Nieuwe FPGA-ontwerptools en -architecturen

New FPGA Design Tools and Architectures

Elias Vansteenkiste

Promotor: prof. dr. ir. D. Stroobandt
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: elektrotechniek

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. R. Van de Walle
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2016 - 2017

ISBN 978-90-8578-960-4
NUR 959
Wettelijk depot: D/2016/10.500/92

Examination Commission

prof. dr. ir.	Gert De Cooman Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University
prof. dr. ir.	Joni Dambre, secretary Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University
prof. dr. ir.	Steve Wilton SoC Research Group Department of Electrical and Computer Engineering The University of British Columbia
prof. dr. ir.	Nele Mentens Department of Electrical Engineering - ESAