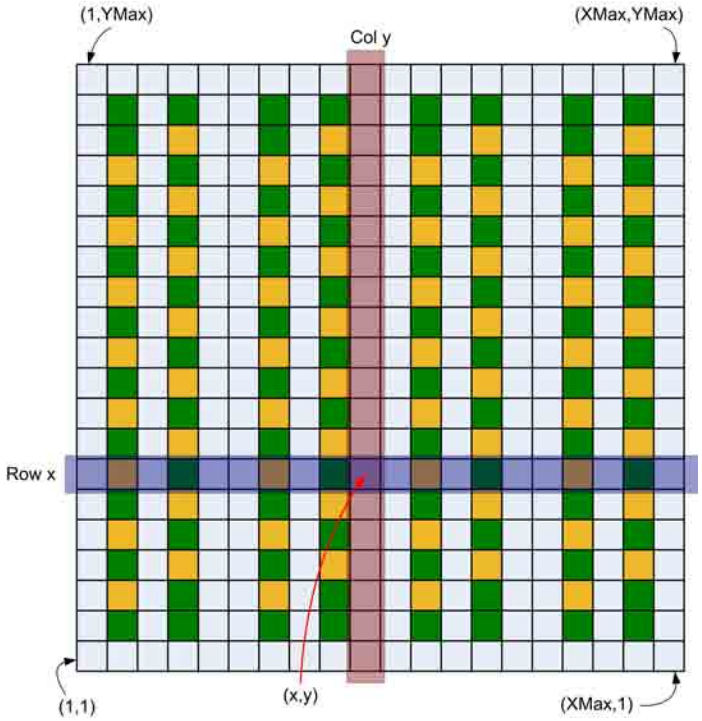


Fig. 5.1: Assigning Cartesian coordinates to silicon objects.

For an object $SO \in O_{FPOA}$, the function $Coord : O_{FPOA} \rightarrow X \times Y$ defines a mapping between the object and its coordinates $(SO_x, SO_y) \in X \times Y$. Using the coordinates (SO_x, SO_y), it is possible to assign a unique id to the object's location. The function $LocID : X \times Y \rightarrow N^+$, defined in eq. (5.1), maps the coordinates of an object to a unique numerical value in N^+, where N^+ is the set of natural numbers greater than 0. This unique value is referred to as the id of SO.

$$LocID(SO_x, SO_y) = (SO_y - 1) \times XMax + SO_x \tag{5.1}$$

Equation (5.1) assigns ids to objects using the row major order, which means that the difference between the ids of two adjacent objects in the same row is 1. Even though it is possible to use column major order for assigning the unique ids, using row major order offers certain advantages, which are discussed in sec.5.4.1.

In order to simplify the mapping between a silicon object and its unique location id, the function $ID : O_{FPOA} \rightarrow N^+$ is defined in eq. (5.2).

$$ID(SO) = LocID(Coord(SO)) \tag{5.2}$$

Additionally, we define $SO_{id} = ID(SO)$, for all silicon objects $SO \in O_{FPOA}$, as a shortened notation. Based on the above discussion, the set of ids of all silicon objects in the FPOA, ALU_SOid, can now be defined and is shown in eq. (5.3). Figure 5.2 shows a portion of the FPOA grid with object locations denoted by their Cartesian coordinates and the corresponding unique ids.

$$ALL\_SOid = \{n \in N^+ | \forall\ SO \in O_{FPOA},\ n = ID(SO)\} \tag{5.3}$$

Apart from its location, a silicon object is also characterized by the type of operation it performs. The function $ObjType : O_{FPOA} \rightarrow OprType$ gives the type of operation supported by an object $SO \in O_{FPOA}$, where $OprType = \{ALU, MAC, RF\}$ as defined previously in sec. 4.3.1.

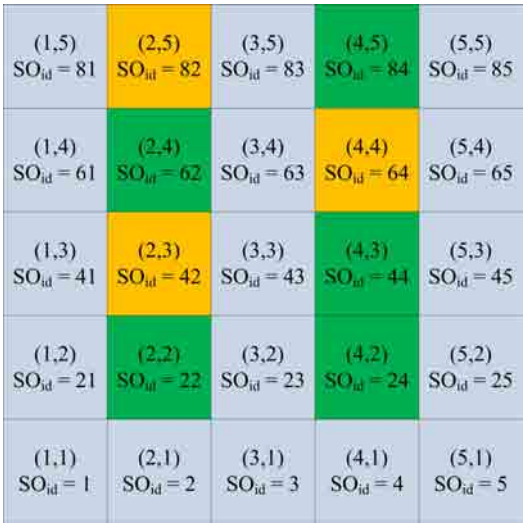


Fig. 5.2: Silicon object locations and corresponding SO_{id}.

Since the number of objects is finite and their locations on the FPOA is fixed, we can partition the set of all SO_{id} into mutually exclusive subsets, based on object type. This partitioning generates three subsets, one each for ALU, MAC, and RF objects, such that $ALL\_SOid = ALU\_SOid \cup MAC\_SOid \cup RF\_SOid$, where ALU_SOid, MAC_SOid, and RF_SOid are defined in eq. (5.4a-c).

$$ALU\_SOid = \{n \in N^{+} | \forall\ SO \in O_{FPOA},\ n = ID(SO)\ \&\ ObjType(SO) = ALU\} \quad (5.4a)$$

$$MAC\_SOid = \{n \in N^{+} | \forall\ SO \in O_{FPOA},\ n = ID(SO)\ \&\ ObjType(SO) = MAC\} \quad (5.4b)$$

$$RF\_SOid = \{n \in N^{+} | \forall\ SO \in O_{FPOA},\ n = ID(SO)\ \&\ ObjType(SO) = RF\} \quad (5.4c)$$

Communication delay between two silicon objects is a function of their proximity, as measured by Manhattan distance. A unit Manhattan distance is referred to as a *hop*. Manhattan distance between two points is defined as the sum of absolute differences of their respective x and y coordinates. Applying this concept to the FPOA grid, we obtain eq. (5.5), which defines the function $Dist : O_{FPOA} \times O_{FPOA} \rightarrow N^{+}$. This function computes the Manhattan distance between two silicon objects $SO\_Src, SO\_Dst \in O_{FPOA}$, where $(SO\_Src_x, SO\_Src_y) = Coord(SO\_Src)$, and $(SO\_Dst_x, SO\_Dst_y) = Coord(SO\_Dst)$.

$$Dist(SO\_Src, SO\_Dst) = |SO\_Src_x - SO\_Dst_x| + |SO\_Src_y - SO\_Dst_y| \ (hops) \quad (5.5)$$

5.2 Placement Problem

A TDFG is obtained from a DFG using the procedure described in Chapter 4. The nodes in a TDFG may represent operations that are co-located, which is eventually placed on the same silicon object. The number of nodes in the TDFG can be reduced by merging all the co-located nodes that share the same resource into a single node. The edges in the TDFG are preserved, except for cases where two merged nodes share an edge. In such a case, no external communication is required because the operations can communicate using internal registers, and the edge is removed. Placement and routing procedures are defined over this modified TDFG.

Since each node in a TDFG represents a resource that must be implemented by a silicon object we must develop a procedure that facilitates the association of a node with a corresponding silicon object. This procedure must also check whether a node and a candidate silicon object are compatible, i.e., whether the candidate SO is able to execute the node's operation. To define this procedure we introduce four new attributes for each node $v \in V_T$: v_{res_type}, v_{id}, v_x, and v_y. The first attribute denotes the node's type, or $v_{res\_type} = NodeType(v)$.

The second attribute, v_{id}, represents the id of the silicon object where the placement algorithm assigns v to execute. We define $VID(v) = v_{id}$ to denote the v_{id} attribute for all $v \in V_T$. If v is placed on silicon object SO, then $VID(v) = ID(SO)$, where $VID(v) = v_{id}$ and $VID(v) \in ALL\_SOid$. However, for a valid placement, v_{res_type} must be equal to $ObjType(SO)$. Due to the fact that a node v can only be placed on an object SO if $v_{res\_type} = ObjType(SO)$, the initial domain of v_{id} can be restricted to a subset of the

originally defined domain ALL_SOid, as shown in eq. (5.6).

$$v_{id} \notin \begin{cases} MAC\_SOid \cup RF\_SOid, & if\ v_{res\_type} = ALU \\ ALU\_SOid \cup RF\_SOid, & if\ v_{res\_type} = MAC \\ ALU\_SOid \cup MAC\_SOid, & if\ v_{res\_type} = RF \end{cases} \tag{5.6}$$

Furthermore, two nodes cannot be placed on the same silicon object and a silicon object cannot be assigned to two different nodes in a TDFG. For example, if two nodes $v_m, v_n \in V_T$ are assigned to two silicon objects, SO_m and SO_n, respectively, then $\forall v_m \neq v_n$, $VID(v_m) = ID(SO_m)$, $VID(v_n) = ID(SO_n)$, and $VID(v_m) \neq VID(v_n)$.

The last two attributes v_x and v_y are the coordinates of the silicon object on which node v is placed. Suppose that the placement algorithm determined that v is to be placed on silicon object SO, then $v_{id} = ID(SO)$, $v_x = SO_x$, and $v_y = SO_y$. Equation (5.7) uses eq. (5.1) to relate attributes v_{id}, v_x, and v_y.

$$v_{id} = LocID(v_x, v_y) \tag{5.7}$$

The goal of placement is to bind each node in a TDFG to exactly one silicon object. The placement problem can be initially defined as the determination of a one-to-one mapping of V_T to O_{FPOA}, such that operation type constraints are not violated. While the initial description of placement may appear straight forwad, placement is also subject to timing constraints, which impact object proximity. The FPOA requires delay aware placement, which means that the placement is restricted by the required communication delay between two connected objects. Objects in a FPOA communicate in two ways: NN connection or PL connection. Nearest neighbor connectivity uses special NN registers and supports zero clock cycle communication delay. In contrast, party lines are used for sending data to objects located farther away from the source. Irrespective of the type of network used, placement is subject to certain architectural restrictions which are discussed in the following sections.

5.2.1 Nearest Neighbor Communication

Any object can have a maximum of eight nearest neighbors as shown in fig. 5.3. Each local NN register of an object can output data to two of its adjacent neighbors. Similarly, each object can read data from NN registers of its nearest neighbors. NN connectivity is used when data produced from one object must be consumed by another object in the next clock cycle. While NN connections reduce communication delay, they introduce tight constraints which decrease placement flexibility.

Two silicon objects SO_NN_Src and SO_NN_Dst are NNs only if the proximity constraint specified in eq. (5.8) is satisfied.

$$|SO\_NN\_Src_x - SO\_NN\_Dst_x| \leq 1 \ \wedge \ |SO\_NN\_Src_y - SO\_NN\_Dst_y| \leq 1, \qquad (5.8)$$

where $(SO\_NN\_Src_x,\ SO\_NN\_Src_y)$ and $(SO\_NN\_Dst_x,\ SO\_NN\_Dst_y)$ are the Cartesian coordinates of silicon objects SO_NN_Src and SO_NN_Dst, respectively. Further, we define $Dist_{NN}(SO\_NN\_Src, SO\_NN\_Dst)$ as a shortened notation to denote the proxim-

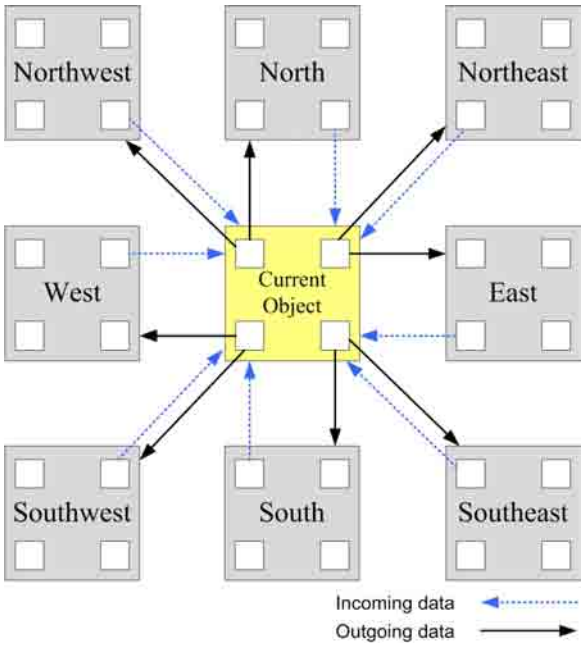


Fig. 5.3: Nearest neighbor input and output.

ity constraint presented in eq. (5.8). Since no two MAC objects can satisfy this criterion, Algorithm 4.3 forbids TDFGs from having any edge $e \in E_T$ connecting two MAC resources with $Delay(e) = 0$. The condition specified in eq. (5.8) is necessary, but not sufficient for NN communication, because two adjacent objects can communicate using Party lines.

5.2.2 Party Line Communication

Party lines are used for sending data over long distances when handling non-zero delay communication. Output data from a source object is routed through a party line and travels from one object to another till it reaches the destination. Figure 5.4 shows the variable number of hops that can be traversed in a single clock cycle. When operating at a clock frequency of 1GHz, data can travel up to four hops in a single clock cycle. A communication delay of n clock cycles between two silicon objects is implemented on an FPOA by routing the data through *(n-1)* LL registers on a PL network as shown in fig. 5.5. Equation (5.9) gives the maximum physical distance that can be traveled between a source and a destination object within n clock cycles when using PL connections.

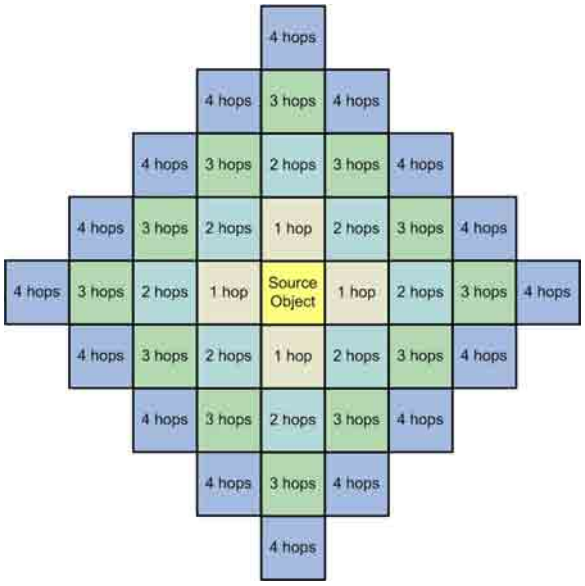


Fig. 5.4: Party line communication - four hops in one clock cycle.

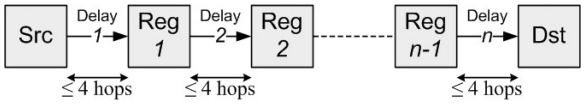


Fig. 5.5: Route with n clock cycle delay.

$$MaxDist_{PL}(n) = n \times 4 \ (hops) \tag{5.9}$$

When two nodes v_pl_src and v_pl_dst, connected by an edge e with $Delay(e) > 0$ in a TDFG are placed on an FPOA, they must use Party line network for communication. If nodes v_pl_src and v_pl_dst are placed on objects SO_PL_Src and SO_PL_Dst, respectively, then proximity constraint specified in eq. (5.10) must be satisfied.

$$Dist(SO\_PL\_Src, SO\_PL\_Dst) \leq MaxDist_{PL}(Delay(e)) \tag{5.10}$$

Equation (5.10) is a necessary and sufficient condition for placement involving objects that require PL communication. It eliminates all possible object locations that yield unroutable placement due to violation of delay requirements. In the absence of any other routes, it also guarantees the existence of at least one route between two connected objects. However, in a typical design, multiple routes exist which occupy routing resources and may result in a valid but unroutable placement.

The placement problem can now be refined as: $\forall v \in V_T$, find unique assignments $v_{id} = SO_{id}$, where $SO \in O_{FPOA}$, such that $v_{res\_type} = ObjType(SO)$ and all proximity constraints are satisfied.

5.3 Placement Algorithm

Algorithm 5.1 describes the methodology for placing a TDFG on an FPOA. This algorithm is divided into two parts: assign silicon objects to nodes that share an edge such that the proximity constraints are satisfied, and assign a silicon object to any unconnected nodes. The first part iterates over all the edges in the TDFG and places the connected node pairs first because they have less placement flexibility due to proximity constraints. The

second part places all the remaining unplaced nodes in the TDFG.

The inputs to the algorithm are a TDFG and the set of silicon objects in the FPOA, O_{FPOA}. Initially, none of the nodes in the TDFG are placed and all objects in O_{FPOA} are available for placement, as indicated in lines (3) and (4), respectively. Lines (6) to (35) form the first part of the algorithm. Line (6) initites a loop which iterates over all the edges in the TDFG. An edge $e = (v_{src}, v_{dst}) \in edge\_set$ is selected in line (7), such that e has the least communication delay among all edges in *edge_set*. This is to ensure that the node pair with most stringent constraint is selected first. Subsequently, the edge is removed from *edge_set*. Lines (9) and (10) determine if either the source node or the destination node has been placed previously. Depending on the case, one of the following three scenarios is applicable.

1. In the first case, neither the source node nor the destination node have been placed. Two silicon objects, SO_{src}, SO_{dst}, are selected from the available pool in line (11), such that v_{src}, So_{src}, and v_{dst}, So_{dst} are respectively compatible. In addition to the compatibility, SO_{src}, SO_{dst} must also satisfy the proximity constraints. Line (13) uses a shorthand notation of *if-then-else* and states that if $Delay(e)$ is 0 then SO_{src}, SO_{dst} are subject to NN proximity constraint, otherwise they must satisfy the PL proximity constraint. The id of SO_{dst} is assigned to v_{id} of v_{dst} to indicate that v_{dst} is placed on SO_{dst}. Lines (15) and (16) remove v_{dst}, and SO_{dst} from *unplaced_node_set* and *available_silicon_object_set*, respectively. Similarly, line (22) indicates that v_{src} is placed on SO_{src} and lines (23), (34) remove v_{src}, and SO_{src} from *unplaced_node_set* and *available_silicon_object_set*, respectively.

2. If a silicon object has been assigned to the destination node, then only the source node needs to be placed. Lines (18) to (19) select an available node SO_{src} from *available_silicon_object_set*, such that v_{src} and SO_{src} are compatible. Moreover, they also identify the silicon object SO_{dst} on which v_{dst} was placed by comparing $VID(v_{dst}$ with $ID(SO_{dst})$. Selection of SO_{src} is further subject to the proximity constraints

Algorithm 5.1 Placement algorithm

```
// Placement
(1)  input  : TDFG = (V_T, E_T, D_e)
(2)  input  : Silicon Object set O_FPOA
(3)  unplaced_node_set = V_T
(4)  available_silicon_object_set = O_FPOA
(5)  edge_set = E_T
(6)   while edge_set ≠ ∅ do
(7)    select edge e = (v_src, v_dst) ∈ edge_set with the least value of Delay(e)
(8)    edge_set = edge_set − e
(9)    if (v_src ∈ unplaced_node_set) then
(10)     if (v_dst ∈ unplaced_node_set) then
(11)       select SO_src, SO_dst ∈ available_silicon_object_set such that
(12)         NodeType(v_src) == ObjType(SO_src) and NodeType(v_dst) == ObjType(SO_dst)
(13)         (Delay(e) == 0) ? Dist_NN(SO_src, SO_dst) : Dist(SO_src, SO_dst) ≤ Delay(e) × 4
(14)       VID(v_dst) = ID(SO_dst)
(15)       unplaced_node_set = unplaced_node_set − v_dst
(16)       available_silicon_object_set = available_silicon_object_set − SO_dst
(17)     else
(18)       select SO_src ∈ available_silicon_object_set, SO_dst ∈ O_FPOA such that
(19)          NodeType(v_src) == ObjType(SO_src) and VID(v_dst) == ID(SO_dst)
(20)          (Delay(e) == 0) ? Dist_NN(SO_src, SO_dst) : Dist(SO_src, SO_dst) ≤ Delay(e) × 4
(21)     end
(22)     VID(v_src) = ID(SO_src)
(23)     unplaced_node_set = unplaced_node_set − v_src
(24)     available_silicon_object_set = available_silicon_object_set − SO_src
(25)   else
(26)     if (v_dst ∈ unplaced_node_set) then
(27)       select SO_src ∈ O_FPOA, SO_dst ∈ available_silicon_object_set such that
(28)          NodeType(v_dst) == ObjType(SO_dst) and VID(v_src) == ID(SO_src)
(29)          (Delay(e) == 0) ? Dist_NN(SO_src, SO_dst) : Dist(SO_src, SO_dst) ≤ Delay(e) × 4
(30)       VID(v_dst) = ID(SO_dst)
(31)       unplaced_node_set = unplaced_node_set − v_dst
(32)       available_silicon_object_set = available_silicon_object_set − SO_dst
(33)     end
(34)   end
(35)  end
(36)  forall v_rem ∈ unplaced_node_set do
(37)    select SO_avbl ∈ available_silicon_object_set such that
(38)      NodeType(v_rem) == ObjType(SO_avbl)
(39)    VID(v_rem) = ID(SO_avbl)
(40)    unplaced_node_set = unplaced_node_set − v_rem
(41)    available_silicon_object_set = available_silicon_object_set − SO_avbl
(42)  end
```

specified in line (20). Lines (22) to (24) map v_{src} to SO_{src} and subsequently remove v_{src}, and SO_{src} from the *unplaced_node_set* and *available_silicon_object_set*, respectively.

3. If only the source node has been placed, then an available silicon object SO_{dst} is selected in line (27), subject to the conditions of lines (28) and (29). Line (28) requires SO_{dst} to be compatible with v_{dst}, while line (29) applies to proximity constraint to SO_{src} and SO_{dst}, where SO_{src} is the silicon object assigned to the already placed node v_{src}. In line (30), the selected silicon object is assigned to v_{dst}, after which v_{dst}, and SO_{dst} are removed from *unplaced_node_set* and *available_silicon_object_set* in lines (31) and (32), respectively.

Apart from connected nodes, a TDFG may contain unconnected nodes, which are not considered in the procedure described above. In order to place these nodes, an iterative procedure is employed in lines (36) to (42). An unplaced node v_{rem} is selected in line (36) and is placed on a compatible silicon object, which is identified in lines (37) and (38). Since the node is not connected, no proximity constraints apply and the compatibility of the node and the silicon object alone is sufficient for placement of v_{rem} on SO_{avbl} in line (39). Line (40) removes v_{rem} from *unplaced_node_set*, and line (41) removes SO_{avbl} from *available_silicon_object_set* to avoid placing multiple nodes on the same object. This process is repeated until all remaining nodes have been placed.

To analyze the time complexity of Algorithm 5.1, we consider the *while* loop starting in line (6). This loop iterates $|E|$ times, where $|E|$ is the number of edges in the input TDFG. However, line (11) to (13) are dependent on the successful selection of SO_{src} and SO_{dst} such that conditions in lines (12) and (13) are satisfied. If no two slicon objects are available, such that these conditions are satisfied, then an alternative placement for the previously placed node pair is performed. In the worst case, each iteration of the *while* loop must be performed $|O_{FPOA}| \times |O_{FPOA}|$, where $|O_{FPOA}|$ denotes the number of silicon objects in an FPOA. Hence, the time compexity of the algorithm is $O((|O_{FPOA}| \times |O_{FPOA}|)^{|E|})$ and is exponential in the worst case.

5.4 Solving Placement Problem Using FD Constraints

Algorithm 5.1 outlines the steps involved in a delay aware placement of a TDFG on an FPOA. The placement process must not only ensure compatibility between a node and silicon object, but must also be cognizant of the permitted distance between locations of two connected nodes. As placement proceeds, the number of available silicon objects reduce, decreasing the choices for placing a node. Even more significant is the impact of proximity constraints on pairs of connected nodes. Suppose that a node v_0 connects directly to a set of distinct nodes $\{v_1, v_2, ...v_n\}$. After v_0 is placed, proximity constraints limit the number of potential placement choices for each node in the set. While Algorithm 5.1 enforces proximity constraints to ensure correct selection of a placement location, it does not benefit from the reduction in potential choices.

A more efficient approach is to formulate the algorithm as a set of two concurrent processes, where the first process assigns nodes to a silicon object based on compatibility, and the second process proactively applies proximity constraints, reducing the number of choices for placing nodes. The use of constraint solver facilitates an elegant encoding of the concurrent processes, and offers an efficient solution. In order to solve the problem using finite domain constraint satisfaction, we must translate the algorithm into a set of finite domain variables and constraints. In this research, Oz is employed to formulate the placement problem and an Oz implementation is described in the following sections.

5.4.1 FD Variables and Constraints

A TDFG, as obtained from a DFG, is initially represented as the data structure *TDFGNodeTuple*. However, as mentioned in sec. 5.2, the nodes in a TDFG may represent operations that are co-located, which are eventually placed on the same silicon object. Hence, all the nodes in *TDFGNodeTuple*, which share the same resource are collapsed into a single node to generate a new data structure called *TDFGNodeList*. The list of nodes, *TDFGNodeList*, is a list of tuple, one tuple per node, of the form [[$Node\_Num_1$, $Node\_Type_1$] [$Node\_Num_2$, $Node\_Type_2$] ...], where $Node\_Num$ is a unique number between 1 and N and is used for identifying a node. $Node\_Type$ indicates whether the node

is of type ALU, MAC, or RF.

Figure 5.6 gives an implementation of a procedure *PlacInitialize*. The inputs to *PlacInitialize* are the list of nodes, and the number of nodes, N, in *TDFGNodeList*. In line (2) a data structure called *NodeInitRecTuple* is created for storing placement information of all the nodes. *NodeInitRecTuple* consists of N fields, where each field holds a data structure containing placement information of a single node. Procedure *PlacInitialize* iterates over all the nodes in the input TDFG, operating on one element of *TDFGNodeList* during each iteration. Line (3) defines a loop construct which applies the procedure in line (4) to an element, *NodeElement*, of *TDFGNodeList*. Local variables *NodeNum*, *NodeType*, and *V_Rec* are declared in line (5) for use later in the procedure. In line (7), an FD data structure called *V_Rec* is created. This data structure, called a record, has four fields *v_id*, *v_res_type*, *v_x*, and *v_y*. Each field is a finite domain variable with domains indicated in lines (8) -

```
(1)  proc {PlacInitialize TDFGNodeList N ?NodeInitRecTuple}
(2)      NodeInitRecTuple = {Tuple.make nodeintrectup N}
(3)      {List.forAll TDFGNodeList
(4)          proc {$ NodeElement}
(5)              NodeNum NodeType V_Rec          %Local variables
(6)          in
(7)              V_Rec = {FD.record [v_id v_res_type v_x v_y] FD.sup}
(8)              V_Rec.v_id ::: 1#400
(9)              V_Rec.v_res_type ::: 0#2
(10)             V_Rec.v_x ::: 1#20
(11)             V_Rec.v_y ::: 1#20
(12)
(13)             V_Rec.v_id =: (V_Rec.v_y - 1) * 20 + V_Rec.v_x
(14)
(15)             NodeNum|NodeType|_ = NodeElement
(16)             V_Rec.v_res_type =: NodeType
(17)
(18)             {RemoveInvalidIdValuesFromDomain V_Rec.v_res_type V_Rec.v_id}
(19)
(20)             NodeInitRecTuple.NodeNum = V_Rec
(21)         end
(22)     }                                       %End of List.forAll
(23) end
```

Fig. 5.6: Creating and initializing finite domain variables for placement problem.

(11) and represent attributes v_{id}, v_{res_type}, v_x, and v_y of a node, respectively. Initially, the domain of v_id is the unique ids of all the silicon objects in an FPOA and ranges from 1 to 400. The domain of v_res_type is based on the following numerical encoding: 0 represents an ALU node, 1 represents a MAC node, and 2 represents an RF node.

Line (13) binds the coordinates (v_x, v_y) of a node v to its *id* attribute as defined in eq. (5.7). In Oz syntax, line (13) does not imply assignment, instead it should be interpreted as a relation among finite domain variables v_id, v_x, and v_y. If the domain of variables on either side of "=:" changes, so does the domain of variables on the opposite side. Statements like line (13) in particular make Oz very elegant for modeling search problems like placement, since the user specifies the relationship, and the solver figures out the binding of values to the constraint variables. Line (15) assigns the first field of *NodeElement* to variable *NodeNum*, and the second field to variable *NodeType*. Since the node type in a TDFG is fixed and known prior to placement, v_res_type is grounded by assigning the value of *NodeType* in line (16). Equations (5.4a-b) and eq. (5.6) suggest that as soon as v_res_type for a node v is known, the domain of v_{id} can be narrowed. Since the value of v_res_type is already known in line (16), line (18) invokes the procedure *RemoveInvalidIdValuesFromDomain* in order to shrink the domain of v_id. After processing all nodes, procedure *PlacInitialize* returns *NodeInitRecTuple* which contains records of all N nodes in the input TDFG.

Figure 5.7 shows the Oz implementation of procedure *RemoveInvalidIdValuesFromDomain*, which can be divided into three parts, one for each type of resource in an FPOA. Irrespective of the type of node, the v_id variables have a domain of 1#400, which suggests that a node can be placed on any object in the FPOA. However, a node cannot be placed on an incompatible object. This requires us to remove any invalid values in the domain of v_id which correspond to incompatible objects. Lines (2) to (4) declare symbolic constants for each type of node. Line (6) checks if the node type is ALU followed by line (7) which removes any unique id values corresponding to non-ALU objects from the domain of the node's v_id variable, where ALU_SOID is a list of elements in the set ALU_SOid of eq. (5.4a). Similarly, lines (9)-(11) and (12)-(14) use the same analogy to prune the domains of v_id

```
(1)   proc {RemoveInvalidIdValuesFromDomain v_res_type v_id}
(2)       ALU = 0
(3)       MAC = 1
(4)       RF = 2
(5)   in
(6)       if (v_res_type == ALU) then
(7)           v_id ::: ALU_SOID                    %SO_id of ALU objects
(8)       end
(9)       if (v_res_type == MAC) then
(10)          v_id ::: MAC_SOID                    %SO_id of MAC objects
(11)      end
(12)      if (v_res_type == RF) then
(13)          v_id ::: RF_SOID                     %SO_id of RF objects
(14)      end
(15)  end
```

Fig. 5.7: Narrowing domains by removing invalid unique identifier values.

variables for MAC and RF type nodes, respectively, where *MAC_SOID* and *RF_SOID* are lists of elements in their respective sets, *MAC_SOid* and *RF_SOid*, defined in eq. (5.4a-b).

No two nodes in a TDFG can occupy the same silicon object. This restriction is implemented in the Oz procedure *MakeVidDistinct*, shown in fig. 5.8. Procedure *MakeVidDistinct* builds a list of *v_id* variables for all the nodes in a TDFG and imposes a constraint on all the members of this list, requiring them to have distinct values. The output of procedure *PlacInitialize*, *NodeInitRecTuple*, is passed as the input to procedure *MakeVidDistinct*. Variable *AllVidList* is declared in line (2) and represents an Oz data structure called List.

```
(1)   proc {MakeVidDistinct NodeInitRecTuple}
(2)       AllVidList                          %Local variable
(3)   in
(4)       AllVidList = {Record.foldL NodeInitRecTuple
(5)           proc {$ Accum NodeRecElem ?Output}
(6)                 Output = {List.append Accum NodeRecElem.v_id}
(7)           end
(8)       nil}                                %End of Record.foldL
(9)       {FD.distinct AllVidList}
(10)  end
```

Fig. 5.8: Imposing distinct constraint on *v_id* FD variables.

A folding operation is applied to *NodeInitRecTuple* in lines (4) to (8), which populates list *AllVidList* with *v_id* variables. A built-in constraint, FD.distinct, is applied to *AllVidList* in line (9). FD.distinct constraint creates a propagator which does not allow any two variables in *AllVidList* to have identical values, ensuring that no two nodes are placed on the same silicon object.

A valid placement must conform to proximity constraints that originate from communication delays between nodes in a TDFG. Figure 5.9 illustrates a procedure *ApplyProximityConstraints* that operates upon *NodeInitRecTuple* and a list of all the edges, *TDFGEdgeList*, in the input TDFG. Members of *TDFGEdgeList* have the following format: [< Src_Node_{number}, Dst_Node_{number}, $Delay$ >], where Src_Node_{number} and Dst_Node_{number} are the unique numbers assigned to source and destination nodes of an edge in the TDFG, and $Delay$ is the communication delay along this edge. *ApplyProximityConstraints* iteratively applies proximity constraints to source-destination node pairs. At the beginning of each iteration, a new member of *TDFGEdgeList* is passed to the procedure as parameter *EgdeElem* in line (3). Local variables used by the anonymous procedure of line (3) are declared in line (4). Variables *X_dist* and *Y_dist* are defined as FD variables and are later used for holding absolute values of horizontal and vertical distance between two nodes, respectively. Line (8) dissects *EdgeElem* to extract the three fields of *EdgeElem*, and respectively assigns them to *V_src*, *V_dst*, and *Delay*. Values of *V_src* and *V_dst* are used to access the records of their respective nodes in *NodeInitRecTuple* data structure. Variables representing the coordinates of these two nodes are obtained from *NodeInitRecTuple* in lines (9) to (12). These coordinates facilitate Manhattan distance calculation between the source and destination node. Line (14) determines the horizontal distance between the pair of nodes and assigns it to *X_dist*. Similarly, vertical distance between the pair of nodes is computed in line (15) and is assigned to *Y_dist*. It should be noted that the exact locations of these nodes may not be known when calculating distances in lines (14) and (15), which poses a challenge when finding the absolute difference between two ungrounded FD variables because conditional statements such as if-then-else is not permitted.

```
(1)    proc {ApplyProximityConstraints NodeInitRecTuple TDFGEdgeList}
(2)        {List.forAll TDFGEdgeList
(3)            proc {$ EdgeElem}
(4)                X_dist Y_dist Delay V_src V_dst V_src_x V_src_y V_dst_x V_dst_y
(5)            in
(6)                X_dist = {FD.int 0#20}
(7)                Y_dist = {FD.int 0#20}
(8)                V_src|V_dst|Delay|_ = EdgeElem
(9)                V_src_x = NodeInitRecTuple.V_src.v_x
(10)               V_src_y = NodeInitRecTuple.V_src.v_y
(11)               V_dst_x = NodeInitRecTuple.V_dst.v_x
(12)               V_dst_y = NodeInitRecTuple.V_dst.v_y
(13)
(14)               X_dist =: (V_src_x >=: V_dst_x) * (V_src_x - V_dst_x)
(14a)                  + (V_src_x <: V_dst_x) * (V_dst_x - V_src_x)
(15)               Y_dist =: (V_src_y >=: V_dst_y) * (V_src_y - V_dst_y)
(15a)                  + (V_src_y <: V_dst_y) * (V_dst_y - V_src_y)
(16)
(17)               if (Delay == 0) then
(18)                   X_dist =<: 1
(19)                   Y_dist =<: 1
(20)               else
(21)                   X_dist + Y_dist =<: Delay * 4
(22)               end
(23)           end
(24)       }
(25)   end
```

Fig. 5.9: Proximity constraints.

Specifying distance relations using reified constraints allows us to restrict the truth value of a constraint to 0 or 1, and offers an elegant method to calculate absolute difference. For example, in line (14), X_dist is composed of two parts: $(V\_src\_x >=: V\_dst\_x) \times (V\_src\_x - V\_dst\_x)$ and $(V\_src\_x <: V\_dst\_x) \times (V\_dst\_x - V\_src\_x)$. For any integer assignment, either $(V\_src\_x >=: V\_dst\_x) = 1$ or $(V\_src\_x <: V\_dst\_x) = 1$, making only one part of the expression contribute to X_dist, and thereby allowing us to compute absolute difference. If upon evaluation $(V\_src\_x >=: V\_dst\_x) = 0$, then $(V\_src\_x - V\_dst\_x)$ is ignored.

Lines (18) and (19) apply to the case with zero-delay communication between a node

pair and requires that the nodes be placed as nearest neighbors. The statements in lines (18) and(19) restrict the maximum horizontal and vertical distance between the node pair to one hop, imposing nearest neighbor proximity. It should be noted that $X\_dist = Y\_dist = 0$ is not possible because procedure *MakeVidDistinct* does not allow two different nodes to be co-located. On the other hand, if two nodes must use PL communication, then line (21) transforms eq. (5.10) into a suitable distance constraint. Lines (18), (19), and (21) restrict the domains of X_dist and Y_dist, which in turn shrinks the domain of each node's coordinates, pruning the domain of a node's v_id variable. Thus, procedure *ApplyProximityConstraints* makes the placement process cognizant of the communication delay between two nodes by permitting placement on only those locations which can satisfy the delay requirement.

As mentioned earlier, eq. (5.1) uses row major order as against column major order for assigning unique ids to objects. The advantages of using row major order are observed in procedure *ApplyProximityConstraints* for ALU type operations. Lines (14) to (22) of this procedure define a region which contains both V_src and V_dst. However, the proposed placement methodology employs interval propagation (see Chapter 3), which defines the domains of v_id variables of these nodes as a range of numbers. This range is specified as a lower bound and an upper bound of silicon object ids that can be assigned to the v_id variable. The lower bound is the smallest silicon object id in the region and the upper bound is the largest silicon object id in the region. If column major order is used, all the objects in the columns between these bounds are included in v_id variable's domain. Similarly, if row major order is used, all objects in the rows between these bounds become part of the range. The number of ALU objects in most rows is 12, but varies between 2 and 20 for columns. Experiments indicate that using row major order typically results in smaller v_id variable domains as compared to column major order. Moreover, unlike column major order, row major order avoids placing two ALUs in adjacent locations unless they are NN. This is because the least value in the domain is assigned to v_id variable first. If the assignment does not yield a valid solution, the lower bound is increased to the next higher value. Since

the next higher silicon object id value in row major order is usually of a non-djacent ALU, adjacent ALUs are typically not assigned to nodes connected by non-NN connection.

5.4.2 Improving Propagation

Propagation attempts to narrow down domains of variables occurring in a constraint. This narrowing down is referred to as constraint propagation. Effective propagation requires stronger propagators, that are able to identify and remove invalid values from a variables domain, as soon as possible. Stronger propagation reduces the search space, decreases the search time, and improves search convergence. The following methods are used in this research for improving propagation.

Additional Proximity Constraints

Proximity constraints discussed so far operate on only those nodes that share an edge. When two nodes are connected via a chain of nodes in between, the effect of grounding a node at one end of this chain may take some time to propagate to the node at the opposite end. Through experiments it was found that by introducing virtual connections between two such nodes, a stronger propagation can be obtained at the cost of increased computation burden. However, an improvement in search convergence is noticeable only when a connection chain consists of only zero delay edges. It should be noted that these virtual connections do not represent a communication route between two nodes.

In fig. 5.10, nodes v_2 and v_5 are connected via nodes v_1, v_4 and edges e_0, e_1, and e_3, each with zero communication delay. Since there are $n_e = 3$ edges, the maximum post-placement horizontal or vertical distance between v_2 and v_5 can be three hops. Equations (5.11a-b) gives the maximum horizontal and vertical distances between two nodes, v_i and v_j, connected via n_e number of edges, each having zero delay.

$$|v_{i_x} - v_{j_x}| \leq n_e \tag{5.11a}$$

$$|v_{i_y} - v_{j_y}| \leq n_e \tag{5.11b}$$

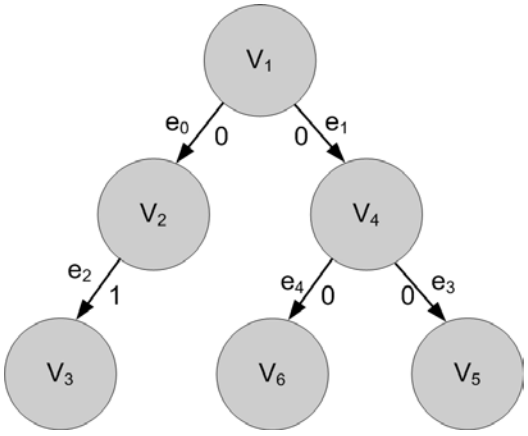


Fig. 5.10: Additional proximity constraints.

Figure 5.11 presents the Oz implementation for imposing these additional proximity constraints. Procedure *AdditionalProximityConstraints* borrows the Manhattan distance calculation concept from procedure *ApplyProximityConstraints* and translates eq. (5.11a-b) into a set of FD constrants given in lines (11) and (12).

```
(1)    proc {AdditionalProximityConstraints V_src V_dst N_e}
(2)        X_dist Y_dist V_src_x V_src_y V_dst_x V_dst_y
(3)    in
(4)        X_dist = {FD.int 0#20}
(5)        Y_dist = {FD.int 0#20}
(6)        V_src_x = NodeInitRecTuple.V_src.v_x
(7)        V_src_y = NodeInitRecTuple.V_src.v_y
(8)        V_dst_x = NodeInitRecTuple.V_dst.v_x
(9)        V_dst_y = NodeInitRecTuple.V_dst.v_y
(10)
(11)       X_dist =: (V_src_x >=: V_dst_x) * (V_src_x - V_dst_x)
(11a)          + (V_src_x <: V_dst_x) * (V_dst_x - V_src_x)
(12)       Y_dist =: (V_src_y >=: V_dst_y) * (V_src_y - V_dst_y)
(12a)          + (V_src_y <: V_dst_y) * (V_dst_y - V_src_y)
(13)       X_dist =<: N_e
(14)       Y_dist =<: N_e
(15)   end
```

Fig. 5.11: Oz implementation for introducing additional proximity constraints.

Reducing Search Space using Bounding Box

The FPOA grid consists of four identical 5×20 sized blocks. Each of these blocks can be subdivided into two identical 5×10 blocks. A 5×10 block consists of two non-identical 5×5 blocks having 17 ALU, 5 RF, 3 MAC units and 15 ALU, 5 RF, 5 MAC units, respectively. We exploit the symmetry of the FPOA grid and define a bounding box with one corner at location (1,1) and opposite corner at (B_x, B_y), as shown in fig. 5.12. The size of this bounding box can be changed by modifying (B_x, B_y) and is incremented in multiples of 5×5, implying that $B_x, B_y \in \{5,\ 10,\ 15,\ 20\}$. Equation (5.12) gives the number of silicon objects of each type contained in the region, which is defined by diagonally opposite coordinates (1,1) and (B_x, B_y).

$$Number\ of\ ALUs = \begin{cases} (B_x \times (B_y \times 3 + 2))/5 & ,if\ (B_y \leq 15) \\ (B_x/5 \times 64) & ,\text{otherwise} \end{cases} \tag{5.12a}$$

$$Number\ of\ MACs = \begin{cases} (B_x \times (B_y - 2))/5 & ,if\ (B_y \leq 15) \\ (B_x/5 \times 16) & ,\text{otherwise} \end{cases} \tag{5.12b}$$

$$Number\ of\ RFs = (B_x \times B_y)/5 \tag{5.12c}$$

An Oz implementation for specifying a bounding box is given in fig. 5.13. B_x and B_y are denoted by FD variables X_limit and Y_limit, respectively, and are declared in lines (4)

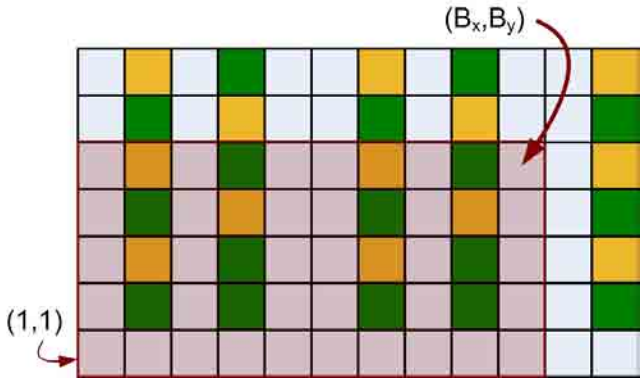


Fig. 5.12: Limiting search area using a bounding box.

```
(1)    proc {BoundingBox NodeInitRecTuple TDFG_ALU_Count TDFG_MAC_Count
(1a)       TDFG_RF_Count}
(2)        X_limit Y_limit                              %Local variables
(3)    in
(4)        X_limit = {FD.int 1#20}
(5)        Y_limit = {FD.int 1#20}
(6)        X_limit ::: [5 10 15 20]
(7)        Y_limit ::: [5 10 15 20]
(8)
(9)        (X_limit >=: Y_limit) * (X_limit - Y_limit)
(9a)           + (X_limit <: Y_limit) * (Y_limit - X_limit)=<: 5
(10)       TDFG_ALU_Count =: (Y_limit =<: 15) * (X_limit * (Y_limit * 3 + 2))/5
(10a)          + (Y_limit =: 20) * (X_limit/5 * 64)
(11)       TDFG_MAC_Count =: (Y_limit =<: 15) * (X_limit * (Y_limit - 2))/5
(11a)          + (Y_limit =: 20) * (X_limit/5 * 16)
(12)       TDFG_RF_Count =: (X_limit * Y_limit) / 5
(13)
(14)       {Record.forAll NodeInitRecTuple
(15)           proc {$ NodeRecElem}
(16)               NodeRecElem.v_x =<: X_limit      %Upper bound on X co-ordinate
(17)               NodeRecElem.v_y =<: Y_limit      %Upper bound on Y co-ordinate
(18)           end
(19)       }                                        %End of Record.forAll
(20)   end
```

Fig. 5.13: Using a bounding box to reduce search space.

and (5). Lines (6) and (7) narrow the domains of these variables, which are now permitted to only take the following values: 5, 10, 15, and 20. It was empirically determined that $|B_x - B_y| \leq 5$ improves search convergence, and is implemented as a constraint in line (9). Finally, lines (14) and (19) impose these bounds iteratively on the coordinates of all nodes in the TDFG .

The initial size of the bounding box must be large enough to accommodate all the nodes that are present in the input TDFG. If variables $TDFG\_ALU\_Count$, $TDFG\_MAC\_Count$, and $TDFG\_RF\_Count$ denote the number of ALU, MAC, and RF resources in a input TDFG, then lines (10), (11), and (12) guarantee that the bounding box specifies an area large enough to have sufficient resource for placing the input TDFG design. Thus, a lower bound for X_limit and Y_limit is established as per eq. (5.12) before the search begins.

During the search, if no solution is found, then the bound is increased in increments of 5×5 until a solution is found or it is determined that no solution exists. Any increment is first made in horizontal direction because it leads to an increased number of ALUs and experiments have shown that it improves convergence.

Divide and Conquer

In order to improve performance for placement for large designs, a divide and conquer approach is also supported by the placement tool. A design can be subdivided into two or more modules, and each of them can be placed one at a time. This feature significantly improves placement performance since it makes better use of a bounding box than a single large design. If a design uses the entire chip area, the bounding box is set equal to the chip area, which negates the purpose of a bounding box. Instead, initially, if only half of the design is placed, then the bounding box restricts the search to half of the chip, improving search time and reducing memory requirements. The partial placement further reduces the number of available objects on the FPOA, thereby shrinking the domains of all unplaced variables. This approach provides benefits similar to the application of a bounding box, and helps in faster convergence of a placement problem.

Additionally, if a portion of the design has already been placed, the user can incrementally place the remaining part rather than repeating the entire placement process. However, the addition of new modules and rescheduling can make a previous placement invalid. In such a case, the placement tool backtracks to undo any invalid placement and subsequently resumes search for a valid placement.

5.4.3 Distribution Strategy

A key to efficient constraint solving is the customization of variable selection strategy, based on the knowledge of the problem domain. While many selection strategies are possible, some provide better convergence than others by dramatically reducing the size of the search tree. To solve the placement problem, we employ the empirically derived variable selection strategy depicted in fig. 5.14.

Since bounding box defines a region of interest which is typically smaller than the entire FPOA, it has the most direct impact on shrinking the design space. Hence, distribution begins by first selecting values for B_x and B_y. A standard first fail strategy is used which always selects the least values in the domains of B_x and B_y, which implies that the distributor always selects the smallest possible bounding box.

Once a bounding box has been established, distribution continues with ungrounded variables belonging to nodes of resource type MAC or RF. This is because MACs and RFs are fewer in number than ALUs and a smaller number of distribution steps are required to ground these variables. A split domain distribution strategy, described in Chapter 3, is used for these variables because it leads to smaller search trees. This strategy splits the domain of a variable and tries the lower part of the domain first. Finally, any variables associated with nodes belonging to resource type ALU are selected for distribution. Once again, a split domain distribution strategy is used. If a speculated value results in a constraint violation, then the search backtracks to the immediately preceding stable state and distribution continues by making a different guess.

The separation of ALU type nodes from MAC and RF type nodes may seem unnecessary because nodes corresponding to MACs and RFs have smaller domains, and first fail strategy always selects the variable with the smallest domain size. In practice, propagation

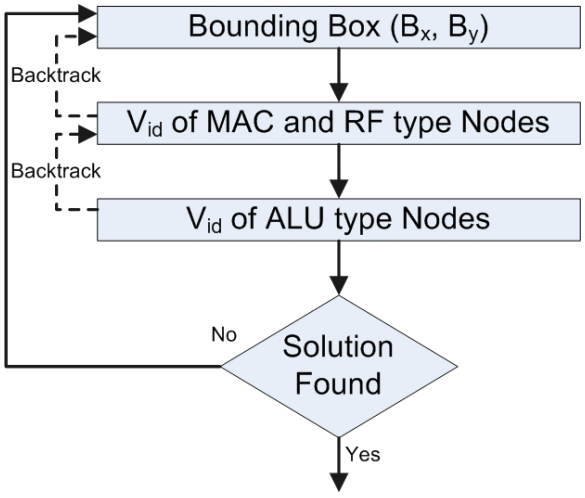


Fig. 5.14: Distribution strategy for placement.

shrinks the domains of variables associated with ALU type nodes as distribution proceeds with MAC and RF type nodes. This results in a scenario where the domains of variables corresponding to ALU type nodes have smaller domains than MAC and RF type nodes causing the first fail strategy to distribute on variables associated with ALUs. Typically, this results in the placement of ALU nodes at locations which do not lead to a solution and requires extensive backtracking. Hence, ALU nodes are handled only after all MAC and RF nodes have been placed.

5.5 Placement Summary

As explained in this chapter, a formal model for depicting the computational elements of an FPOA is successfully developed. An interconnect framework supports two different types of networks to enable communication between them, namely NN and PL. The communication delay between two connected objects depends on the type of network used, and is a function of the distance between the objects when communication happens over PL network. The placement problem is formulated by specifying a set of mapping rules, defined over the node attributes and properties, such as the location and type, of silicon objects. The delay between two nodes is translated into a distance requirement between the silicon objects executing these nodes. A successful placement not only ensures node and silicon object compatibility, but also satisfies the distance requirements imposed by the design specification. An algorithm is proposed as a solution to the placement problem, and is represented as two concurrent processes, where one process assigns a compatible silicon object to a node, while the other process ensures that no distance requirement is violated. These processes are translated into a finite domain constraint representation, which is issued to the Mozart constraint solver. Additional strategies for improving propagation, distribution, and search convergence have also been proposed and implemented.

Chapter 6

Routing

Chapter 3 described two types of communication mechanisms available in an FPOA: NN and PL communication. Delay aware placement discussed in the previous chapter ensures that objects using zero-delay paths are placed as nearest neighbors, allowing them to use NN communication. However, non-zero delay paths must be implemented using PL interconnects. This chapter describes a finite domain constraint satisfaction-based routing methodology that has been developed to route non-zero delay paths using PL communication.

Section 6.1 discusses a mathematical model of FPOA routing resources proposed in this research. Using this model, the FPOA routing problem is described in sec. 6.2. A routing algorithm is presented in sec. 6.3, followed by a discussion in sec. 6.4 on translating and solving this problem using FD constraint satisfaction.

6.1 Mathematical Model of Routing Resources

Chapter 5 presents a mathematical model which provides the foundation for uniquely representing silicon objects on an FPOA as a function of their Cartesian coordinates. Similarly, it is possible to specify a formal mapping between routing resources and their location on an FPOA. This extended model is necessary to facilitate the development of a routing algorithm for the FPOA architecture.

As mentioned in Chapter 3, Party Lines are used by objects for communicating over long distances. There are three party line groups with channels traveling in all four directions where direction of travel is denoted by $PL\_Channel_{direction} \in \{N, S, E, W\}$. An instance of a party line group is denoted by PL_group, where $PL\_group \in \{1, 2, 3\}$. Each silicon object has five LL registers that are used for PL communication: two in PL groups 1, 2,

and one in PL group 3.

A register is characterized by three attributes: the location of a register Reg_{loc}, PL group Reg_{PL_group}, and orientation $Reg_{orientation}$. The location of a register on the chip is the same as the location of its parent silicon object. As mentioned in sec. 5.1, the unique id of a silicon object is sufficient to locate an object on the chip. Consequently, location of a register is denoted by the unique id of its parent silicon object SO, or $Reg_{loc} = ID(SO)$. We define $RegLoc : LL_{FPOA} \rightarrow N^{+}$ to map the set of registers on an FPOA, LL_{FPOA}, to their unique id, where $RegLoc(\ell) = ID(SO)$, if SO is the parent silicon object of $\ell \in LL_{FPOA}$. $Reg_{PL\_group} \in \{1, 2, 3\}$ represents the PL group of a particular register. The function $RegPLG : LL_{FPOA} \rightarrow \{1,\ 2,\ 3\}$ maps a register to its PL group. Each register within a PL group is shared by PL channels traveling in opposite directions. The orientation of an LL register depends on the direction of the pair of channels it serves. Figure 6.1 illustrates the organization of LL registers and associated multiplexers inside a silicon object for PL group 1. The North-South LL register serves the North and South PL channels, whereas the East-West LL register serves the East and West PL channels. Equation (6.1) provides a numerical encoding for $Reg_{orientation}$ based on the PL channels sharing the LL register

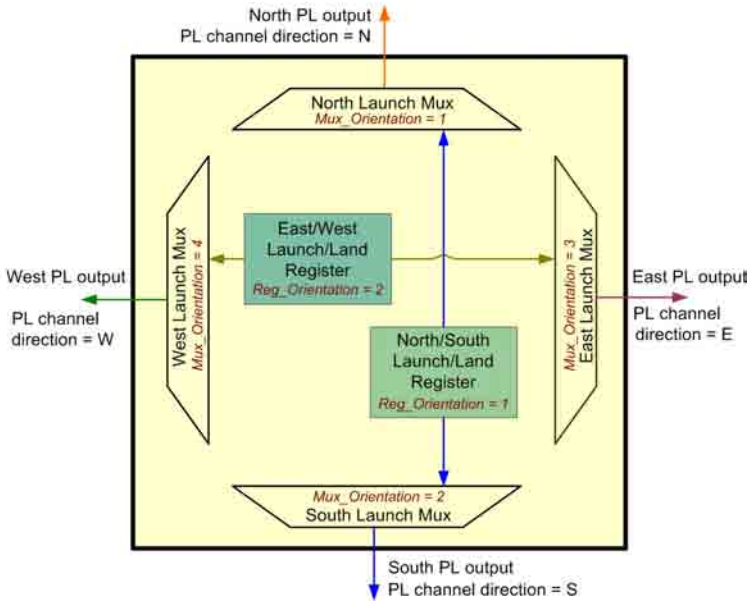


Fig. 6.1: Register and multiplexer orientations.

and the function $RegOrient : LL_{FPOA} \rightarrow \{1,\ 2\}$ represents the orientation of a register $\ell \in LL_{FPOA}$.

$$Reg_{orientation} = \begin{cases} 1, & if\ PL\_Channel_{direction}\ \in \{N, S\} \\ 2, & if\ PL\_Channel_{direction}\ \in \{E, W\} \end{cases} \tag{6.1}$$

In order to identify individual registers, a unique id is assigned to each LL register. Since there are 2000 LL registers in an FPOA, the value of this unique id lies in the interval $(1, 2000)$. The function $RegID : LL_{FPOA} \rightarrow N^+$ maps a register to its unique id. Given the orientation, PL group, and parent silicon object, SO, of an LL register $\ell \in LL_{FPOA}$ is calculated as shown in eq. (6.2).

$$RegID(\ell) = (RegLoc(\ell) - 1) \times 5 + (RegPLG(\ell) - 1) \times 2 + RegOrient(\ell) \tag{6.2}$$

In addition to LL registers, every silicon object also contains 10 multiplexers: four each in PL groups 1 and 2, and two in PL group 3. Unlike LL registers, multiplexers are not shared by pair of PL channels traveling in opposite directions. Instead, each channel has a dedicated multiplexer in its direction of travel. Multiplexers, like registers, are also categorized by three attributes: location Mux_{loc}, PL group Mux_{PL_group}, and orientation $Mux_{orientation}$. The location of a multiplexer is the same as the location of its parent silicon object SO, or $Mux_{loc} = ID(SO)$. $Mux_{PL\_group} \in \{1, 2, 3\}$ represents the PL group of a particular multiplexer. Equation (6.3) gives the numerical encoding for the orientation of a multiplexer, which is decided by the direction of travel of its PL channel, as shown in fig. 6.1.

To simplify the notation, we define the following functions to represent the location, Pl group, and orientation of a multiplexer $mx \in M_{FPOA}$, where M_{FPOA} denotes the set of all 4000 multiplexers in an FPOA. The first function is $MuxLoc : M_{FPOA} \rightarrow N^+$ which maps a multiplexer to its location. If the parent silicon object of $mx \in M_{FPOA}$ is SO, then $MuxLoc(mx) = ID(SO)$. The second function, $MuxOrient : M_{FPOA} \rightarrow \{1..4\}$, gives the orientation of a multiplexer, while $MuxPLG : M_{FPOA} \rightarrow \{1,\ 2,\ 3\}$ maps a multiplexer to

its PL group. If the location, PL group, and orientation of a multiplexer, $mx \in M_{FPOA}$, are known, it can be assigned a unique id given by eq. (6.4), where $MuxID : M_{FPOA} \rightarrow N^{+}$ denotes the unique id of mx.

$$Mux_{orientation} = \begin{cases} 1, & if\ PL\_Channel_{direction} = N \\ 2, & if\ PL\_Channel_{direction} = S \\ 3, & if\ PL\_Channel_{direction} = E \\ 4, & if\ PL\_Channel_{direction} = W \end{cases} \tag{6.3}$$

$$MuxID(mx) = (MuxLoc(mx) \times 10 + (MuxPLG(mx) - 1) \times 4 + MuxOrient(mx) \tag{6.4}$$

Equations 6.2 and 6.4 provide a mechanism to reference a specific register or a multiplexer using a unique id or a combination of their location, PL group, and orientation. While it is possible to formulate the routing problem solely using a unique identifier for the routing resources, this approach is inefficient since it requires iteratively considering each routing resources. Instead, the knowledge of location, PL group, and orientation helps in narrowing down the potential candidates to route a path, offering an efficient and elegant method to approach the FPOA routing problem.

6.2 Routing Problem

Placement assigns a silicon object to each node in the input TDFG. Since placement is delay aware, it guarantees that no two nodes are placed farther away than permitted by communication delay between them. To enable communication between a pair of nodes, the edge between these nodes must be realized as a physical path connecting the corresponding silicon objects. A valid connection is established by allocating sufficient routing resources, such that the delay requirements along the edge is satisfied by the path. Routing refers to the process of allocating routing resources to each edge in the TDFG to implement physical paths connecting two objects, such that no two paths share resources and all connections

are valid, i.e., all delay requirements are met.

Figure 6.2 shows two silicon objects SO_{src} and SO_{dst} that must be connected by a path $p = (SO_{src}, SO_{dst})$ having delay n. Signals originating from the source silicon object, SO_{Src}, must arrive at destination object SO_{Dst} exactly n cycles later. When using PL communication, a signal can travel up to a distance of four hops in one clock cycle, after which it must land on a LL register. Thus, if a signal must be delayed by n clock cycles, it must land on n LL registers along the path from source to destination. Since a signal always lands on an LL register at the destination silicon object, a path requires $n-1$ intermediate registers as shown in the figure. Any two consecutive LL registers represent a path segment whose length is at least one hop and at most four hops. Up to four multiplexers comprise a path segment with the first multiplexer always residing on the silicon object as the segment's starting LL register. For example, fig. 6.2 shows a segment between registers Reg_1 and Reg_2 consisting of multiplexers Mux_0 to Mux_3. The first multiplexer, Mux_0, is co-located with Reg_1 and connects Reg_1 to Mux_1. Depending on the distance between the silicon objects of these registers, Mux_1 to Mux_3 may not be required. Furthermore, the first segment of a path originates from SO_{Src} and hence Mux_0 for the first segment of any path is always located on its source silicon object.

Figure 6.3 presents all four possible scenarios based on length of a path segment. In fig. 6.3(a) and 6.3(b), four and three multiplexers are needed, respectively. For shorter

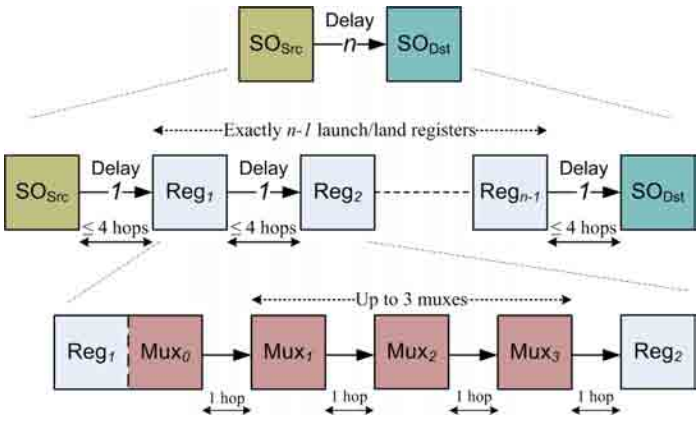


Fig. 6.2: Routing a path with delay n.

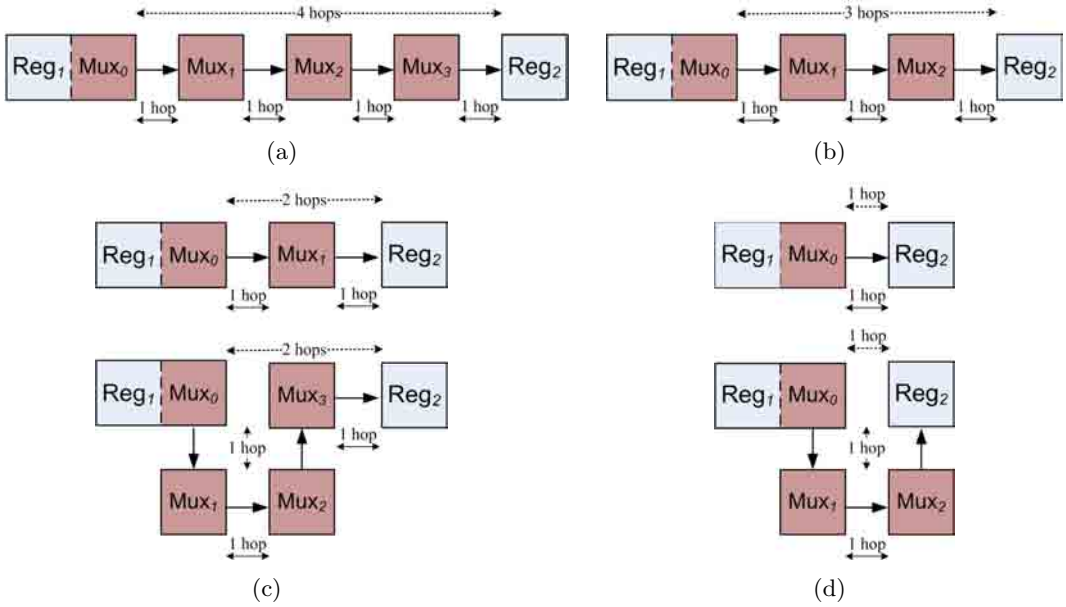


Fig. 6.3: Routing paths for (a) four hops, (b) three hops, (c) two hops, and (d) one hop long path segments. Possible alternative paths are also shown in (c) and (d).

segments, more than the minimum required number of multiplexers may occasionally be used to implement the segment. Though using more multiplexers than required may seem inefficient, this feature is helpful when a shorter route is not possible due to unavailability of resources. Figures 6.3(c) and 6.3(d) show only two of the many possible alternatives for two hops and one hop long path segments.

Routing can be viewed as a two step process for finding a path with delay n between two silicon objects:

1. Find $n-1$ unique intermediate LL registers such that no two successive registers are more than four hops apart. According to eq. (6.2), a unique LL register is characterized by three attributes: location or parent silicon object id, PL group, and register orientation. Determining these attributes for each of the register is sufficient to establish a coarse route.

2. For each path segment, allocate unique multiplexers to establish a connection between two consecutive registers. The function $MuxID()$, defined in eq. (6.4), is sufficient to

uniquely identify a multiplexer but requires knowledge of three attributes: location or parent silicon object id, PL group, and multiplexer orientation.

Since the number of registers is far less than the number of multiplexers, a coarse route consisting only of LL registers can be quickly established. This further narrows down the potential multiplexer candidates required to complete the n path segments. Moreover, the number of multiplexers required in each path segment is dependent on the locations of LL registers. Hence, multiplexers are allocated only after the LL registers have been identified.

6.2.1 Register and Multiplexer Location

The location of an LL register or a multiplexer is the same as the location of their parent silicon object. Location is represented using Cartesian coordinates and can be obtained from an object's unique id. Hence, instead of finding the exact coordinates of a parent silicon object, determining the unique id of an object is sufficient to locate it.

Figure 6.4 shows an intermediate register, Reg_i, along a path with delay n. Signals from SO_{src} arrive at Reg_i after p clock cycles, after which they take q clock cycles to reach SO_{dst}, where $p+q=n$. All possible locations for the parent silicon object of Reg_i must be located at most $4p$ and $4q$ hops away from SO_{src} and SO_{dst}, respectively. If SO_Reg_i is the parent silicon object of Reg_i, then $RegLoc(Reg_i) = ID(SO\_Reg_i)$ the potential locations of Reg_i must satisfy eq. (6.5), where $Dist()$ is defined in eq. (5.5).

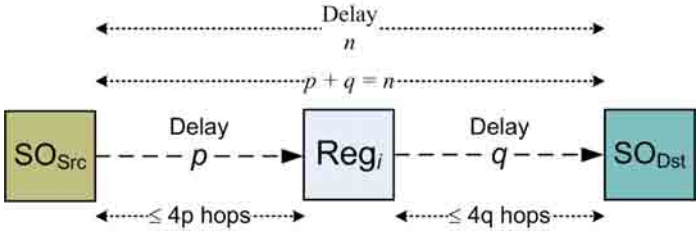


Fig. 6.4: A launch and land register i along a path with delay n.

$$Dist(SO_{src}, SO\_Reg_i) \leq 4p \ (hops) \tag{6.5a}$$

$$Dist(SO\_Reg_i, SO_{dst}) \leq 4q \ (hops) \tag{6.5b}$$

There are up to four multiplexers in any path segment. As mentioned earlier, the first multiplexer is always located on the source silicon object of the path segment, and hence its location is known as soon as all LL register locations are determined. However, multiplexers are handled differently than registers because the number of multiplexers in a path segment is not fixed. Each path segment has at least one and at most four multiplexers. To simplify the problem definition, the following is assumed without loss of generality.

1. A multiplexer in a path segment is denoted by Mux_i, where $0 \leq i \leq 3$.
2. Mux_0 is the first multiplexer and resides on the source silicon object of a path segment.
3. A path segment is implemented by connecting every multiplexer pair, Mux_i and Mux_{i+1}, as shown in fig. 6.2, where $0 \leq i < 3$.
4. If k multiplexers implement a path segment, then Mux_{k-1} is the last multiplexer in the segment as shown in fig. 6.3, where $1 \leq k \leq 4$.
5. SO_Mux_i is the parent silicon object of Mux_i.

Equation (6.6) imposes a distance restriction on multiplexers in a path segment. A connected multiplexer pair must occupy adjacent locations on an FPOA. Potential locations for all multiplexers in a path are determined by applying eq. (6.6) to all segments of that path.

$$\forall i \geq 0 \ Dist(MuxLoc(Mux_i), MuxLoc(Mux_{(i+1)}) = 1 \tag{6.6}$$

6.2.2 Party Line Groups

The FPOA architecture contains 2000 LL registers from which $n-1$ unique registers can be selected to route a path. The same path can be routed in multiple ways by choosing different permutations of registers. However, all the registers in a path must belong to the same PL group. Similarly, all multiplexers allocated to a path must also have the same PL group. Based on the previous discussion, a path can be represented as sequence of registers and multiplexers, or path $p = [< mux_0, mux_1, mux_2, mux_3, reg_1 >, < mux_4, mux_5, mux_6, mux_7, reg_2 >, ...reg_n]$. Therefore, all registers Reg_i and Reg_j in a delay n path must have the same PL group as shown in eq. (6.7).

$$\forall i, j \in \{1..n\},\ RegPLG(Reg_i) = RegPLG(Reg_j) \tag{6.7}$$

Similarly, all multiplexers allocated to a delay n path have the same PL group as specified by eq. (6.8), where Mux_l and Mux_m belong to the same path. Furthermore, if Reg_i and Mux_l belong to the same path, then $RegPLG(Reg_i) = MuxPLG(Mux_l)$.

$$\forall l, m \in \{0..(4n-1)\},\ MuxPLG(Mux_l) = MuxPLG(Mux_m) \tag{6.8}$$

All PL groups except PL group 3 have channels traveling in N, S, E, and W directions. PL group 3 contains channels traveling in N and S directions only. A necessary condition for a path using PL group 3 is that the entire path must be contained within the same vertical column on an FPOA. Thus, the source and destination objects, any intermediate registers, and all multiplexers in the path must be in the same column and no change in direction of travel is permitted.

6.2.3 Register and Multiplexer Orientation

Orientations of LL registers and multiplexers are decided by their PL group and relative locations in a path. The impact of a PL group is limited to restricting the orientation of a register and a multiplexer to North and South directions only, when PL group 3 is selected.

Relative locations of adjacent multiplexer-multiplexer or multiplexer-register pairs play a significant role in finalizing orientations and give rise to the following three scenarios.

1. Multiplexer-Multiplexer connection: For a valid path, a connected multiplexer pair, Mux_j and Mux_{j+1}, must reside at adjacent locations. The direction of Mux_j's PL channel is a function of $MuxLoc(Mux_j)$ and $MuxLoc(Mux_{(j+1)})$ and is equal to the relative position of Mux_{j+1}, with respect to Mux_j. Based on fig. 6.5 and eq. (5.1), eq. (6.9) defines a relation between multiplexer locations and PL channel direction. $PL\_Channel_{direction}$ obtained from eq. (6.9) is used in eq. (6.3) to determine the orientation of Mux_j. Due to the encoding used for denoting location ids, the id of $Mux_{(j+1)}$ is offset by $+XMax$, if it is to the North of Mux_j or by $-XMax$, if it is to the South of Mux_j. Similarly, $Mux_{(j+1)}$'s id is obtained by adding 1 to Mux_j's id, if it is to the East of Mux_j, or by subtracting 1 if it is to the West.

$$PL\_Channel_{direction} = \begin{cases} N, & if\ MuxLoc(Mux_{(j+1)}) = MuxLoc(Mux_j) + XMax \\ S, & if\ MuxLoc(Mux_{(j+1)}) = MuxLoc(Mux_j) - XMax \\ E, & if\ MuxLoc(Mux_{(j+1)}) = MuxLoc(Mux_j) + 1 \\ W, & if\ MuxLoc(Mux_{(j+1)}) = MuxLoc(Mux_j) - 1 \end{cases} \tag{6.9}$$

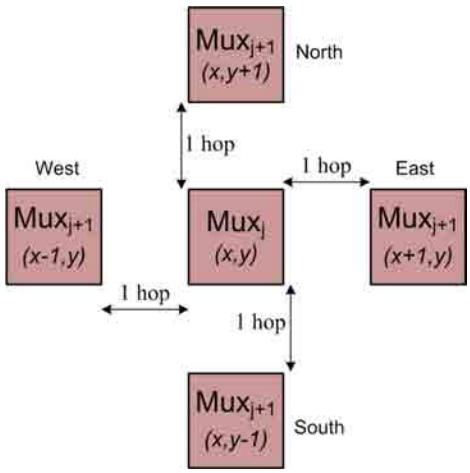


Fig. 6.5: Possible locations of adjacent multiplexers in a connected multiplexer pair.

2. LL Register-Multiplexer connection: A signal launched from an LL register is sent through multiplexer Mux_0 in a direction that depends on the orientation of the launching register, as shown in fig. 6.6. Equation (6.1) defines a relation between register orientation and PL channel direction. While the knowledge of the launching register's orientation is not sufficient in itself to calculate Mux_0's orientation, in conjunction with eq. (6.3), it does reduce the range of possible orientations as presented in eq. (6.10). It should be noted that orientation of Mux_0 alone is sufficient to ascertain the orientation of the associated launching register.

$$MuxOrient(Mux_0) \in \begin{cases} \{1,2\}, & Launch\_Reg_{orientation} = 1 \\ \{3,4\}, & Launch\_Reg_{orientation} = 2 \end{cases} \tag{6.10}$$

3. Multiplexer-LL Register connection: This scenario is similar to the one discussed above. If Mux_k is the last multiplexer in a path segment, the orientation of the landing register can only suggest that $MuxOrient(Mux_k) \in \{N, S\}$, or $MuxOrient(Mux_k) \in \{E, W\}$. On the contrary, eq. (6.11) shows that a known value of $MuxOrient(Mux_k)$

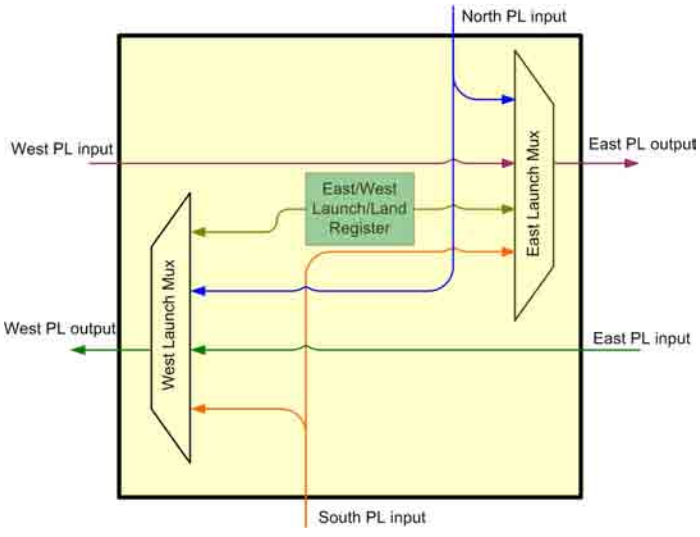


Fig. 6.6: Launch register and Mux_0 orientation.

is sufficient to determine the orientation of the landing register.

$$RegOrient(Reg_{dst}) = \begin{cases} 1, & MuxOrient(Mux_k) \in \{1,2\} \\ 2, & MuxOrient(Mux_k) \in \{3,4\} \end{cases} \tag{6.11}$$

The routing problem can now be defined as: Given a set of paths P, $\forall p \in P$, where path p has delay p_n, find locations, PL groups, and orientations of $p_n - 1$ intermediate LL registers and all necessary multiplexers, such that a continuous physical connection is established between source and destination silicon objects of the path, subject to the constraint that no LL register or multiplexer is reused.

6.3 Routing Algorithm

The placement tool generates a placed TDFG, denoted by $TDFG_{placed}$. Each edge $e = (v_s, v_d)$ in $TDFG_{placed}$ corresponds to a physical path $p = (O_s, O_d)$, where O_s and O_d are the objects on which nodes v_s and v_d have been placed, respectively. The delay along path p is equal to the delay along edge e. Algorithm 6.1 describes a methodology for routing each path p by assigning a party line, and allocating LL registers and multiplexers to establish a physical connection between the placed nodes.

The inputs to the algorithm are $TDFG_{placed}$, $\Re_{FPOA}$, and all the available routing resources on the FPOA. The algorithm iterates over all unrouted paths and routes them one by one. In line (7) a path $p \in unrouted\_path\_set$ is selected, where $unrouted\_path\_set$ represents the set of all unrouted paths in the $TDFG_{placed}$. A PL group is assigned to p in line (8). The following three attributes are associated with each path: delay $Delay_p$, source silicon object $SrcObj$, and a destination silicon object $DstObj$. $SrcObj$ and $DstObj$ are the silicon objects on which source and destination nodes of edge e are placed, respectively. As mentioned in the previous section, $Delay_p$ determines the number of LL registers required for routing a path. A path is comprised of $Delay_p$ segments, where the first segment begins with $SrcObj$, last segment ends at $DstObj$, and all intermediate segments end at an LL register.

Algorithm 6.1 Routing algorithm

```
// Routing
(1) input  : Resource set ℜ_FPOA
(2) input  : All routing resources in the FPOA
(3) input  : Placed TDFG = (V_R, E_R, D_e, Placement : V_R → ℜ_FPOA)
(4) available_reg_set = all LL registers in the FPOA
(5) available_mux_set = all multiplexers in the FPOA
(6) unrouted_path_set = all unrouted paths corresponding to edges e ∈ E_R
(7) forall  p = (SrcObj, DstObj) ∈ unrouted_path_set do
(8)  select a PL group for path p
(9)  n = Delay_p;
(10)  forall  i from 1 to n do
(11)   select Reg_i from available_reg_set such that
(12)     Dist(ID(SrcObj), RegLoc(Reg_i)) ≤ 4i
(13)     Dist(RegLoc(Reg_i), ID(DstObj)) ≤ 4(n − i)
(14)   available_reg_set ← available_reg_set − {Reg_i}
(15)   select Mux_{i_0} from available_mux_set such that
(16)     if (i == 1) then
(17)      MuxLoc(Mux_{i_0}) = ID(SrcObj)
(18)     else
(19)      MuxLoc(Mux_{i_0}) = RegLoc(Reg_{(i−1)})
(20)      switch (RegOrient(Reg_{(i−1)}))
(21)       case North-South: Determine MuxOrient(Mux_{i_0}) = North or South
(22)       case East-West: Determine MuxOrient(Mux_{i_0}) = East or West
(23)      end
(24)     end
(25)   available_mux_set ← available_mux_set − {Mux_{i_0}}
(26)   last_mux = 0
(27)   forall  j from 1 to 3 do
(28)    if (path segment i is incomplete) then
(29)     select Mux_{i_j} from available_mux_set such that
(30)       Dist(MuxLoc(Mux_{i_{j−1}}), MuxLoc(Mux_{i_j})) = 1
(31)       Dist(RegLoc(Reg_i), MuxLoc(Mux_{i_j})) ≤ 4 − j
(32)       MuxOrient(Mux_{i_{(j−1)}}) = Position of Mux_{i_j} in reference to Mux_{i_{(j−1)}}
(33)    last_mux = j
(34)    available_mux_set ← available_mux_set − {Mux_{i_j}}
(35)   end
(36)  end
(37)   switch (RegOrient(Reg_i)))
(38)    case North-South: Determine MuxOrient(Mux_{i_{last_mux}}) = North or South
(39)    case East-West: Determine MuxOrient(Mux_{i_{last_mux}}) = East or West
(40)   end
(41)  end
(42) end
```

Line (9) declares a variable n which denotes the delay along the path. The next task is to iteratively assign n LL registers to the path. Routing proceeds by selecting a physical LL register Reg_i in line (11). Lines (12) and (13) require that this register selection satisfies eq. (6.5a-b). The selected register is removed from the set of available registers in line (14). If the selected register is the first LL register in the path, then this is the first path segment in p and the first multiplexer in the first path segment always resides on the source object $SrcObj$. A multiplexer Mux_{i_0} for the current path segment is selected in line (15) subject to location and orientation restrictions imposed in lines (16) to (24). Line (16) evaluates the condition to test if this is the first path segment in the path, and if the condition evaluates to true, then the location of Mux_{i_0} is set equal to the location of $SrcObj$ in line (17). Otherwise, Mux_{i_0} does not belong to the first path segment in p and line (19) co-locates Mux_{i_0} with the previous LL register Reg_{i-1}. The orientation of Mux_{i_0} is determined in lines (20) to (24), depending on the orientation of Reg_{i-1}. Mux_{i_0} is now removed from the available multiplexer set.

After selecting the first multiplexer, the algorithm continues multiplexer allocation based on routing requirement. If more multiplexers are needed to route a path segment, another multiplexer Mux_{i_j} is selected in line (29) and is subject to the adjacency criteria imposed by eq. (6.6), in lines (30) and (31). Line (32) indicates that the relative location of Mux_{i_j} and its predecessor $Mux_{(i-1)_j}$ must be in the same direction as indicated by the orientation of $Mux_{(i-1)_j}$. Mux_{i_j} is removed from the available multiplexer set in line (34). The orientation of the last multiplexer in the current path segment is determined using the orientation of the destination LL register in lines (37) to (40). Routing of the current path segment is complete when all four multiplexers have been allocated, or if no more multiplexers are needed.

After allocating multiplexers to the current path segment, the routing process continues by selecting the next LL register. If all n registers in the current path have been allocated, the routing for this path is complete. Routing continues by selecting another unrouted path and the above procedure is repeated until no more unrouted paths remain.

Similar to the placement algorithm, the time complexity of Algorithm 6.1 is exponential in worst case. Consider the loop starting in line (7). In the worst case, the number of iterations of this loop is equal $|E|$, where $|E|$ is the number of paths, corresponding to the edges in the input TDFG. For a path i with delay = n_i, lines (11) to (13) select n_i registers by considering $(N_{Reg})!/(N_{Reg} - n_i)!$ permutations in the worst case, where N_{Reg} is the number of registers in *available_reg_set*. However, if suitable registers are unavailable to implement a path, a previously routed path must be re-routed. Hence, for $i = 1..|E|$ paths, register selection for all the paths must consider a worst case permutation of $(N_{Reg})!/(N_{Reg} - \sum_{i=1}^{|E|} n_i)!$. Hence, the algorithm has complexity $O((N_{Reg})^{|E|})$, and is exponential in the worst case.

6.4 Solving Routing Problem Using FD Constraints

Algorithm 6.1 enumerates the steps involved in routing paths on an FPOA. The next challenge is to translate the routing algorithm into a constraint satisfaction problem. This is done by representing all unknowns in the algorithm using FD variables and defining constraints over these variables, as described in the following section. For simplicity, translation of a problem involving a single path is discussed, but the approach can be extended to problems with more than one path.

6.4.1 FD Variables and Constraints

Routing a path with delay n, requires n registers, a maximum of $4n$ multiplexers, and one party line group. Oz procedure *InitializeLLRegisters* creates and initializes FD variables for all LL registers in a path, as shown in fig. 6.7. Line (7) creates a data structure with six FD variables denoting a register's unique id, location, x and y coordinates, orientation, and PL group, respectively. Lines (8) - (13) specify the domain of these variables. Line (15) implements eq. (6.2) by defining a relationship among the following variables: id, loc, orient, and plgroup. Location of a register is bound to its coordinates in line (17). Lines (19) uses reified variables to constrain the orientation of the register based on its party line group. Specifically, if a register belongs to PL group 3, then it is only allowed to have an NS

```
(1)  proc {InitializeLLRegisters SrcObj DstObj NumOfReg ?RegTup}
(2)    XMax CurrReg
(3)  in
(4)    RegTup = {Tuple.make pathreg NumOfReg}
(5)    XMax = 20
(6)    {For CurrReg in 1 to NumOfReg
(7)      RegTup.CurrReg = {FD.record llreg [id loc x y orient plgroup] FD.sup}
(8)      RegTup.CurrReg.id ::: 1#2000
(9)      RegTup.CurrReg.loc ::: 1#400
(10)     RegTup.CurrReg.x ::: 1#20
(11)     RegTup.CurrReg.y ::: 1#20
(12)     RegTup.CurrReg.orient ::: 1#2
(13)     RegTup.CurrReg.plgroup ::: 1#3
(14)
(15)     RegTup.CurrReg.id =: (RegTup.CurrReg.loc-1) * 5
(15a)      +(RegTup.CurrReg.plgroup-1) * 3+RegTup.CurrReg.orient
(16)
(17)     RegTup.CurrReg.loc =: (RegTup.CurrReg.y-1) * XMax + RegTup.CurrReg.x
(18)
(19)     (RegTup.CurrReg.plgroup =: 3) * (RegTup.CurrReg.orient =:1)
(19a)      + (RegTup.CurrReg.plgroup \=: 3) =: 1
(20)
(21)     if (CurrReg \= 1) then
(22)       {ApplyRoutingProximityConstraint CurrReg RegTup 4}
(23)     end
(24)   }
(25)
(26)   PathRegTuple.NumOfReg.loc =: DstObj
(27) end
```

Fig. 6.7: Initializing finite domain variables for all launch and land registers in a single path.

orientation as mentioned in eq. (6.1). Lines (21) - (23) invoke procedure *ApplyRoutingProximityConstraint*, shown in fig. 6.8, to enforce the distance restriction specified in eq. (6.5). Lines (6)-(24) execute *NumOfReg* times, where *NumOfReg* is equal to delay n, creating records for all n registers. The case of the n^{th} register is special because it always resides on the destination silicon object of a path. Since its location is known apriori, line (26) binds the location of this last register to the destination silicon object. After completing execution, procedure *InitializeLLRegisters* returns a data structure, *RegTuple*, containing individual records of all registers.

```
(1)  proc {ApplyRoutingProximityConstraint Index InputTup MaxDist}
(2)    X_dist Y_dist SrcX SrcY DstX DstY
(3)  in
(4)    X_dist = {FD.int 0#20}
(5)    Y_dist = {FD.int 0#20}
(6)    SrcX = InputTup.(Index-1).x
(7)    SrcY = InputTup.(Index-1).y
(8)    DstX = InputTup.Index.x
(9)    DstY = InputTup.Index.y
(10)
(11)   X_dist =: (SrcX >=: DstX) * (SrcX - DstX)
(11a)    + (SrcX <: DstX) * (DstX - SrcX)
(12)   Y_dist =: (SrcY >=: DstY) * (SrcY - DstY)
(12a)    + (SrcY <: DstY) * (DstY - SrcY)
(13)
(14)   X_dist + Y_dist =<: MaxDist
(15) end
```

Fig. 6.8: Proximity constraints for consecutive launch and land registers in a path.

Procedure *ApplyRoutingProximityConstraint* operates on data structure *RegTup* to specify an upper bound on the distance between two connected registers in a path. *Index* denotes a successor register in a pair of connected registers. *X_dist* and *Y_dist* are declared as FD variables with domains 0#20 in lines (4) and (5). Lines (6) to (9) retrieve the coordinates of both registers. Horizontal and vertical distances between these registers are calculated in lines (11) to (12) and the total distance is constrained by upper bound *MaxDist* in line (14).

Figure 6.9 presents an Oz implementation for allocating multiplexers to a path segment. Unlike register allocation, where the number of registers is fixed by the delay along the path, the number of multiplexers vary between 1 and 4. Procedure *InitializePathSegMuxes* is similar to procedure *InitializeLLRegisters*. Line (7) creates a data structure to store FD variables that represent unique id, location, x and y coordinates, orientation, and PL group of a multiplexer. Lines (8) - (13) assign appropriate domains to these FD variables. In spite of having 4000 multiplexers, the domain of *id* ranges from 0 to 4000. Value 0 is reserved for indicating that a multiplexer allocation is not required, and is used for handling cases when

```
(1)  proc {InitializePathSegMuxes StartObj EndObj ?MuxTup}
(2)    XMax CurrMux
(3)  in
(4)    PathMuxTuple = {Tuple.make pathmux 4}
(5)    XMax = 20
(6)    {For CurrMux in 1 to 4
(7)      MuxTup.CurrMux = {FD.record muxes [id loc x y orient plgroup] FD.sup}
(8)      MuxTup.CurrMux.id ::: 0#4000
(9)      MuxTup.CurrMux.loc ::: 1#400
(10)     MuxTup.CurrMux.x ::: 1#20
(11)     MuxTup.CurrMux.y ::: 1#20
(12)     MuxTup.CurrMux.orient ::: 1#4
(13)     MuxTup.CurrMux.plgroup ::: 1#3
(14)
(15)     (MuxTup.CurrMux.id =: (MuxTup.CurrMux.loc-1) * 10
(15a)      +(MuxTup.CurrMux.plgroup-1) * 3+MuxTup.CurrMux.orient)
(15b)      +(MuxTup.CurrMux.id =: 0) =: 1
(16)
(17)     MuxTup.CurrMux.loc =: (MuxTup.CurrMux.y-1) * XMax
(17a)      + MuxTup.CurrMux.x
(18)
(19)     (MuxTup.CurrMux.plgroup =: 3) * (MuxTup.CurrMux.orient =: 1)
(19a)      +(MuxTup.CurrMux.plgroup =: 3) * (MuxTup.CurrMux.orient =: 2)
(19b)      + (MuxTup.CurrMux.plgroup \=: 3) =: 1
(20)
(21)     if (CurrMux \= 1) then
(22)       {ApplyRoutingProximityConstraint CurrMux MuxTup 1}
(23)     end
(24)   }
(25)
(26)   PathMuxTuple.0.id \=: 0
(27)   PathMuxTuple.0.loc =: SrcObj
(28) end
```

Fig. 6.9: Initializing finite domain variables for all multiplexers in a path segment.

a path segment needs less than four multiplexers. The reified constraint in lines (15) binds the unique id of a multiplexer to its location, PL group and orientation, or assigns a value 0 to the id. Line (17) defines a relation between a multiplexer's location and its coordinates. If PL group 3 is used, then a multiplexer must be oriented in North or South direction only. This restriction is imposed by the constraint in line (19). Line (21) applies procedure *ApplyRoutingProximityConstraint* to all multiplexers in the path segment and constrains

any two connected multiplexers to be located a unit distance apart. Lines (26) and (27) pertain to the first multiplexer in a path segment. In a path segment, the first multiplexer is always used, even if no other multiplexers are needed. Additionally, the first multiplexer is co-located with the starting element of the path segment. Both these requirements are imposed by lines (16) and (27), respectively.

6.4.2 Improving Search Convergence

Two strategies have been found to be effective in improving search convergence, both in terms of convergence time and memory usage. The first one is a divide and conquer approach and the second is to route shorter paths first.

In the divide and conquer approach, instead of routing all paths at once, only one path is routed at a time because it offers the following advantages.

1. Since only one path is routed at a time, any resource from the pool of available resources can be selected. Thus, either the path gets routed or it is quickly ascertained that no valid route exists due to unavailability of routing resources and the search backtracks. Instead, if all routes are handled concurrently, resource conflicts may not surface until later causing severe backtracking.

2. Propagators associated with a particular path get entailed when the path gets routed. This reduces memory consumption and improves performance as search proceeds because the amount of propagation required after each distribution step steadily decreases.

3. If routing cannot proceed beyond a certain path, then the unroutable path is immediately identified. This information is useful for determining the causes of unroutability and helps in making decisions regarding delay relaxation along a path or need for a new placement of the TDFG.

The second strategy is to route shorter paths first. Shorter paths typically have the least amount of flexibility and are prone to resource conflicts. This approach helps in early

detection of cases where multiple short routes cause resource conflicts and the design is not routable.

6.4.3 Distribution Strategy

The routing algorithm involves several FD variables representing locations, unique ids, orientations, and party line groups of registers as well as multiplexers. As mentioned earlier, propagation attempts to narrow down the domains of these variables but is typically insufficient for converging to a solution and that is when distribution plays an important role in resuming propagation. Theoretically, it is possible to find a solution by distributing on any variable; but in practice, variable selection influences search convergence in terms of both time and memory. A preferred selection strategy is the one that causes the search to converge faster while keeping memory usage within allowed limits.

Figure 6.10 illustrates the distribution strategy used in the routing tool. This strategy specifies the variable selection policy. A first-fail heuristic is used within each variable group. For example, when distributing on register locations, all register location variables are subject to a first-fail heuristic, implying that a register location variable with minimal domain size is distributed on first.

After selecting the shortest path, all its ungrounded register location variables are identified and one of these variables is chosen according to the first-fail heuristic. The domain of this variable is split and the upper part is tried first. Experiments indicate that trying the lower part first results in longer path segments and increases congestion in the bottom left region of the FPOA. On the other hand, selecting the upper part of the domain typically results in shorter and direct routes, and favors North-South channels, increasing the use of PL group 3 whenever possible.

The above process is repeated until no more ungrounded register variables remain. Next, all ungrounded multiplexer locations belonging to the current path are selected and once again the first fail heuristic is applied to select a multiplexer location. Distribution continues until all multiplexer location variables have been grounded. Once again a domain splitting strategy is used, however, the lower part of the domain is tried first. This approach

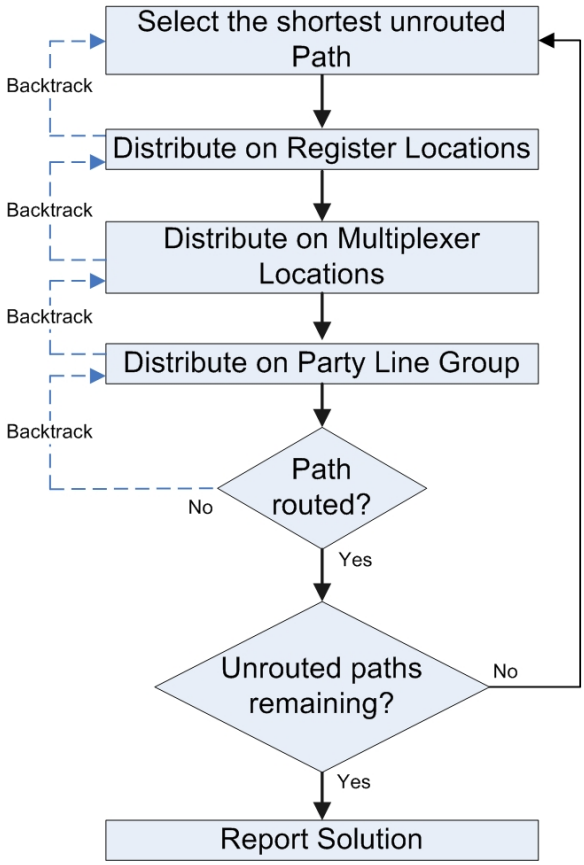


Fig. 6.10: Distribution strategy for routing.

compliments the register location distribution strategy by favoring North-South channels.

The distribution is performed on the register and multiplexer location variables first because of the following reasons. First of all, if the variables representing the orientations of these routing resources are distributed on first, most of the resources are assigned orientations which do not generate valid routes, requiring extensive backtracking. On the other hand, once the location of registers and multiplexers is known, propagation alone is sufficient to ground the orientation variables, offering a more efficient and elegant approach.

The next step assigns a party line group to the current path. A first fail heuristic assigns the largest possible value in the domain to the PL group. Since PL group 3 only contains North-South channels, it is preferred whenever the route travels strictly in the North-South direction. However, if any segment of a route does not travel North-South, propagation removes the value 3 from the domain of PL group variables, and assigns PL

group 2 instead. This biased approach decreases the demand for PL groups 1 and 2 whenever possible, and makes more resources available to paths that must use a mix of channels traveling in North-South and East-West directions.

If the path is unroutable due to resource conflict, distribution backtracks and assigns a different PL group. If changing PL group does not lead to a valid route, multiplexer and/or register locations are reassigned, followed by PL group allocation. If all these steps fail, then a previously routed path is re-routed followed by a new attempt to route the current path.

It may be argued that since there are only three party lines, grounding PL group variables first would be more appropriate. But in practice, selecting a PL group early on in the search negatively affects search convergence because of three main reasons. First, a single PL group gets selected for a majority of the paths, limiting the availability of registers and multiplexers. Only after excessive backtracking is it determined that the PL group needs to be changed. Second, even though PL group 3 is an ideal choice for paths traveling in the North or South directions, PL groups 1 or 2 may be assigned to such paths, committing resources that are required by paths traveling in the East or West directions. The problem arises from the fact that enough information about a path's direction is not available when the search commences. Finally, if FD variables are grounded at the beginning of search, they cannot be changed at a later stage without backtracking. Thus, in the case of a resource conflict, moving an entire route to a different party line becomes a non-trivial process because it requires backtracking and redoing most of the search. Hence, PL groups are distributed upon last. If one assignment to a PL group causes a resource conflict, then a different group is assigned which effectively moves all registers and multiplexers to a different PL channel, resolving resource conflicts. This approach does not guarantee a resolution in all cases, but is experimentally found to be effective in improving convergence.

6.5 Summary

A formal model for representing the routing resources of an FPOA is proposed in this chapter. A routing methodology using the PL communication network of an FPOA is

developed. The proposed approach focuses on the assignment of routing resources to all non-zero delay edges in a placed TDFG such that a physical path is established between the silicon objects on which the nodes, connected by an edge, are placed. The PL communication consists of two types of routing resources: the LL registers and the multiplexers, which are configured to form physical interconnects. Each resource is identified using its three properties: location, orientation, and PL group. The routing problem is phrased as a search problem where the solution lies in finding resources at suitable locations, with the desired orientations, in the correct PL group to form a contiguous path between a source and destination object. An algorithm is proposed as a solution to the routing problem and routes a single path at a time. This algorithm is translated into a finite domain constraint representation that includes finite domain variables and a set of constraints defined over these variables. These variables represent the various properties of resources as defined above. The constraint solver assigns values to these variables, such that no constraints are violated.

Chapter 7

Results

The scheduling, placement and routing approaches, described in the previous chapters, are implemented in Oz and C++. All these tools are interfaced to form a design tool chain targeting the FPOA architecture. This end-to-end tool chain is applied to a set of benchmarks in order to demonstrate and evaluate the constraint satisfaction-based tools. This chapter showcases the results obtained during each of these three phases. Section 7.1 introduces the benchmarks that are used to evaluate the approach proposed in this research. The performance of each individual tool is evaluated in sec. 7.2, and sec. 7.3 presents the performance results obtained by varying problem size for one of the benchmarks. All the results are obtained using an Intel® Q6600 quad core processor running at 2448 MHz, with 4GB RAM.

7.1 Overview of Benchmarks

Eight benchmarks were used for evaluating the scheduling, placement, and routing tools developed in this research. Table 7.1 lists these benchmarks. These test cases belong to various types of application domains such as scientific computing, signal processing, and multimedia applications. Benchmark 1 corresponds to a one-level 1-Dimensional (1D) Discrete Wavelet Transform (DWT) [113] for a signal with length = 18. Benchmark 2 represents a single iteration in the *mdct_short()* function used in LAME [114] encoder for computing the Modified Discrete Cosine Transform (MDCT). An 8-point Discrete Fourier Transform (DFT) comprises benchmark 3. Benchmark 4 represents the Sum of Absolute Transformed Differences (SATD) function in the H.264 encoder [115] and computes the SATD of a 4×4 block using Hadamard transform. A $2 \times 8 \times 2$ integer matrix multiplication is considered in benchmark 5 while benchmark 6 is obtained from the *window_subband()* function in the

Table 7.1: Benchmarks used for evaluating scheduling, placement, and routing tools.

Id	Benchmark	Source/ Application	Nodes	Edges
1	Discrete Wavelet Transform (DWT)	GSL	120	118
2	Modified Discrete Cosine Transform (MDCT)	LAME MP3 encoder	58	68
3	Discrete Fourier Transform (DFT)	Signal Processing	48	60
4	Sum of Absolute Transformed Difference (SATD)	H.264 encoder	144	160
5	Matrix Multiplication (MM)	Scientific Applications	192	160
6	MP3 Window Subband (MWS)	LAME MP3 encoder	188	184
7	Finite Impulse Response filter (FIR)	Signal Processing	321	320
8	Five Step Search (FSS)	H.264 encoder	290	289

LAME encoder [114]. Benchmark 7 represents a 64-tap FIR filter. Benchmark 8 is comprised of nine 16×16 Sum of Absolute Difference (SAD) computation engines and a 9-input comparator. The Five Step Search (FSS) algorithm [116] employs benchmark 8 for block motion estimation during each of the five steps. It should be noted that benchmarks 1, 2, 4, and 6 are generated from their respective C implementations.

Out of the eight cases, benchmarks 5, 7, and 8 either use an FPOA to 100% capacity, or almost fill the entire chip. Each benchmark is assigned a unique id for easy reference in subsequent sections. The second column in the table gives the name of each benchmark, followed by its source or application domain, and nodes and edges provide the size of DFG representing a particular problem instance. An estimate of the gate count for a functionally equivalent ASIC implementation and the execution latency of the benchmarks is presented in Appendix B.

7.2 Performance Evaluation of Proposed Tools

The goal of the proposed Scheduling, Placement, and Routing tools is to find a valid solution while minimizing communication delay, area, and routing resource utilization. All three tools must work in tandem to achieve this goal. At the same time, the convergence time and the memory usage of these tools should be reasonable so that these tools are

practical and usable. The following sections present the performance data for each tool using the set of benchmarks provided in sec. 7.1.

7.2.1 Scheduling

The Scheduling tool is the first tool in the toolchain and is responsible for generating the schedule while minimizing communication delays, and for merging ALU operations. Figure 7.1 indicates that the Scheduling tool attempts to maximize zero delay edges in the generated TDFG in order to minimize the schedule length. An increased number of zero delay edges further improves search convergence during placement and routing phases as described later in sec. 7.2.2 and 7.2.3. For edges with non-zero delays, fig. 7.2 shows the average delay for each benchmark. This average is a weighted mean, and is calculated using the relation given in eq. (7.1), where K is the maximum delay assigned to an edge in the TDFG and $NumEdge_n$ is the number of edges with delay equal to n.

$$Average\ delay = \sum_{n=1}^{K} n * (NumEdge_n) \tag{7.1}$$

For all the benchmarks, the average delay is found to be between 1 and 2.2, suggesting that the Scheduling tool is biased towards smaller communication delays, which is desirable because it decreases the computation burden and amount of resources required during routing. The largest value of K is found to be four for benchmarks 2 and 8 and is found to be smaller for all other benchmarks.

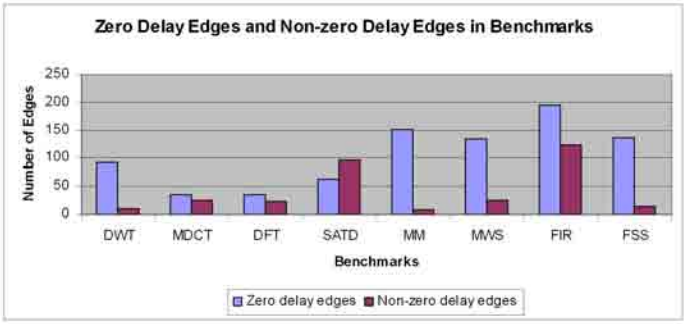


Fig. 7.1: Zero delay edges vs. non-zero delay edges.

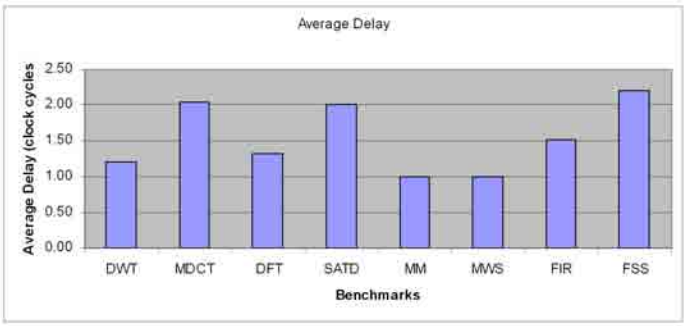


Fig. 7.2: Average delay (does not include zero delay edges).

As described in sec. 4.3, the Scheduling tool supports merging of ALU operations onto the same ALU by assigning each operation to a different ALU instruction state. Figures 7.3 and 7.4 show the number of nodes and edges in the DFG before and after scheduling. Due to merging, the total number of nodes and edges in a TDFG tend to be lower than the original DFG, which decreases the footprint by occupying fewer ALU objects, allows more than 256 ALU operations to be placed on an FPOA, and reduces both the placement and the routing problem size. An interesting case is Benchmark 8 which is comprised of 290 ALUs. Since there are only 256 ALU objects on an FPOA, it is impossible to fit the design in the absence of ALU operation merging. However, the Scheduling tool is able to allocate all these operations onto less than 256 ALUs. By reducing nodes in the resulting TDFG, a number of edges are eliminated which directly translates into reduced computational burden during routing.

As can be observed in fig 7.3, not every DFG sees a reduction in the number of nodes. Even if the Scheduling tool proposes node merging, the Schedule Analyzer may discard any suggestions that can lead to an unplaceable design arising from violation of topological constraints of an FPOA. For example, two MACs can never be nearest neighbors, and hence the Scheduling tool will not generate any schedule that may require two MACs to be NN. However, during ALU operation merging, the Scheduling tool may generate schedules that require two ALUs to be NN, where each is NN with three MACs. The Schedule Analyzer examines and avoids such outcomes. Thus, if no reduction in the number of nodes and edges is observed, it is either because no merging is possible, or because the merging is

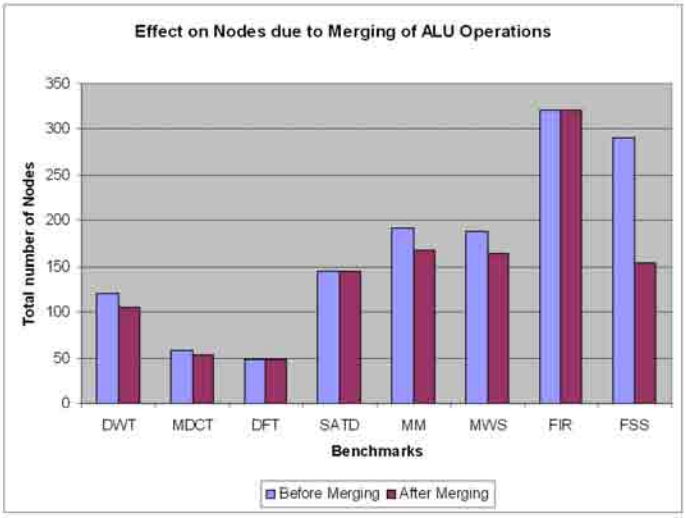


Fig. 7.3: Node reduction due to ALU operation merging.

not placeable and was rejected during the schedule analysis. The former case applies to benchmark 4 where no ALU merging is observed despite the fact that it is comprised of only ALUs.

Figure 7.5 shows the convergence time for scheduling DFGs for all eight benchmarks including the execution time of the Schedule Analyzer. Convergence time of a tool includes problem initiation, propagation, and distribution times, along with any output generation

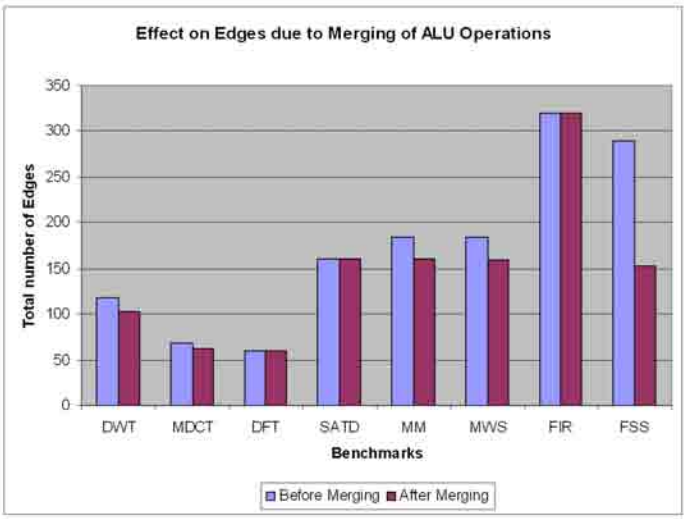


Fig. 7.4: Edge reduction due to ALU operation merging.

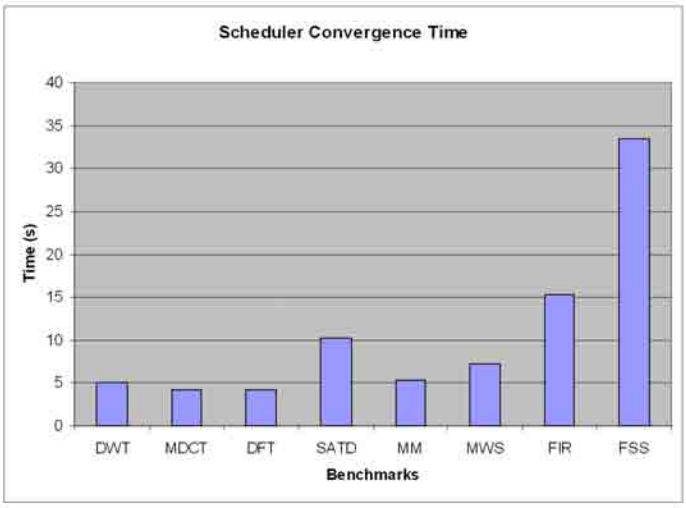


Fig. 7.5: Scheduling tool convergence time.

overhead. The Scheduling tool's memory usage is shown in fig. 7.6, which correlates with the convergence time, where memory usage refers to the peak memory used during a particular execution of a tool. Typically, more distribution steps imply higher convergence time and increased memory usage, with the exception that re-computation can trade-off the amount of memory needed for search with the execution time.

For all the test cases except 7 and 8, the Scheduling tool converged quickly using less than 1 GB memory. Benchmarks 7 and 8 have a large number of nodes as well as edges,

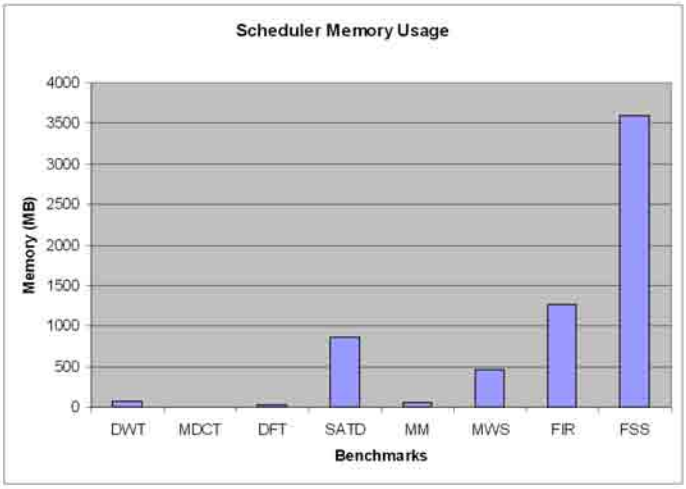


Fig. 7.6: Scheduling tool memory usage.

which translates into a large number of propagators. Since the scheduler involves variables representing time, and the domains of times variables tend to be large, the search is prone to huge memory usage, which may result in premature termination of the search. As described in Chapter 3, it is possible to use re-computation to reduce the amount of memory required for a search at the cost of increased computation burden. The proposed implementation of the Scheduling tool allows a user to configure the tool by specifying recomputation step size S_{RC}, where $S_{RC} = 1$ implies a standard search that saves state after each distribution step. After using re-computation with $S_{RC} = 20$, both benchmarks 7 and 8 were scheduled in 15.35 and 33.45 seconds, respectively, with memory usage shown in fig.7.6. Thus, the scheduler is found to converge in each case, in less than 16 seconds for seven out of the eight test cases, and supports re-computation for cases requiring a large amount of memory.

Figure 7.7 shows the total number of distribution steps along with backtracks performed in order to arrive at a solution. For all the benchmarks except 7 and 8, the Scheduling tool performs minimal backtracking, indicating that a correct path was quickly identified. The case of benchmarks 7 and 8 is different since they use re-computation and even though the number of distribution and backtracks are considerably less than benchmark 5 and 6, re-computation increases the computation time spent per distribution step, resulting in a longer convergence time.

In fig. 7.7, it appears that the Scheduling tool distributes significantly for Benchmarks

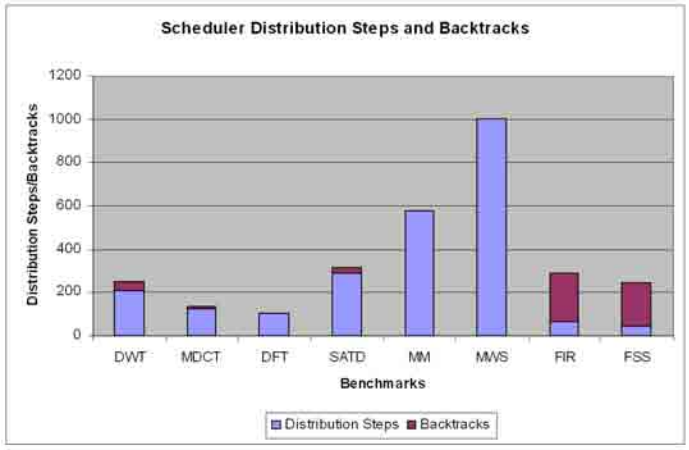


Fig. 7.7: Scheduling tool distribution and backtracks.

5 and 6, than for benchmarks 1-4, but converges faster for these cases than benchmark 4. Similarly, in spite of more distribution and backtracking in the case of benchmark 7 vs. benchmark 8, the Scheduling tool performs better for benchmark 8 than for benchmark 7. The reason for this anomaly lies in the fact that benchmarks 5, 6, and 7 use the divide and conquer approach, described in sec. 5.4.2.

While the real benefits of divide and conquer approach are noticeable during placement, it requires the Scheduling tool to be invoked once for each new module added to the design. Initially, the DFG is small, resulting in fewer variables, propagators, and lower distribution overhead. However, the distribution overhead increases with the size of the DFG. In other words, an increase in size of a DFG may not only increase the number of distribution steps, but it also increases the computation time per distribution step. Since distribution overhead grows with problem size, the rate of growth of cumulative distribution and backtracking steps may not correlate with a slower increase in cumulative convergence time and memory usage. Due to this reason, the cumulative distribution and backtracking steps for benchmarks 5 and 6 are more than the single execution distribution and backtracking steps for benchmark 4, but the convergence time and memory usage of the Scheduling tool for benchmarks 5 and 6 are smaller than that for benchmark 4. A similar effect is observed between the Scheduling tool's performance for benchmarks 7 and 8 which additionally require re-computation.

7.2.2 Placement

In this research, the TDFGs for all the benchmarks given in Table 7.1 are successfully placed on an FPOA. The primary goal of placement is to assign compatible silicon objects to all the nodes such that all proximity constraints are satisfied. For each of the benchmarks, the tool successfully performs a delay aware placement.

Figures 7.8 and 7.9 show placement convergence time and memory usage, respectively. These graphs establish a strong correlation between memory demand and convergence time. Though it may seem obvious that the Placement tool takes a longer time to place a TDFG with more nodes, benchmarks 1 and 4 indicate otherwise. Both the number of nodes as well

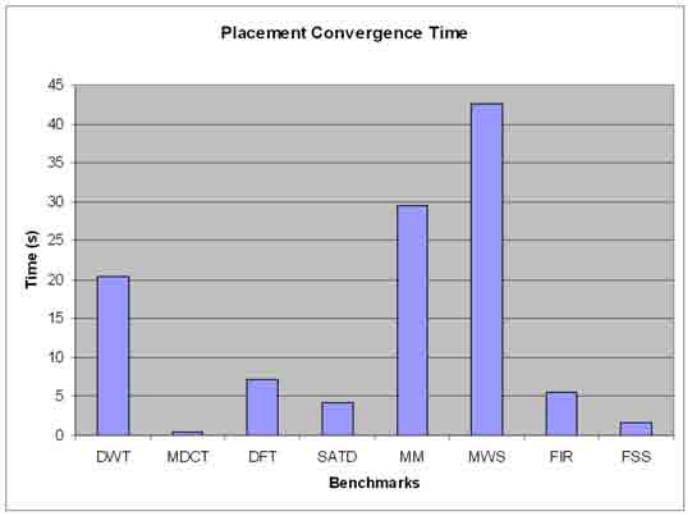


Fig. 7.8: Placement tool convergence time.

as edges in benchmark 1 are lower than for benchmark 4, but the Placement tool converges faster in the latter case. This can be explained based on the distribution, as shown in fig. 7.10. In particular, the tool backtracks significantly more when applied to benchmarks 1 and 3, as compared to benchmarks 2 and 4, which results in a longer convergence time for the smaller problem.

As mentioned in Chapter 3, backtracking indicates that search proceeded along incorrect paths until the right path was selected. A good solver attempts to improve backtracking

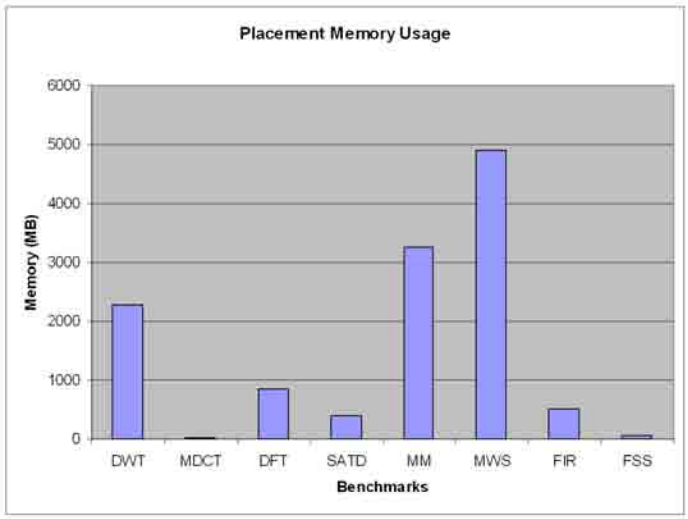


Fig. 7.9: Placement tool memory usage.

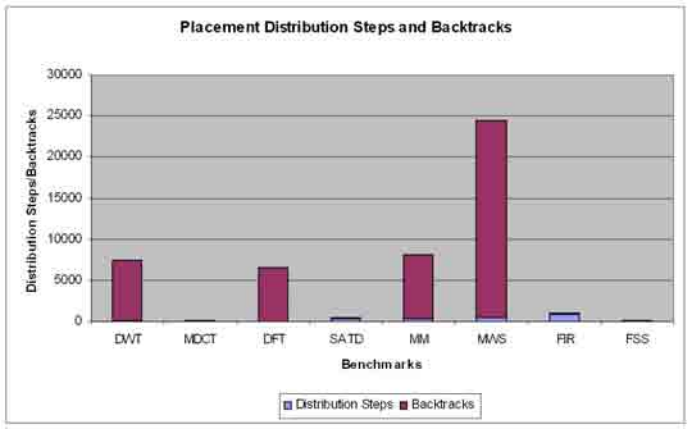


Fig. 7.10: Placement tool distribution and backtracks.

by using heuristics to establish early enough that the search is on a deadend path. The Placement tool uses redundant constraints called additional proximity constraints (see sec. 5.4.2) for improving propagation to quickly establish if the current path does not yield a valid solution. While these improvements are apparent in fig. 7.10 for benchmarks 2, 4, 7, and 8, the distribution and backtracking steps for the remaining benchmarks are significantly larger in the absence of redundant constraints.

The placement tool takes longer to converge for benchmarks 5, 6, and 7 with higher memory requirements, but is kept in check by applying a divide and conquer approach as shown in fig. 7.8. The benefits of the divide and conquer approach are particularly noticable in the case of benchmark 7. For bencmark 7, the convergence time of the placement tool would be significantly higher in the absence of the divide and conquer approach. It should be noted that cumulative results have been reported for these three cases. As mentioned earlier, more distribution steps typically mean longer search convergence time and memory requirements. However, the amount of propagation and overhead during each distribution step significantly impacts the actual execution time and memory usage. The divide and conquer approach reduces the amount of propagation per step, as well as the distribution overhead, thereby improving convergence time while utilizing less memory.

Another interesting comparison is between benchmarks 5 and 6. The number of distribution and backtracking steps reported by the tool are approximately three times more

for the latter case than in the former, but convergence time of the tool is only 33% more for benchmark 6 than for benchmark 5. Once again distribution overhead and propagation are responsible for only a marginal increase in convergence time. Though these benchmarks benefit from divide and conquer, maximal backtracking happens during initial placement of benchmark 6, due to variable selection. During the initial placement, lower distribution overhead compensates for a large number of distribution steps, bringing down the overall convergence time for benchmark 6.

The performance of placement tool is further improved by applying the bounding box, and divide and conquer approaches as discussed in sec. 5.4.2. Table 7.2 compares the performance of the placement tool, with and without the application of the bounding box. In all the cases, bounding box improves the convergence times and reduces the memory usage. Diminishing gains are observed for large problems because the initial size of the bounding box increases with the probem size. The effect of divide and conquer approach on placement tool performance is presented in Table 7.3. Performance gains are observed for benchmarks 2 and 4 while a performance deterioration is noticed in the case of benchmakrs 3 and 8. It should be noted that cumulative execution time and memory usage of each execution are reported when divide and conquer approach is used. A higher convergence time is observed in the case of benchmark 3 because the initial placement is modified during incremental placement steps, which increases the convergence time and the memory usage. For benchmark 8, the convergence time during each placement tool execution is found to be

Table 7.2: Effect of bounding box on placement tool performance.

Id	**Benchmark**	**With bounding box**		**Without bounding box**	
		Time (sec)	**Memory (MB)**	**Time (sec)**	**Memory (MB)**
2	Modified Discrete Cosine Transform	0.52	19.27	3509.99	512578.16
3	Discrete Fourier Transform	7.20	843.15	36.95	4336.20
4	Sum of Absolute Transformed Difference	4.14	399.52	8.14	849.60
8	Five Step Search	1.61	66.70	1.64	72.40

Table 7.3: Effect of divide and conquer on placement tool performance.

Id	**Benchmark**	**Without divide and conquer**		**With divide and conquer**	
		Time (sec)	**Memory (MB)**	**Time (sec)**	**Memory (MB)**
2	Modified Discrete Cosine Transform	0.52	19.27	0.36	18.49
3	Discrete Fourier Transform	7.20	843.15	11.19	1073.80
4	Sum of Absolute Transformed Difference	4.14	399.52	2.69	186.11
8	Five Step Search	1.61	66.70	1.66	69.15

less than 1 second. However, the cumulative overhead overshadows the benefits of applying the divide and conquer approach.

In addition to improving search convergence time, the Placement tool uses the bounding box approach to avoid scattering the nodes throughout the FPOA, in order to minimize the chip area occupied by the design. However, the post-placement design footprint depends on the topology of the TDFG and is strongly linked to the number of MAC and RF nodes in the TDFG because the MAC and RF objects are spread throughout the FPOA chip. For example, after scheduling the DFGs of benchmarks 6 and 8, their respective TDFGs are composed of comparable number of nodes. However, the placement solution reported for benchmark 6 is spread throughout the FPOA chip since it contains 64 MAC operations which must be placed on MAC objects that are sparsely located over the entire chip. On the other hand, benchmark 8 is placed using only 75% of the chip area because it is comprised of ALU operations, which are placed on ALU objects that are more readily available than MACs. The post-placement layouts of all the benchmarks are shown in Appendix C.

In order to compare the performance of the FD constraint-based placement tool, another placement tool is required. However, no other tools exist for placing a design on an FPOA. Hence, a placement tool is developed using the traditional simulated annealing approach, independent of the proposed FD constraint-based placement methodology, and serves as a reference for performance comparison. The Simulated Annealing-based Placement (SAP) tool borrows some of the concepts that are discussed in Chapter 5. Each node

in the input TDFG is characterized by four attributes: a unique id, pair of coordinates, and the operation type as described in sec. 5.2. Placement of a node on an object is indicated by assigning the unique id of the object to the unique id attribute of the node.

The SAP tool uses a random number generator which uses the current system time as the seed value. Initially, a placement solution is obtained by randomly assigning a compatible object to each node in the TDFG. However, this placement may be invalid and may not satisfy the communication delay requirements imposed by the design specification. The cost function used in the SAP tool determines the number of unsatisfied communication delays in the design. The placement objective is to minimize this cost to zero, implying that the solution satisfies all delay requirements. The invalid placement is modified by randomly selecting a node and placing it on a randomly selected but compatible object. This replacement or move is considered valid if it does not increase the value of the cost function. Valid moves are always allowed, however, invalid moves may be permitted depending on the annealing temperature to avoid getting stuck in a local minima. Initially the temperature is high and a large number of invalid moves are allowed even if they increase the cost function. As the temperature reduces, so does the number of invalid moves. After each move, the placement tool checks if a solution satisfying all delay requirements has been found. If a valid solution is found, it is reported and the SAP tool terminates; otherwise, another move is performed. If a solution is not found and successive moves do not improve the placement cost, the temperature is increased in an attempt to avoid local minima and the placement process continues.

Table 7.4 shows the convergence time and memory usage of the SAP tool and compares it with the performance of the FD constraint-based placement tool. Except for benchmark 1, the FD constraint approach converges faster than the simulated annealing-based method, but the former uses more memory in all test cases. However, for benchmarks 3, 5, and 6, the SAP tool did not report a solution. For these three cases, the final placement cost varied between 1 and 6, indicating that while the tool was able to satisfy most of the delay requirements, all the communication delay requirements imposed by the design specification

Table 7.4: Performance of simulated annealing-based placement.

Id	Benchmark	Simulated Annealing		FD Constraints	
		Time (sec)	Memory (MB)	Time (sec)	Memory (MB)
1	Discrete Wavelet Transform	1.31	2.50	20.39	2284.98
2	Modified Discrete Cosine Transform	41.05	2.42	0.52	19.28
3	Discrete Fourier Transform	-.--	-.--	7.20	843.15
4	Sum of Absolute Transformed Difference	37.40	2.52	4.14	399.53
5	Matrix Multiplication	-.--	-.--	29.53	3245.82
6	MP3 Window Subband	-.--	-.--	42.61	4901.17
7	Finite Impulse Response filter	1589.12	2.91	5.53	521.22
8	Five Step Search	427.50	2.91	1.61	66.70

could not be satisfied.

In summary, the Placement tool successfully finds placement solutions for all the eight benchmarks considered in this research. These solutions satisfy all the proximity constraints imposed by the design specification. The results indicate that the search convergence time and memory usage are not only related to the problem size, but are also dependent on the amount of backtracking, which in turn depends on the variable selection strategy. The performance of the Placement tool is improved by employing strategies such as divide and conquer, and redundant constraints, which decrease distribution overhead and improve backtracking.

7.2.3 Routing

After scheduling and placing a DFG, the final step in the tool flow is routing. In a TDFG, an NN communication channel corresponds to a zero delay edge, and PL communication channels are required for routing non-zero delay edges. Figure 7.11 shows the number of NN and PL communication channels required for routing each benchmark. All zero delay edges are handled during delay aware placement by ensuring that any pair of objects sharing a zero delay edge are nearest neighbors. The proposed Routing tool is concerned only with the non-zero delay edges which translate into a non-zero delay communication

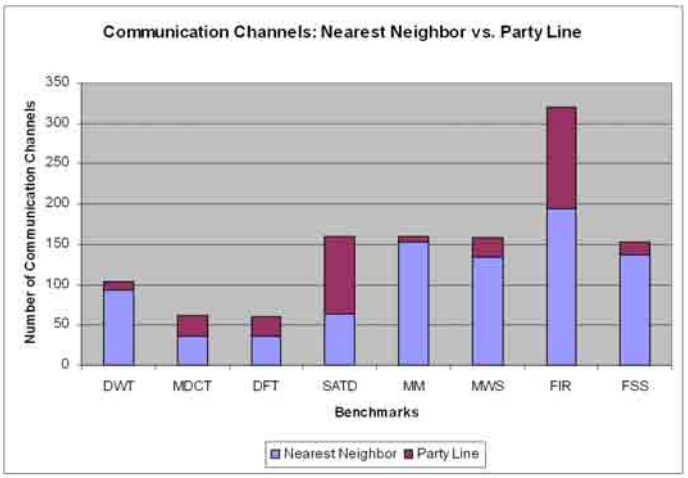


Fig. 7.11: Zero delay (NN) and non-zero delay (PL) communication channels.

path between two objects that must be routed by assigning routing resources such as LL registers and multiplexers.

All non-zero delay edges of TDFGs for each benchmark are successfully routed using the constraint satisfaction-based routing tool. As shown in fig. 7.12, the convergence time in most cases is found to be less than eight seconds with the exception of benchmark 7, the reasons for which are explained later in this discussion. Figure 7.13 shows the amount of memory required by the routing tool in each case and indicates a higher memory usage for longer search convergence time. Since the convergence times of benchmark 7 dominates in fig. 7.13, a rescaled view of the histogram is presented in fig. 7.14.

Figure 7.15 presents the distribution and backtracking steps associated with the routing of these benchmarks. In Chapter 6, it is mentioned that shorter routes are typically more resource constrained and hence are routed before longer routes. However, this approach results in the scenario where a shorter route has more flexibility than a longer route, but because the shorter route is handled first, it blocks resources critical for routing a longer path. In such a scenario, the Routing tool will eventually backtrack to rip-up and re-route the shorter route in order to make resources available for longer routes. In the worst case, if two routes depend on the same set of resources, a solution is not feasible and an alternate placement is sought by executing the Placement tool again. The increase in search

convergence time due to resource conflict is observed in the case of benchmark 7 which has the largest number of non-zero delay paths, as shown in fig. 7.11. Most of these paths are short paths with delay equal to one. The Routing tool takes the maximum amount of time and uses a large amount of memory in this case, as indicated in figs. 7.12 and 7.13, respectively. As shown in fig. 7.15, the Routing tool encounters maximum backtracking for benchmark 7 among all the test cases, which is the result of resource conflicts among short paths as explained above.

The task of the Routing tool is to assign LL registers and multiplexers to route each edge in a TDFG. Figure 7.16 provides the number of LL registers and multiplexers used for routing non-zero delay edges. Table 7.5 shows the percentage utilization of Party Line

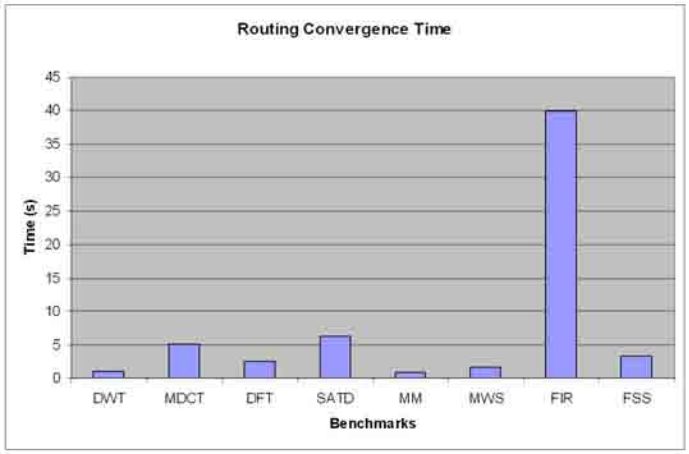


Fig. 7.12: Routing tool convergence time.

Table 7.5: Party line resource utilization for routing non-zero delay edges.

Id	**Benchmark**	**Launch/Land Registers (%)**	**Multiplexers (%)**	**Party Line Groups (Max 3)**
1	DWT	0.60	1.05	2
2	MDCT	2.65	4.95	3
3	DFT	1.60	2.85	3
4	SATD	4.80	7.85	3
5	MM	0.40	0.60	1
6	MWS	1.25	2.23	2
7	FIR	9.45	17.13	3
8	FSS	1.65	2.88	2

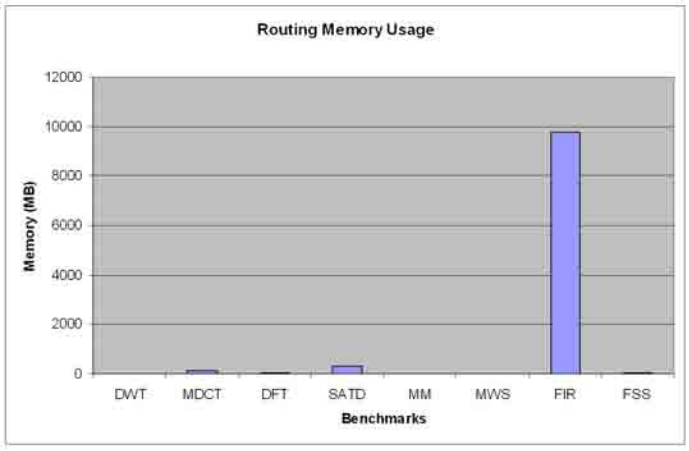


Fig. 7.13: Routing tool memory usage for all eight benchmarks.

routing resources available on an FPOA for each of these benchmarks. These numbers indicate a low utilization for PL communication, which is expected since the tool flow is biased towards NN communication in an attempt to minimize overall communication delay in the placed and routed design.

Benchmarks 2 and 3 present an interesting scenario because in spite of having comparable non-zero delay edges, results show that benchmark 3 is routed faster. This is because routing complexity not only depends on the number of non-zero delay edges, but also on the delay along each edge. Figure 7.2 shows that benchmark 2 has a higher average delay than benchmark 3, indicating that it is comprised of longer paths as compared to benchmark

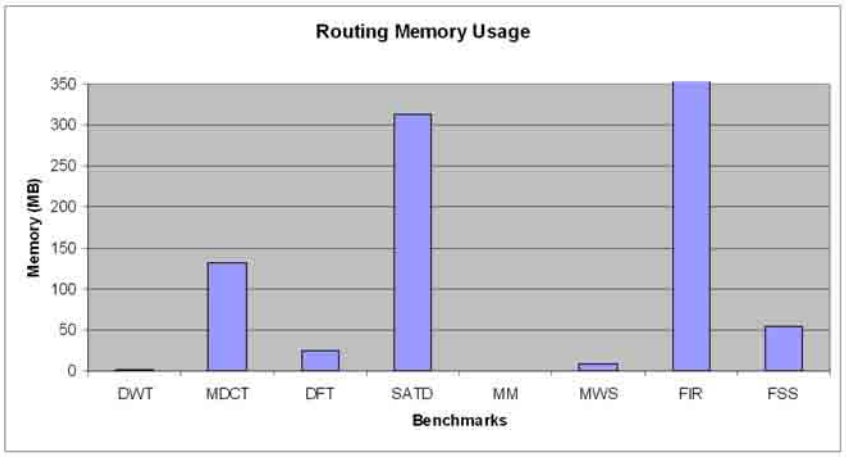


Fig. 7.14: A rescaled view of routing memory usage for all eight benchmarks.

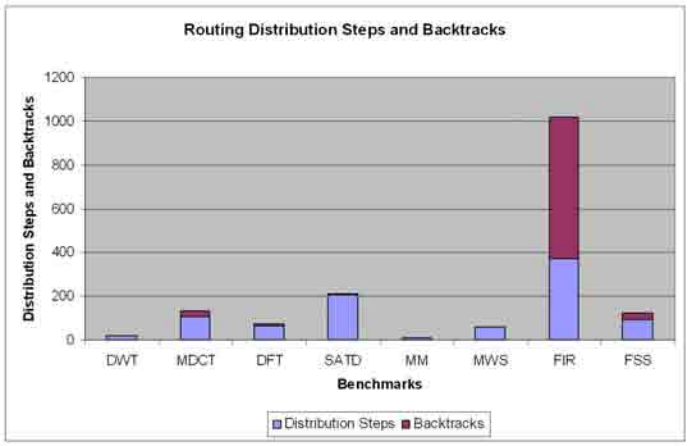


Fig. 7.15: Routing tool distribution and backtracks.

3. An edge with larger delay requires more routing resources than an edge with shorter delay, increasing the routing problem size. This characteristic is highlighted in fig. 7.16, which indicates that the number of routing resources needed by benchmark 3 is less than for benchmark 2.

The results obtained after routing each benchmark indicate that the Routing tool successfully routes all the paths. The convergence time and memory usage of the tool for routing these benchmarks is reported and depends on the number of non-zero delay paths, length of individual paths, and resource conflicts which require backtracking and re-routing. In particular, the Routing tool benefits from the minimization of communication delay which

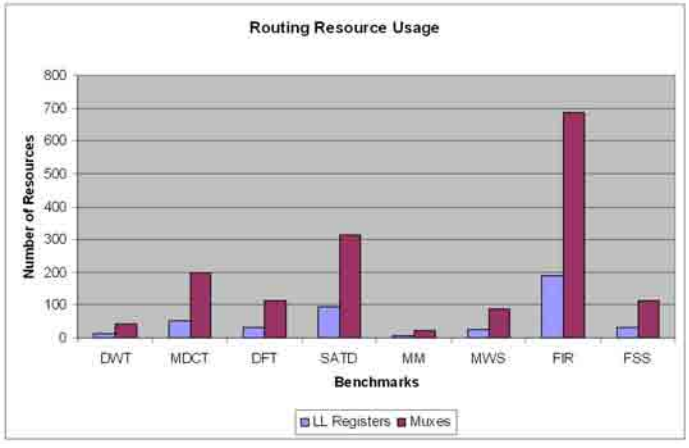


Fig. 7.16: Routing resource usage: Launch and land registers and multiplexers.

reduces the routing problem size by increasing zero-delay paths that are handled during the placement phase.

7.3 Tool Performance for Varying Problem Size

The FIR filter benchmark is chosen to study and analyze the effect of increasing problem size on tool performance. It is a scalable application and is the largest problem that is considered for evaluating the tool flow developed in this research. Moreover, this benchmark benefits from re-computation as well as the divide and conquer approach during scheduling and placement. For these reasons, it is an ideal candidate for evaluating tool behavior when problem size increases. Eight configurations, starting with an 8-tap FIR filter, are used. For placing larger than 32-tap FIR configurations, the divide and conquer approach is used, where the first 32-taps in the FIR configuration are placed first, followed by the remaining components. The performance of scheduling, placement, and routing tools is analyzed for each FIR configuration.

Figure 7.17 shows convergence time of all three tools for all eight FIR configurations, while the corresponding memory usage is presented in fig. 7.18. A monotonic increase in convergence time and memory usage is observed for each tool and for all configurations except for placement of the 8-tap FIR. The time and memory requirements for placing the

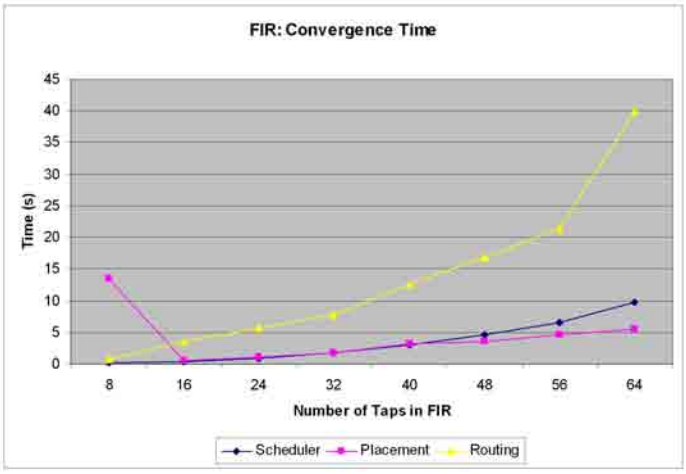


Fig. 7.17: Search convergence time for FIR configurations.

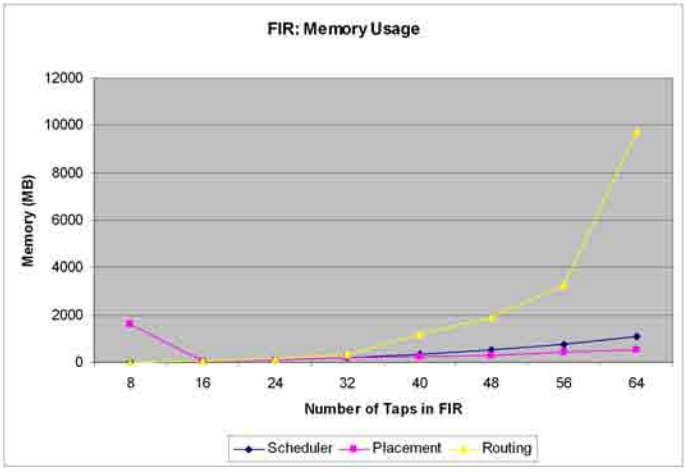
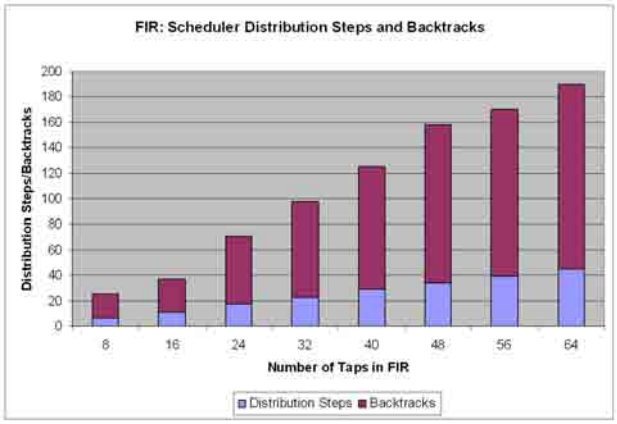


Fig. 7.18: Memory usage during scheduling, placement, and routing of FIR configurations.

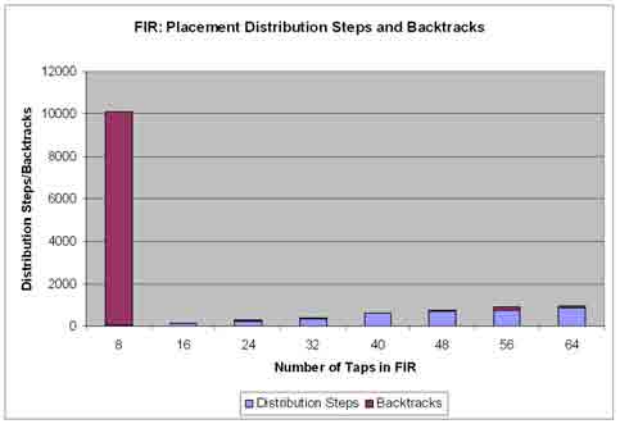
8-tap FIR configuration exceed the time and memory requirements for placing the 64-tap configuration.

Even though the 8-tap FIR filter is small in size, it has maximum placement flexibility, implying that most variables have large domain sizes. As mentioned earlier, variable selection affects search convergence and is worsened if much flexibility is offered to the variables. Consider the following scenario for a variable v with a small domain. Variable v is selected early in the search and is assigned a value which would eventually yield an invalid solution. However, not enough information is available at this point to backtrack until several remaining variables are grounded. Only after evaluating a large combination of variable assignments does the search backtracks and assigns a different value to v.

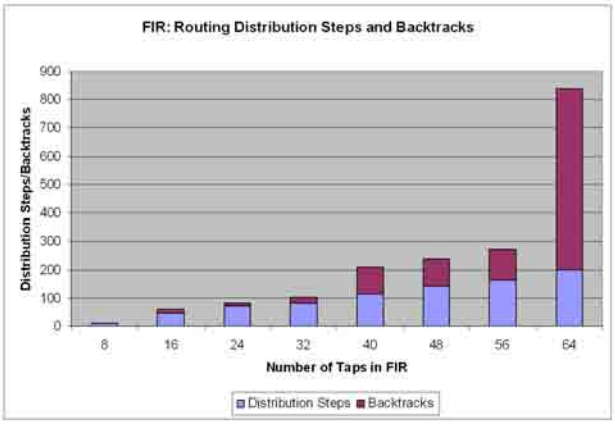
For both the 8-tap and the 16-tap FIR configurations, the size of bounding box is identical, but a smaller size offers more placement flexibility to the former case. Figure 7.19(b) shows extensive backtracking for the 8-tap FIR configuration which results in longer convergence time and higher memory requirement during placement. For the 16-tap to the 64-tap configurations, distribution steps increase monotonically during placement. Figures 7.19(a) and 7.19(c) illustrate a monotonic increase in distribution steps for all configurations. However, increased backtracking is observed during scheduling and routing with increments in problem size, which is expected because the bounding box grows in size as well. A steep



(a) Scheduling tool distribution steps and backtracks.



(b) Placement tool distribution steps and backtracks.



(c) Routing tool distribution steps and backtracks.

Fig. 7.19: Distribution steps and backtracks for FIR configurations during (a) scheduling, (b) placement, and (c) routing.

increase in backtracking is noticed during routing when moving from 56-tap to a 64-tap configuration. During routing, one of the newly added routes was ripped-up and re-routed several times before a valid route was obtained. As previously mentioned, such a scenario arises when a shorter route shares the set of routing resources with a longer route and the former blocks critical resources needed by the latter.

The number of nearest neighbor and party line communication channels for each configuration are given in fig. 7.20. In most cases, 35% less party line communication channels are used than nearest neighbor connections, which is inteneded by the aggressive scheduling tool. Finally, fig. 7.21 compares the number of routing resources used for routing non-zero delay routes using party lines for all eight configurations.

7.4 Tool Performance Beyond Arrix Architecture

In the previous sections, the performance of the scheduling, placement, and routing tools is evaluated using the FPOA Arrix architecture which contains 400 objects arranged in a 20×20 grid. In order to study and evaluate the performance of these tools as the size of the design and the underlying architecture is increased, the existing FPOA architecture is scaled up by a factor of 4. This new architecture contains 1600 objects which are arranged in a 40×40 grid. Due to the properties of the FIR benchmark, mentioned in the previous

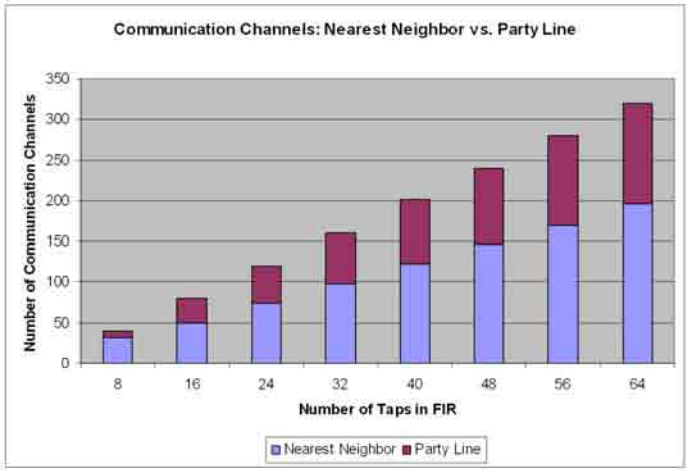


Fig. 7.20: Zero delay and non-zero delay communication channels for FIR configurations.

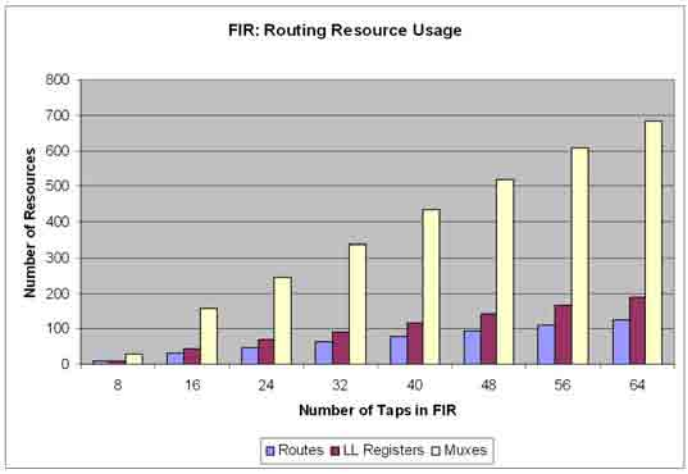


Fig. 7.21: Registers and multiplexers used for routing FIR configurations.

section, it is again considered for evaluating the scalability of the proposed approach. FIR configurations starting from a 32-tap filter are considered, and are increased in steps of 16-taps. Divide and conquer approach, as well as the bounding box approach are used for each configuration. The tools converge for up to 128-tap FIR filter configurations, after which all of them report failure due to memory limitations.

Figures 7.22 and 7.23 show the search convergence time and memory usage for the scheduling tool. An increase in convergence time is observed as problem size increases,

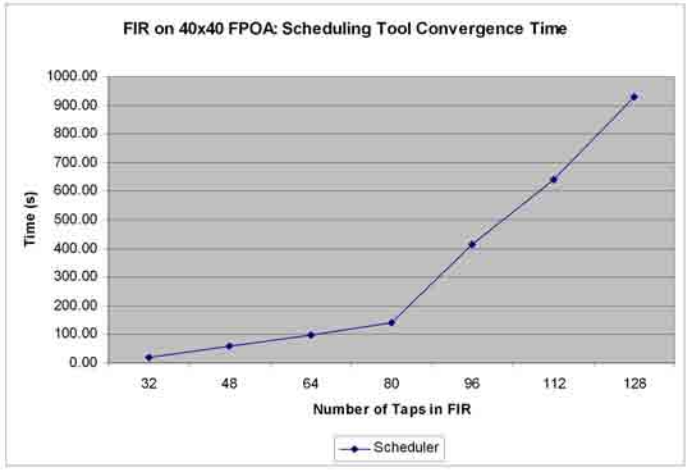


Fig. 7.22: Scheduling tool search convergence time for FIR configurations on a 40×40 FPOA.

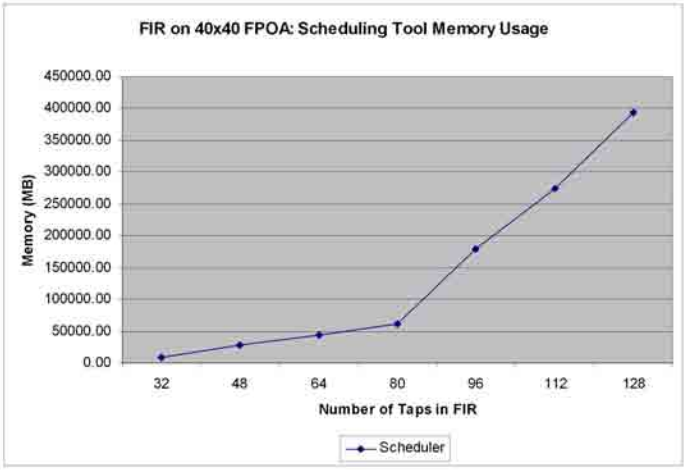


Fig. 7.23: Scheduling tool memory usage for FIR configurations on a 40×40 FPOA.

with a sharp increase observed after 80-tap FIR configuration. For the first three cases, a recomputation step of $S_{RC} = 20$ is used, but is increased to $S_{RC} = 25$, $S_{RC} = 50$, $S_{RC} = 80$, and $S_{RC} = 120$ steps for 80-tap, 96-tap, 112-tap, and 128-tap FIR configurations, respetively. The need for larger recomputation steps is indicative of the high demand for memory, as shown in fig. 7.23. At the same time, the effect of recomputation in reducing memory requirement is evident from fig. 7.23, making recomputation a promising candidate to improve the scalability of the scheduling tool.

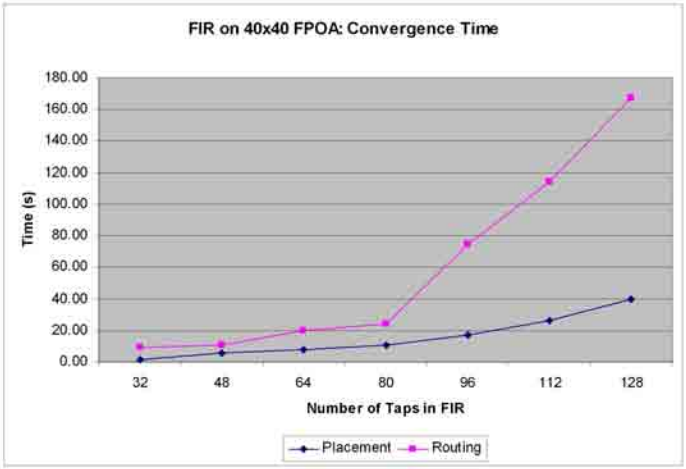


Fig. 7.24: Placement and Routing tool search convergence time for FIR configurations on a 40×40 FPOA.

Figure 7.24 shows the search convergence time for the placement and routing tools. A steep curve indicates a sharp rise in convergence time of the routing tool for FIR filter configurations larger than 80-taps. On the other hand, the placement tool scales better than the Routing and Scheduling tools, though a nonlinear increase in search convergence time indicates diminishing returns with increase in design size. The memory usage for the placement and routing tools is shown in fig. 7.25. The memory usage increases with problem size and a recomputation step of size $SRC = 3$ is required for routing the 128-tap FIR filter configuration. In addition to the recomputation, a bounding box approach, similar to the placement tool, is also applied to the routing tool to improve convergence.

The results presented above illustrate the fact that the proposed tools are not limited to a specific 20×20 FPOA Arrix architecture or for small designs. Instead, the proposed methodology can be applied to larger architectures and designs. Additionally, recomputation, bounding box, and divide and conquer approaches are potential candidates that can be explored further to improve scalability of the proposed tools.

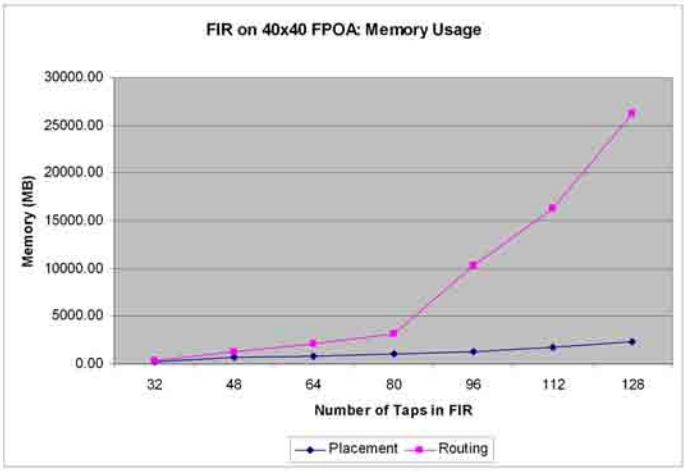


Fig. 7.25: Placement and Routing tool memory usage for FIR configurations on a 40×40 FPOA.

Chapter 8

Conclusions and Future Work

Scheduling, placement, and routing are important steps in VLSI design, whether a design is implemented as an ASIC, on an FPGA, or a CGRA. Literature indicates that a variety of techniques have been developed for ASIC and FPGA-based design. Most of these techniques attempt to minimize wirelength in order to optimize design performance.

The proposed research targets an FPOA, which is a coarse-grained reconfigurable computing device consisting of multiple processing elements. These elements communicate using a configurable communication network that takes a deterministic amount of time to send a signal over a fixed distance. A direct implication of this characteristic is that wirelength minimization is no longer the desired optimization objective. Instead, a scheduling tool determines the communication delay along the edges of an application's DFG. Resources are allocated to operations in the DFG and a placement tool must guarantee that two communicating resources must be placed such that the distance between them allows implementing a path with the desired delay. Similarly, the routing tool must route a path such that the physical connection supports the specified delay. In this research scheduling, placement, and routing tools have been developed to meet the above mentioned requirements.

This research explores the application of finite domain constraint satisfaction for developing scheduling, placement, and routing methodologies for an FPOA. An end-to-end tool chain has been implemented in the Mozart programming environment using the Oz and C++ programming languages. During each step in the tool flow, the problem is formulated using FD variables, and constraints are defined over these variables. FD variables denote attributes such as delay, time, location, etc. Relationships between these attributes, or restrictions on the values that these attributes can assume, are captured using FD constraints. Once the problem specification is complete, it is solved by an FD constraint solver through

propagation, distribution, and search.

The proposed tools have been evaluated using a set of eight benchmarks that are derived from multimedia, signal processing, and scientific applications. Analysis of data flow graphs for each of these benchmarks reveals that they represent scenarios with different problem sizes and inter-object communication. Each of the three tools converged in all eight test cases yielding a placeable schedule, a routable placement, and a routed design. Search convergence time and memory usage have been reported and analyzed for each benchmark at each step in the tool flow.

During the course of this research, the following observations were made.

1. Search convergence is dependent on variable selection strategy during distribution. However, efficient propagation and application of redundant constraints attempt to mitigate the effects of not selecting the most appropriate variable.

2. Aggressive minimization of delay during scheduling does not always generate placeable schedules. Typically, failure to place a design arises from geometrical limitations imposed by the FPOA architecture. Some of these restrictions are easily identified and can be handled in the scheduling phase, while other cases require detailed analysis because they arise from design requirements. A scheduler relaxation technique is implemented to identify and relax potential paths to improve placeability of a schedule.

3. Since ALU objects can support more than one operation, it is wise to merge operations on an ALU to increase resource utilization. Results indicate the advantages of this approach, such as accommodating designs with more than 256 ALU operations and reduction in inter-object connections. Experiments also revealed that aggressive merging of operations can lead to unplaceable schedules, primarily due to geometrical restrictions.

4. The divide and conquer approach enables the placement tool to handle large designs and drastically improves convergence time for large cases. A partial design is placed

first, and the remaining design is added later without repeating the entire placement process.

5. The scheduler attempts to minimize communication delay between two operations, generating schedules which use more NN connectivity. Results also show that most designs contain a higher number of NN connections than PL communication. More NN connections are usually indicative of a shorter schedule length. Furthermore, since NN connections are handled during placement, a larger ratio of NN vs. PL communication implies reduced computation burden during routing.

6. Routing can be performed in two ways: route all paths at once or route one path at a time. While the former approach is unbiased towards any path and exposes all available routing resources to all the paths in a design, it imposes severe memory requirements causing the search to terminate prematurely. The latter approach is a divide and conquer approach which selects the shortest path first and is found to converge in all test cases.

Evaluation of scheduling, placement, and routing tools indicates that these tools attempt to minimize schedule length, communication delay, utilize less chip area when possible, and route paths using minimal routing resources.

The primary contribution of this research is an end-to-end tool chain for scheduling, placing, and routing designs on an FPOA. Some of the possible directions that this research can investigate in future are specified below.

1. In its present form, the placement and routing methodology is applicable only to FPOAs. Even the resource allocation and scheduling algorithm is architecture dependent. Instead of focusing on a particular architecture, a broad category of architectures can be supported by developing a high-level abstract model for 2D mesh CGRAs. These design methodologies can be extended to target the abstract CGRA model. The tools can be configured for a particular architecture by supplying an architectural template.

2. The primary goal in the this research is to find a valid solution during each phase of the tool flow, along with minimization of delay, area, and routing resources. Additional objectives such as power and I/O data rate can be added to improve design performance. This approach necessitates a sound understanding of factors influencing power consumption on an FPOA, the relationship and interdependence among delay, area, routing resources, power, and I/O rate, and development of a formal model to capture these relationships.

3. An FPOA offers limited number of computational resources which do not permit a large design to be placed if the demand for resources exceeds the available capacity on the chip. One such scenario is the demand for more than 64 multipliers. To address this problem, a tool can be developed to perform a demand analysis prior to scheduling, and available ALUs can be used for implementing multiplication through repeated addition. A second scenario arises due to large designs which cannot be accomodated on the FPOA because the demand for one or more types of resources exceeds the FPOA's capacity. This issue can be solved in two ways: resource reuse and reconfiguration. If the design contains identical modules, then a module can be reused for different data sets by implementing necessary control logic. Alternatively, the design can be partitioned into smaller components and the FPOA can be repeatedly reconfigured with these components. In either of these cases, the scheduler needs to be modified to support resource reuse and to account for reconfiguration time.

4. Scheduling, placement, and routing are currently implemented as separate tools because combining them increases problem complexity manifold, hampering search convergence and imposing unsatisfiable memory demands. With further improvement in computing technology, by employing aggressive divide and conquer, or implementing multi-threaded constraint solvers capable of execution on multiple cores, the feasibility of a simultaneous scheduling, placement, and routing methodology can be explored.

5. Search convergence depends on many factors such as nature of DFGs and variable

selection. A search strategy may cause one search to converge quickly for one application, but may not show similar improvements in another case. Applying different strategies to a large set of applications followed by an analysis of search convergence times and memory usage can help categorize applications on the basis of search strategy giving maximum improvement in search performance. This information can later be used for profiling the input and automatically selecting the best search strategy. A similar profiling technique can also be implemented in schedule relaxer to perform minimal relaxation yet improve placeability.

6. In the last couple of decades, the research community has shown a growing interest in generating hardware from a high-level software description in programming language such as C. A C language to FPOA tool flow can be implemented based on the work presented in this dissertation by developing and integrating a C to DFG translator in the existing tool chain.

References

[1] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA: Kluwer Academic Publishers, 1999.

[2] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 2.0: An efficient analytical placer for mixed-mode designs," in *ASP-DAC '06: Proceedings of the 2006 Conference on Asia South Pacific Design Automation*, pp. 195–200, 2006.

[3] J. R. Gao, P. C. Wu, and T. C. Wang, "A new global router for modern designs," in *ASP-DAC '08: Proceedings of the 2008 Conference on Asia and South Pacific Design Automation*, pp. 232–237, 2008.

[4] M. Hanan, "On steiner's problem with rectilinear distance," *SIAM Journal on Applied Mathematics*, vol. 14, no. 2, pp. 255–265, Mar. 1966.

[5] "Xilinx XC2VP100 datasheet," http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf.

[6] D. Cherepacha and D. Lewis, "DP-FPGA: An FPGA architecture optimized for data paths," *VLSI Design*, vol. 4, no. 4, pp. 329–343, 1996.

[7] R. Francis, J. Rose, and Z. Vranesic, "Chortle-CRF: Fast technology mapping for lookup table-based FPGAs," in *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 227–233, 1991.

[8] D. D. Hill and N. S. Woo, "The benefits of flexibility in lookup table-based FPGAs," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 12, no. 2, pp. 349–353, 1993.

[9] R. W. Hartenstein and R. Kress, "A data path synthesis system for the reconfigurable data path architecture," in *ASP-DAC '95: Proceedings of the 1995 conference on Asia Pacific Design Automation*, p. 77, 1995.

[10] "D-Fabrix," `http://www.panasonic.co.uk/b2b/get/params_W0_MThtml/1715552/D-Fabrix.pdf`, July 2008.

[11] R. A. Bittner, P. M. Athanas, and M. D. Musgrove, "Colt: An experiment in wormhole run-time reconfiguration," in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic. SPIE*, pp. 187–194, 1996.

[12] R. Bittner and P. Athanas, "Wormhole run-time reconfiguration," in *FPGA '97: Proceedings of the 1997 Association for Computing Machinery Fifth International Symposium on Field-Programmable Gate Arrays*, pp. 79–85, 1997.

[13] E. Mirsky and A. DeHon, "MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157–166, 1996.

[14] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, p. 12, 1997.

[15] J. Faura, M. A. Aguirre, J. M. Moreno, P. V. Duong, and J. M. Insenser, "FIPSOC: A field programmable system on a chip," in *DCIS'97 XII Design of Circuits and Integrated Systems Conference*, pp. 597–602, Nov. 1997.

[16] J. Faura, C. Horton, P. van Duong, J. Madrenas, M. Aguirre, and J. Inserser, "A novel mixed signal programmable device with on-chip microprocessor," in *Proceedings of the IEEE 1997 Custom Integrated Circuits Conference*, pp. 103–106, May 1997.

[17] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, Sept. 1997.

[18] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J. W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[19] M. B. Taylor, "Design decisions in the implementation of a raw architecture workstation," Master's thesis, Massachusetts Institute of Technology, Boston, MA, Sept. 1999.

[20] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor," in *IEICE Transactions on Information and Systems E82-D*, pp. 389–397, 1998.

[21] Y. Kang, J. Torrellas, and T. Huang, "An IRAM architecture for image analysis and pattern recognition," in *Fourteenth International Conference on Pattern Recognition*, vol. 2, pp. 1561–1564, Aug. 1998.

[22] Y. Kang, W. Huang, S. M. Yoo, D. Keen, Z. Ce, V. Lain, P. Pattnaik, and J. Torrellas, "FlexRAM: Toward an advanced intelligent memory system," in *ICCD '99: Proceedings of the 1999 IEEE International Conference on Computer Design*, p. 192, 1999.

[23] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pp. 135–143, 1999.

[24] H. Singh, M. Lee, G. Lu, F. Kurdahi, and N. Bagherzadeh, "MorphoSys: A reconfigurable architecture for multimedia applications," in *Brazilian Symposium on Integrated Circuit Design and System Design*, p. 134, 1998.

[25] G. Lu, H. Singh, M. H. Lee, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, "The MorphoSys dynamically reconfigurable system-on-chip," in *EH*

'99: Proceedings of the 1st NASA/DOD Workshop on Evolvable Hardware, p. 152, 1999.

[26] H. Singh, G. Lu, E. Filho, R. Maestre, M. H. Lee, F. Kurdahi, and N. Bagherzadeh, "MorphoSys: case study of a reconfigurable computing system targeting multimedia applications," in *DAC '00: Proceedings of the 37th Conference on Design Automation*, pp. 573–578, 2000.

[27] M. H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, "Design and implementation of the MorphoSys reconfigurable computingprocessor," *Journal of VLSI Signal Processing Systems*, vol. 24, no. 2-3, pp. 147–164, 2000.

[28] A. Alsolaim, J. Starzyk, J. Becker, and M. Glesner, "Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 205, 2000.

[29] J. Granacki and M. Vahey, "MONARCH: A morphable networked microarchitecture," in *High Performance Embedded Computing Workshop*, Oct. 2002.

[30] R. Z. Bhatti, J. Draper, and C. Steele, "PBuf: An on-chip packet transfer engine for MONARCH," in *49th IEEE International Midwest Symposium on Circuits and Systems*, vol. 2, pp. 531–535, Aug. 2006.

[31] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, vol. 2778, pp. 61–70, Sept. 2003.

[32] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins, "Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study," in *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, p. 21224, 2004.

[33] "Mathstar ™ Arrix ™ Family FPOA ™ Architecture Guide," http://www.mathstar.com/Architecture.php, May 2007.

[34] "Clearspeed whitepaper: CSX processor architecture," `http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07v2.pdf`.

[35] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano, "Performance improvement methodology for clearspeed's CSX600," in *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, p. 77, 2007.

[36] R. Verma and A. Akoglu, "A coarse-grained and hybrid reconfigurable architecture with flexible noc router for variable block size motion estimation," in *IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, Apr. 2008.

[37] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions Circuits Systems Video Technology*, vol. 13, pp. 560–574, July 2003.

[38] T. Wiegand and G. J. Sullivan, "The H.264/AVC video coding standard," *IEEE Signal Processing Magazine*, vol. 24, no. 2, Mar. 2007.

[39] S. Kelem, B. Box, S. Wasson, R. Plunkett, J. Hassoun, and C. Phillips, "An elemental computing architecture for SD radio," in *SDR Forum Technical Conference*, Denver, CO, Nov. 2007.

[40] "Tilera TILEPro36 Product Brief," http://www.tilera.com/pdf/ProductBrief_TILEPro36_Web_v1.pdf.

[41] "Tilera TILEPro64 Product Brief," http://www.tilera.com/pdf/ProductBrief_TILEPro64_Web_v2.pdf.

[42] "Tilera TILE-Gx Product Brief," http://www.tilera.com/pdf/ProductBrief_TILE-Gx_v2.pdf.

[43] "Tilera Tile Processor Architecture Product Brief," http://www.tilera.com/pdf/ProductBrief_TileArchitecture_Web_v4.pdf.

[44] C. Liang and X. Huang, "Smartcell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications," *EURASIP Journal on Embedded Systems*, p. 15, 2009.

[45] G. B. Dantzig, *Programming in a Linear Structure*. Washington, DC: Wiley-Blackwell, 1949.

[46] G. B. Dantzig, *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press, 1963.

[47] A. Schrijver, *Theory of Linear and Integer Programming*. New York: John Wiley & Sons Ltd, 1986.

[48] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA: A platform and programming language independent interface for search algorithms," in *Evolutionary Multi-Criterion Optimization*, pp. 494–508, 2003.

[49] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[50] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *Journal of Logic Programming*, vol. 20, pp. 503–581, 1994.

[51] K. Marriott and P. Stuckey, *Programming with Constraints: An Introduction*. Cambridge, MA: MIT Press, 1998.

[52] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 355–383, 2003.

[53] K. Schild and J. Wurtz, "Off-line scheduling of a real-time system," in *ACM Symposium on Applied Computing*, pp. 29–38. Atlanta, GA: ACM Press, 1998.

[54] K. Kuchcinski, "Constraints-driven design space exploration for distributed embedded systems," *Journal of Systems Architecture*, vol. 47, no. 3-4, pp. 241–261, 2001.

[55] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," *ACM Computer Survey*, vol. 23, no. 2, pp. 143–220, 1991.

[56] M. Pan and C. Chu, "FastRoute 2.0: A high-quality and efficient global router," in *ASP-DAC '07: Proceedings of the 2007 Conference on Asia and South Pacific Design Automation*, pp. 250–255, Jan. 2007.

[57] M. Sarrafzadeh, M. Wang, and X. Yang, *Modern Placement Techniques*. Norwell, MA: Kluwer Academic Publishers, 2003.

[58] C. Chu, "FLUTE: Fast lookup table based wirelength estimation technique," in *ICCAD '04: Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, pp. 696–701, 2004.

[59] M. Wang, X. Yang, and M. Sarrafzadeh, "Dragon2000: standard-cell placement tool for large industry circuits," in *ICCAD '00: Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided design*, pp. 260–263, 2000.

[60] J. A. Roy, J. F. Lu, and I. L. Markov, "Seeing the forest and the trees: Steiner wirelength optimization in placemen," in *ISPD '06: Proceedings of the 2006 International Symposium on Physical Design*, pp. 78–85, 2006.

[61] B. Krishna, C. R. Chen, and N. K. Sehgal, "A novel ultra-fast heuristic for VLSI CAD steiner trees," in *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pp. 192–197, 2003.

[62] M. A. Breuer, "A class of min-cut placement algorithms," in *DAC '77: Proceedings of the 14th conference on Design Automation*, pp. 284–290, 1977.

[63] U. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation," in *DAC '79: Proceedings of the 16th Conference on Design Automation*, pp. 1–10, 1979.

[64] T. C. Chen, T. C. Hsu, Z. W. Jiang, and Y. W. Chang, "NTUplace: A ratio partitioning based placement algorithm for large-scale mixed-size designs," in *ISPD '05: Proceedings of the 2005 International Symposium on Physical Design*, pp. 236–238, 2005.

[65] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Can recursive bisection alone produce routable placements," in *Proceedings of Design Automation Conference*, pp. 477–482, 2000.

[66] A. Kahng and S. Reda, "Wirelength minimization for min-cut placements via placement feedback," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1301–1312, July 2006.

[67] M. C. Yildiz and P. H. Madden, "Global objectives for standard cell placement," in *GLSVLSI '01: Proceedings of the 11th Great Lakes Symposium on VLSI*, pp. 68–72, 2001.

[68] A. R. Agnihotri, S. Ono, and P. H. Madden, "Recursive bisection placement: Feng shui 5.0 implementation details," in *ISPD '05: Proceedings of the 2005 International Symposium on Physical Design*, pp. 230–232, 2005.

[69] M. Can Yildiz and P. H. Madden, "Improved cut sequences for partitioning based placement," in *DAC '01: Proceedings of the 38th Conference on Design Automation*, pp. 776–779, 2001.

[70] A. Agnihotri, M. C. Yildiz, A. Khatkhate, A. Mathur, S. Ono, and P. H. Madden, "Fractional cut: Improved recursive bisection placement," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM International Conference on Computer Aided Design*, p. 307, 2003.

[71] C. Sechen and A. Sangiovanni Vincentelli, "The timberwolf placement and routing package," *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, pp. 510–522, Apr 1985.

[72] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[73] N. Viswanathan and C.-N. Chu, "FastPlace: Efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 5, pp. 722–733, May 2005.

[74] N. Viswanathan, M. Pan, and C. Chu, "Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific Design Automation*, pp. 135–140, 2007.

[75] C. J. Alpert, T. Chan, D. J. H. Huang, I. Markov, and K. Yan, "Quadratic placement revisited," in *DAC '97: Proceedings of the 34th Annual Conference on Design Automation*, pp. 752–757, 1997.

[76] M. Pan and C. Chu, "FastRoute: A step to integrate global routing into placement," in *ICCAD '06: Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design*, pp. 464–471, 2006.

[77] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: global router with efficient via minimization," in *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pp. 576–581, 2009.

[78] M. Pan and C. Chu, "IPR: an integrated placement and routing algorithm," in *DAC '07: Proceedings of the 44th Annual Conference on Design Automation*, pp. 59–62, 2007.

[79] M. Pan, N. Viswanathan, and C. Chu, "An efficient and effective detailed placement algorithm," in *ICCAD '05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, pp. 48–55, 2005.

[80] L. McMurchie and C. Ebeling, "Pathfinder: a negotiation-based performance-driven router for FPGAs," in *FPGA '95: Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, pp. 111–117, 1995.

[81] Y. J. Chang, Y. T. Lee, and T. C. Wang, "NTHU-Route 2.0: A fast and stable global router," in *IEEE/ACM International Conference on Computer Aided Design*, pp. 338–343, Nov. 2008.

[82] "The ISPD98 circuit benchmark suite," http://vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html.

[83] "Standard cell benchmark circuits from the Microelectronics Center of North Carolina (MCNC)," http://vlsicad.cs.binghamton.edu/gz/PDWorkshop91.tgz.

[84] W. Donath, "Placement and average interconnection lengths of computer logic," *IEEE Transactions on Circuits and Systems*, vol. 26, no. 4, pp. 272–277, Apr 1979.

[85] G. Parthasarathy, M. Marek-Sadowska, A. Mukherjee, and A. Singh, "Interconnect complexity-aware FPGA placement using rent's rule," in *SLIP '01: Proceedings of the 2001 International Workshop on System-level Interconnect Prediction*, pp. 115–121, 2001.

[86] X. Yang, R. Kastner, and M. Sarrafzadeh, "Congestion estimation during top-down placement," in *ISPD '01: Proceedings of the 2001 International Symposium on Physical Design*, pp. 164–169, 2001.

[87] A. Singh, G. Parthasarathy, and M. Marek-Sadowska, "Interconnect resource-aware placement for hierarchical FPGAs," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer Aided Design*, pp. 132–136, 2001.

[88] N. W. Eum, T. Kim, and C. M. Kyung, "A router for symmetrical FPGAs based on exact routing density evaluation," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer Aided Design*, pp. 137–143, 2001.

[89] P. Kannan, S. Balachandran, and D. Bhatia, "On metrics for comparing routability estimation methods for FPGAs," in *DAC '02: Proceedings of the 39th Conference on Design Automation*, pp. 70–75, 2002.

[90] P. Kannan, S. Balachandran, and D. Bhatia, "fGREP - fast generic routing demand estimation for placed FPGA circuits," in *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pp. 37–47, 2001.

[91] C. L. E. Cheng, "RISA: Accurate and efficient placement routability modeling," in *ICCAD '94: Proceedings of the 1994 IEEE/ACM International Conference on Computer Aided Design*, pp. 690–695, 1994.

[92] J. Lou, S. Krishnamoorthy, and H. S. Sheng, "Estimating routing congestion using probabilistic analysis," in *ISPD '01: Proceedings of the 2001 International Symposium on Physical Design*, pp. 112–117, 2001.

[93] D. Jariwala and J. Lillis, "On interactions between routing and detailed placement," in *ICCAD '04: Proceedings of the 2004 IEEE/ACM International Conference on Computer Aided Design*, pp. 387–393, 2004.

[94] G. G. Lemieux and S. D. Brown, "A detailed routing algorithm for allocating wire segments in field-programmable gate arrays," in *Proceedings of ACM/SIGDA Physical Design Workshop, Lake Arrowhead, CA*, pp. 215–226, 1993.

[95] S. K. Nag and R. A. Rutenbar, "Performance-driven simultaneous place and route for island-style FPGAs," in *ICCAD '95: Proceedings of the 1995 IEEE/ACM International Conference on Computer Aided Design*, pp. 332–338, 1995.

[96] Y. S. Lee and A. H. Wu, "A performance and routability-driven router for FPGAs considering path delays," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 2, pp. 179–185, Feb. 1997.

[97] M. J. Alexander, J. P. Cohoon, J. L. Ganley, and G. Robins, "Performance-oriented placement and routing for field-programmable gate arrays," in *EURO-DAC '95/EURO-VHDL '95: Proceedings of the Conference on European Design Automation*, pp. 80–85, 1995.

[98] M. J. Alexander and G. Robins, "New performance-driven fpga routing algorithms," in *DAC '95: Proceedings of the 32nd ACM/IEEE Conference on Design Automation*, pp. 562–567, 1995.

[99] S. K. Rao, P. Sadayappan, F. K. Hwang, and P. W. Shor, "The rectilinear steiner arborescence problem," *Algorithmica*, vol. 7, no. 1-6, pp. 277–288, June 1992.

[100] W. Shi and C. Su, "The rectilinear steiner arborescence problem is NP-complete," in *SODA '00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 780–787, 2000.

[101] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, pp. 213–222, 1997.

[102] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 2, pp. 364–365, 1961.

[103] A. Sharma, C. Ebeling, and S. Hauck, "Architecture-adaptive routability-driven placement for FPGAs," in *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications*, pp. 427–432, Aug. 2005.

[104] M. V. da Silva, R. Ferreira, A. Garcia, and J. M. P. Cardoso, "Mesh mapping exploration for coarse-grained reconfigurable array architectures," *IEEE International Conference on Reconfigurable Computing and FPGA's*, pp. 1–10, Sept. 2006.

[105] R. Ferreira, A. Garcia, T. Teixeira, and J. Cardoso, "A polynomial placement algorithm for data driven coarse-grained reconfigurable architectures," *IEEE Computer Society Annual Symposium on VLSI*, pp. 61–66, Mar. 2007.

[106] Y. T. Lai, H. Y. Lai, and C. N. Yeh, "Placement for the reconfigurable data path architecture," *IEEE International Symposium on Circuits and Systems*, vol. 2, pp. 1875–1878, May 2005.

[107] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, "Logic emulation with virtual wires," *IEEE Transactions on Computer Aided Design*, vol. 16, pp. 609–626, 1997.

[108] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb, "TIERS: Topology independent pipelined routing and scheduling for virtualwire compilation," in *FPGA '95: Proceedings of the 1995 ACM Third International Symposium on Field-programmable Gate Arrays*, pp. 25–31, 1995.

[109] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams," in *ISCA'04: Proceedings of the 2004 International Symposium on Computer Architecture*, pp. 2–13, 2004.

[110] W. O. Fung, T. Arslan, and S. Khawam, "Genetic algorithm based engine for domain-specific reconfigurable arrays," in *AHS '06: Proceedings of the first NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 200–206, 2006.

[111] http://www.mozart-oz.org/, May 2003.

[112] J.-C. Régin, "Generalized arc consistency for global cardinality constraint," in *Association for the Advancement of Artificial Intelligence/Innovative Applications of Artificial Intelligence Conference*, vol. 1, pp. 209–215, 1996.

[113] "GNU scientific library," http://www.gnu.org/software/gsl/.

[114] "LAME MP3 Encoder," http://lame.sourceforge.net/.

[115] "Mediabench II," http://euler.slu.edu/ fritts/mediabench/.

[116] R. Saraswat and B. Eames, "On the use of DesertFD to generate custom architectures for H.264 motion estimation," in *15th IEEE International Conference on Engineering of Computer Based Systems*, pp. 359–368, 2008.

[117] V. K. Kodavalla, "IP gate count estimation methodology during micro-architecture phase," in *IP based Electronic System Conference and Exhibition*, pp. 55–60, 2007.

Appendices

Appendix A

MaxEightALU Constraint Implementation in C++

A.1 MaxEightALU Header File

```
//MaxEightALU_FD.h - Header File for MaxEightALU constraint
#include "mozart_cpi.hh"
#include <stdio.h>
#include <map>
#include <set>
#include <iostream>

using namespace std;

class MaxEightALU : public OZ_Propagator
{
   friend OZ_C_proc_interface *oz_init_module(void);
   private:
      int _tup_size;
      static OZ_PropagatorProfile profile;
      OZ_Term *_tup, _max;
   public:
      MaxEightALU(OZ_Term a, OZ_Term b) :
                  _tup_size (OZ_vectorSize(a)), _max(b)
      {
         _tup = OZ_hallocOzTerms(_tup_size);
```

```
        OZ_getOzTermVector(a, _tup);
    }
    virtual OZ_Return propagate(void);
    virtual size_t sizeOf(void)
    {
        return sizeof(MaxEightALU);
    }
    virtual void gCollect(void)
    {
        _tup = OZ_gCollectAllocBlock(_tup_size, _tup);
        OZ_gCollectTerm(_max);
    }
    virtual void sClone(void)
    {
        _tup = OZ_sCloneAllocBlock(_tup_size, _tup);
        OZ_sCloneTerm(_max);
    }
    virtual OZ_Term getParameters(void) const;
    virtual OZ_PropagatorProfile *getProfile(void) const
    {
        return &profile;
    }
};    //class MaxEightALU

class Iterator_OZ_FDIntVar
{
  private:
    int _l_size;
```

```
    OZ_FDIntVar * _l;
  public:
    Iterator_OZ_FDIntVar(int s, OZ_FDIntVar * l) : _l_size(s), _l(l) {}
    OZ_Boolean leave(void)
    {
      OZ_Boolean vars_left = OZ_FALSE;
      for (int i = _l_size; i--; )
      vars_left |= _l[i].leave();
      return vars_left;
    }
    void fail(void)
    {
      for (int i = _l_size; i--; _l[i].fail());
    }
};   //class Iterator_OZ_FDIntVar

class ExtendedExpect : public OZ_Expect
{
  private:
    OZ_expect_t _expectIntVarAny(OZ_Term t)
    {
      return expectIntVar(t, fd_prop_any);
    }
  public:
    OZ_expect_t expectIntVarSingl(OZ_Term t)
    {
      return expectIntVar(t, fd_prop_singl);
    }
```

```
    OZ_expect_t expectVectorIntVarAny(OZ_Term t)
    {
      return expectVector(t,(OZ_ExpectMeth) &ExtendedExpect::_expectIntVarAny);
    }
    OZ_expect_t expectVectorIntVarSingl(OZ_Term t)
    {
      return expectVector(t, (OZ_ExpectMeth) &ExtendedExpect::expectIntVarSingl);
    }
};  //class ExtendedExpect
```

A.2 MaxEightALU Source Code

```
//MaxEightALU.CPP - Source Code File for MaxEightALU constraint
#include "MaxEightALU_FD.h"
#define FailOnEmpty(X) if((X) == 0) goto failure;

OZ_PropagatorProfile MaxEightALU::profile;
OZ_BI_proto(fd_MaxEightALU);
OZ_C_proc_interface *oz_init_module(void)
{
  static OZ_C_proc_interface i_table[] = {
    {"MaxEightALU", 2, 0, fd_MaxEightALU}, {0,0,0,0}
    };
  MaxEightALU::profile = "MaxEightALU prop";
  return i_table;
}
OZ_Term MaxEightALU::getParameters(void) const
{
  OZ_Term list = OZ_nil();
```

```
    for (int i = _tup_size; i–; )
            list = OZ_cons(_tup[i], list);
    return OZ_cons(_max, OZ_cons(list, OZ_nil()));
}
OZ_Return MaxEightALU::propagate(void)
{
    if (_tup_size == 0)
        return OZ_ENTAILED;
    Declare variables corresponding to OZ Variables
    OZ_FDIntVar oz_tup[_tup_size];
    OZ_FDIntVar maxallowedin(_max);
    Iterator_OZ_FDIntVar tupItr(_tup_size, oz_tup);
    OZ_FiniteDomain oz_tup_aux[_tup_size], oz_grounded_aux(fd_empty),
            oz_ungrounded_aux(fd_empty);
    bool NotFailed = true; //true means not failed
    int *grounded_var_table = new int[_tup_size];
    map<int, int> RepeatedElemMap;
    set<int> ReachedMaxAllowed;
    map<int, int>::iterator RepeatedElemMap_itr;
    int maxallowed = maxallowedin->getSingleElem();

    for (int n=0; n < _tup_size ; n++)
    {
        oz_tup[n].read(_tup[n]);
        oz_tup_aux[n].initEmpty();
        if (NotFailed)
        {
            int elem = oz_tup[n]->getMinElem();
```

```
        if (oz_tup[n]->getSize() == 1)
        {
            grounded_var_table[n] =1;
            RepeatedElemMap_itr = RepeatedElemMap.find(elem);
            if (RepeatedElemMap_itr == RepeatedElemMap.end())
            {
                RepeatedElemMap.insert(pair<int,int>(elem, 1));
            }
            else if (RepeatedElemMap_itr->second < maxallowed)
            {
                int numrep = RepeatedElemMap_itr->second + 1;
                RepeatedElemMap.erase(RepeatedElemMap_itr);
                RepeatedElemMap.insert(pair<int, int>(elem, numrep));
                if (numrep ==maxallowed)
                {
                    oz_grounded_aux += elem;
                    ReachedMaxAllowed.insert(elem);
                }
            }
            else //Propagator failed
            {
                RepeatedElemMap.clear();
                ReachedMaxAllowed.clear();
                NotFailed = false;
            }
        }
    }
}
```

```
FailOnEmpty(NotFailed); //If Propagator Fails, report failure
for (int i=0; i < _tup_size; i++)
{
   if (grounded_var_table[i] != 1)
   {
      FailOnEmpty(*oz_tup[i] -= oz_grounded_aux);
      if (oz_tup[i]->getSize() == 0)
         NotFailed = false;
      else
         if (oz_tup[i]->getSize() == 1)
         {
            int elem = oz_tup[i]->getSingleElem();
            RepeatedElemMap_itr = RepeatedElemMap.find(elem);
            int numrep = RepeatedElemMap_itr->second + 1;
            if (numrep <= maxallowed)
            {
               RepeatedElemMap.erase(RepeatedElemMap_itr);
               RepeatedElemMap.insert(pair<int,int>(elem, numrep));
               if (numrep == maxallowed)
               {
                  oz_grounded_aux += elem;
                  ReachedMaxAllowed.insert(elem);
               }
            }
            else
               NotFailed = false;
         }
   }
```

```
    }
    FailOnEmpty(NotFailed);
    return (tupItr.leave() | maxallowedin.leave())
        ? OZ_SLEEP : OZ_ENTAILED;
    failure:
        delete [] grounded_var_table;
        RepeatedElemMap.clear();
        ReachedMaxAllowed.clear();
        tupItr.fail();
        maxallowedin.fail();
        return OZ_FAILED;
}
OZ_BI_define(fd_MaxEightALU, 2, 0)
{
    OZ_EXPECTED_TYPE(OZ_EM_VECT","OZ_EM_FD);
    ExtendedExpect pe;
    OZ_EXPECT(pe, 0, expectVectorIntVarAny);
    OZ_EXPECT(pe, 1, expectIntVar);
    return pe.impose(new MaxEightALU(OZ_in(0), OZ_in(1)));
}
OZ_BI_end
```

Appendix B

NAND Gate Count and Execution Latency of Benchmarks

The benchmarks used for evaluating the proposed methodology can also be implemented as ASICs. This section provides a comparison between the sizes of these benchmarks when implemented on an FPOA vs. an ASIC implementation. The model presented by Kodavalla [117] is used for estimating the gate count for each of the benchmarks. Table B.1 presents the number of FPOA objects, the estimated NAND gate count for a functionally equivalent implementation, and the execution latency of the benchmarks.

Table B.1: Number of objects, NAND gate count, and execution latency of benchmarks.

Id	**Benchmark**	**Number of objects**	**NAND gate count**	**Latency (clock cycles)**
1	Discrete Wavelet Transform	105	104782	22
2	Modified Discrete Cosine Transform	53	47562	11
3	Discrete Fourier Transform	48	32952	9
4	Sum of Absolute Transformed Difference	144	18000	18
5	Matrix Multiplication	168	167512	10
6	MP3 Window Subband	163	167244	67
7	Finite Impulse Response filter	321	183840	64
8	Five Step Search	153	36250	41

Appendix C

Layout of Benchmarks

The layouts of all eight benchmarks are presented in this section. The number of different types of operations in each benchmark are given in Table C.1. Figure C.1 shows the legend used for representing the layouts. Figure C.2 illustrates the layout of the DWT benchmark. The layouts of the MDCT benchmark and the DFT benchmark are shown in figs. C.3 and C.4, respectively. Both these benchmarks show the effect of applying a bounding box, which limits the placement to a small region. Figure C.5 presents the layout of the SATD benchmark. The MM, MWS, and FIR benchmarks utilize all the 64 MACs which is evident in their layouts shown in figs. C.6, C.7, and C.8, respectively. Figure C.9 presents the layout of the FSS benchmark.

Table C.1: Number of different types of operations in a benchmark.

Id	**Benchmark**	**Number of ALUs**	**Number of MACs**	**Number of RFs**
1	Discrete Wavelet Transform	23	40	42
2	Modified Discrete Cosine Transform	13	18	22
3	Discrete Fourier Transform	24	12	12
4	Sum of Absolute Transformed Difference	144	0	0
5	Matrix Multiplication	32	64	72
6	MP3 Window Subband	35	64	64
7	Finite Impulse Response filter	192	64	65
8	Five Step Search	153	0	0

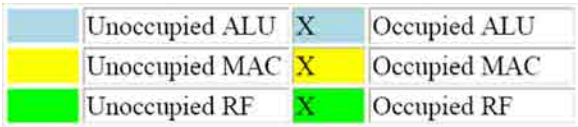


Fig. C.1: Legend for benchmark layouts.

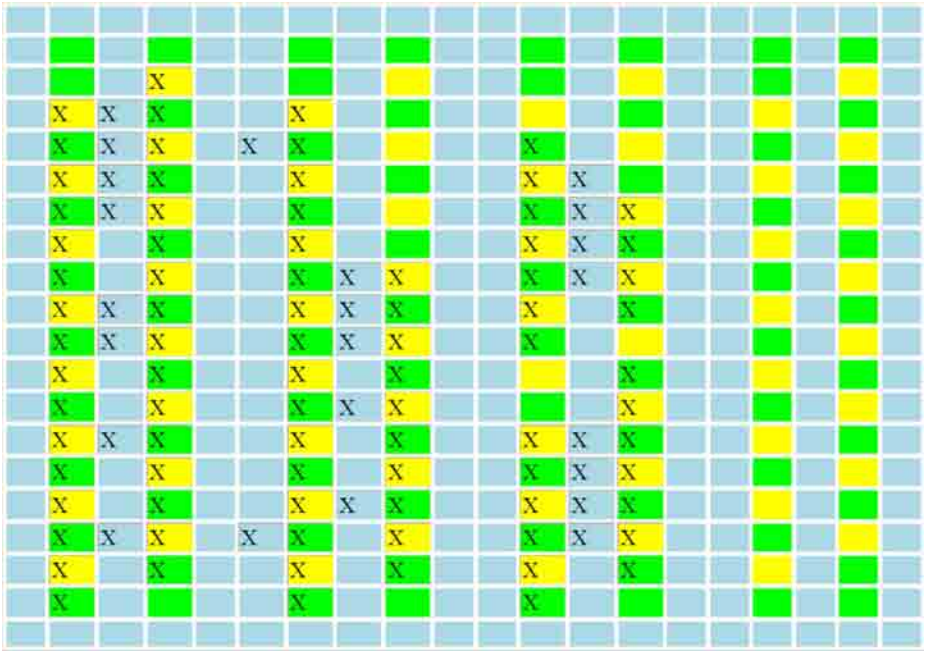


Fig. C.2: Layout of DWT benchmark.

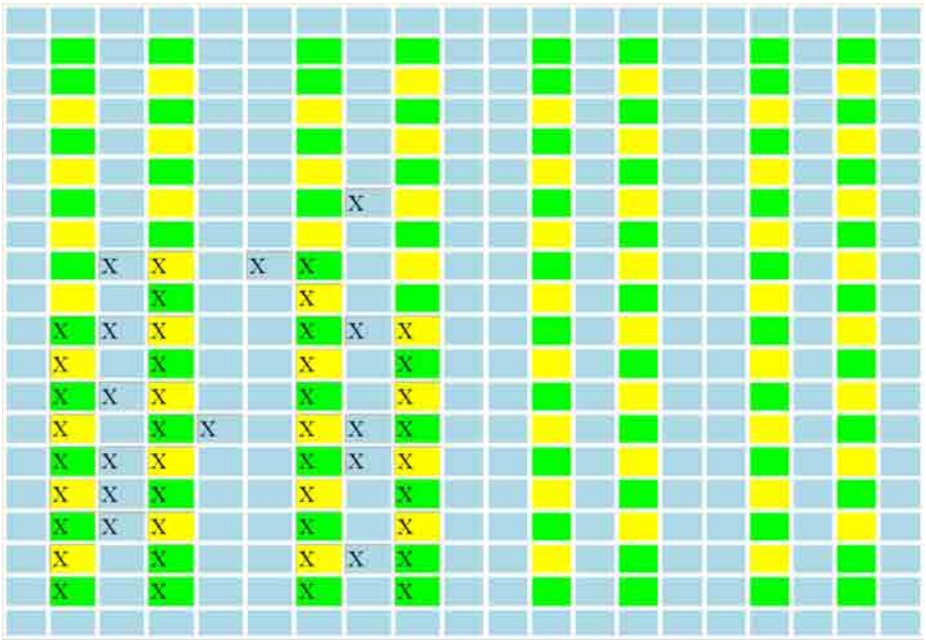


Fig. C.3: Layout of MDCT benchmark.

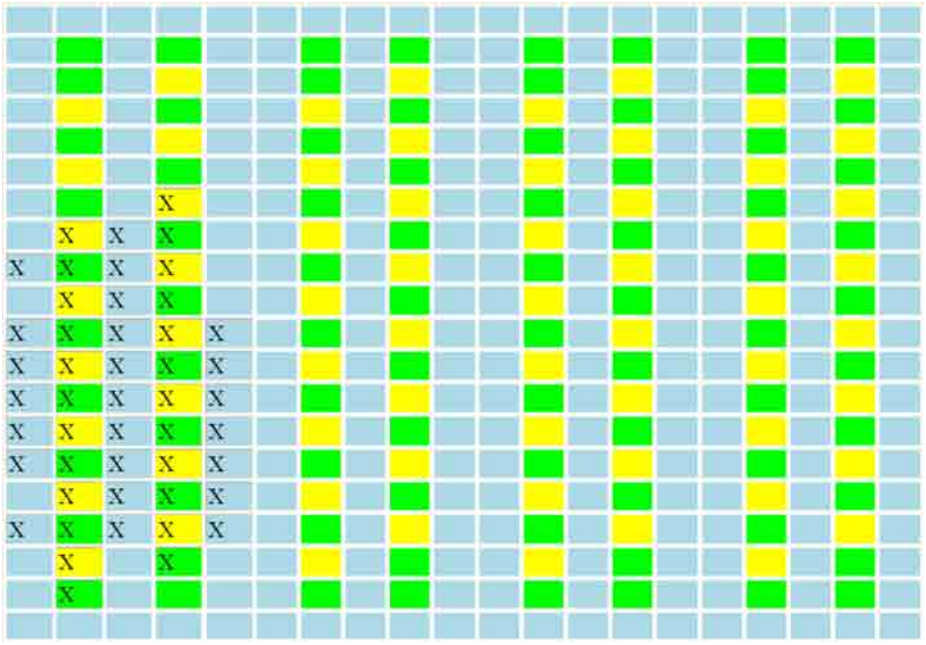


Fig. C.4: Layout of DFT benchmark.

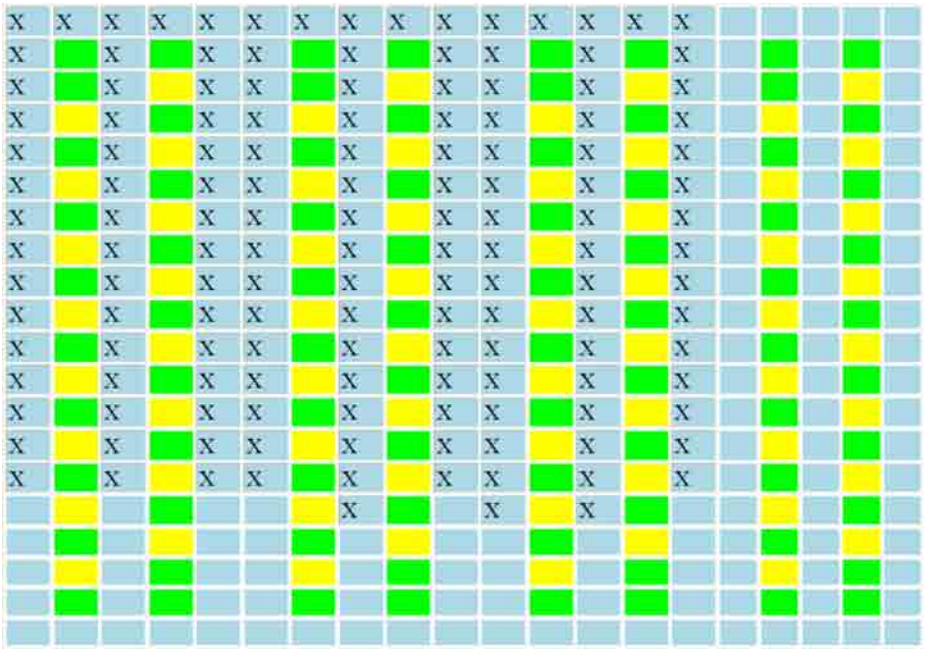


Fig. C.5: Layout of SATD benchmark.

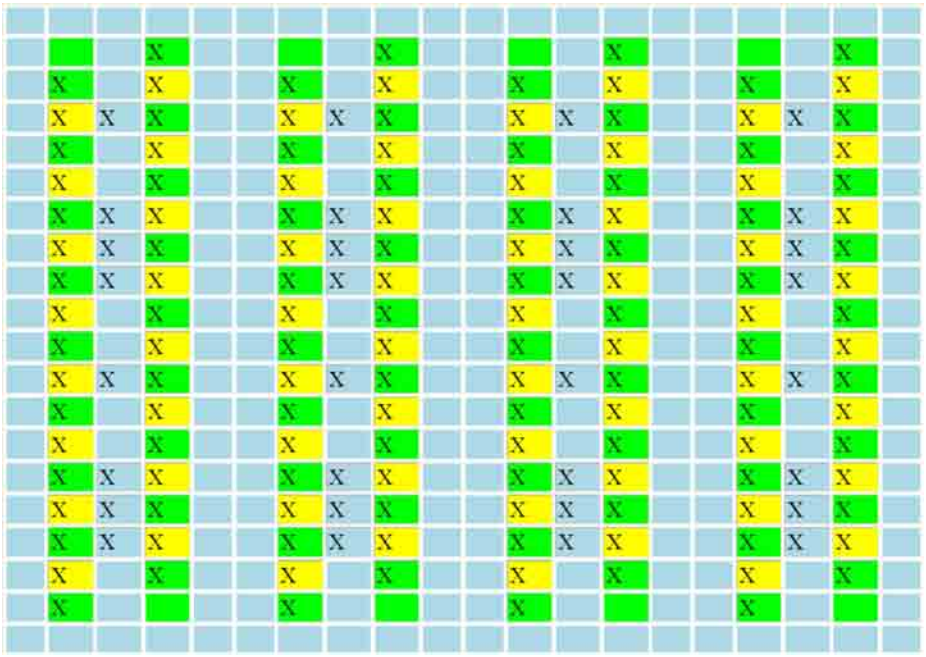


Fig. C.6: Layout of MM benchmark.

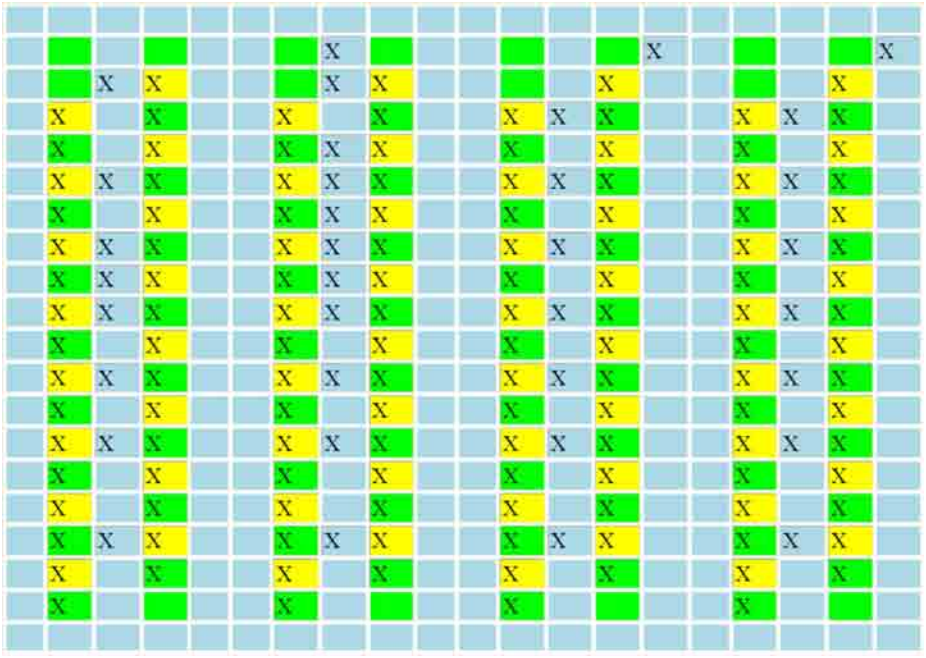


Fig. C.7: Layout of MWS benchmark.

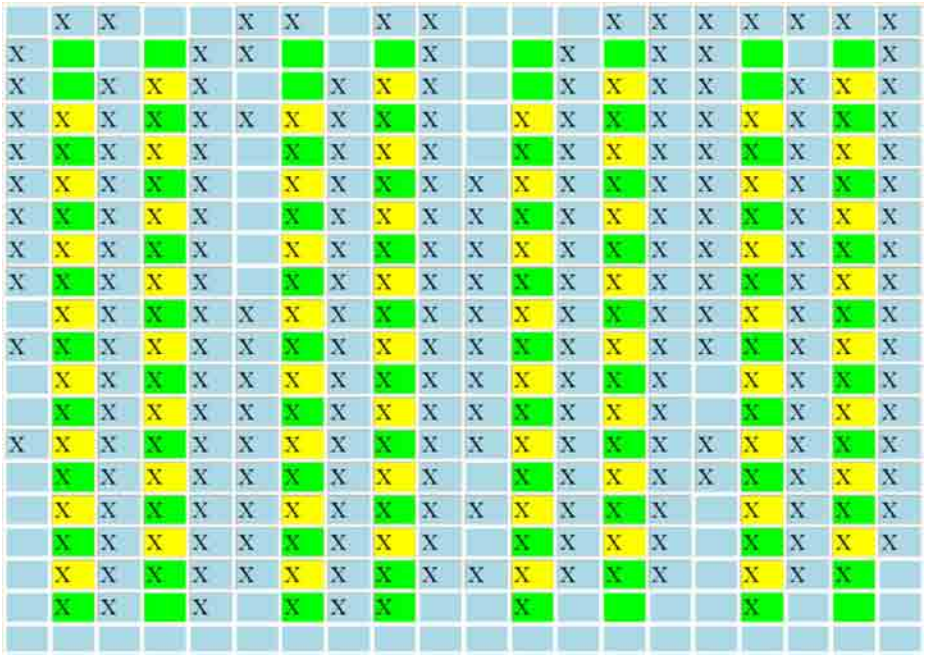


Fig. C.8: Layout of FIR benchmark.

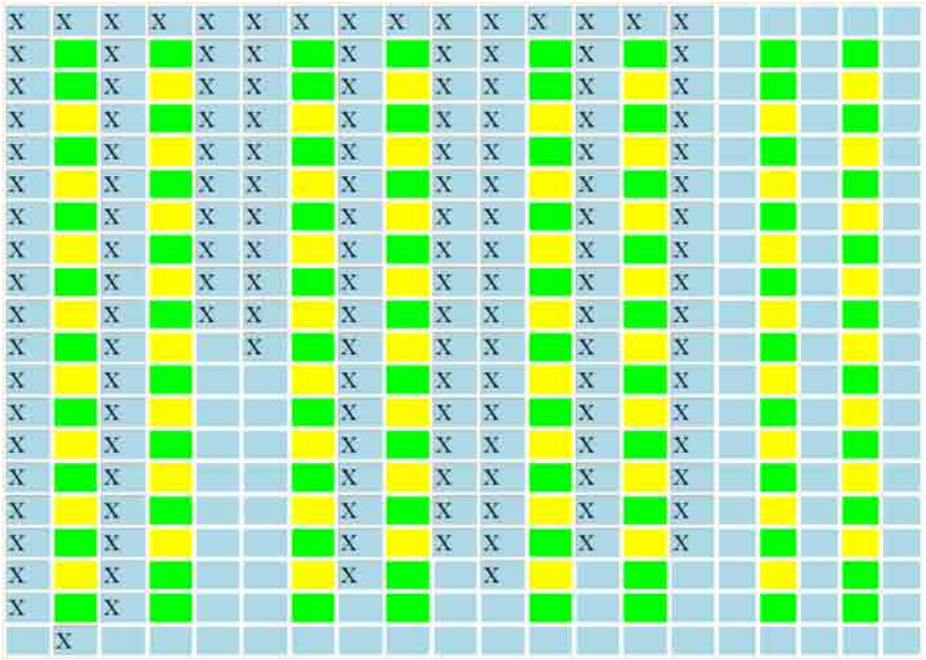


Fig. C.9: Layout of FSS benchmark.

Vita

Rohit Saraswat

Education

- Doctor of Philosophy in Electrical and Computer Engineering, Utah State University, Logan, Utah, 2010.
- Master of Science in Electrical and Computer Engineering, Utah State University, Logan, Utah, 2006.
- Bachelor of Technology in Electronics and Communication Engineering, National Institute of Technology, Calicut, India, 2002.

Published Journal Articles

- B. Eames, S.K. Neema, and R. Saraswat, "DesertFD: A Finite-Domain Constraint Based Tool for Design Space Exploration", Design Automation for Embedded Systems, pp. 43-74, 2009.

Published Conference Papers

- R. Saraswat and B. Eames, "Finite Domain Constraints based Delay Aware Placement Tool for FPOA", in Proceedings of 4th International Conference on ReConFigurable Computing and FPGAs, Cancun, pp. 145-150, Dec. 3-5, 2008.
- R. Saraswat and B. Eames, "On the Use of DesertFD to Generate Custom Architectures for H.264 Motion Estimation", in Proceedings of 15th IEEE International Conference on Engineering of Computer-Based Systems, Belfast, pp. 359-368, Mar 31- Apr. 4, 2008.

Awards and Achievements

- International Student Council service scholarship award, 2008, Utah State University.
- School of graduate studies honor roll, Fall 2005, Utah State University.
- Best poster award, Computer and Information Sciences session, at 85^{th} annual meeting of the AAASPD, 2004.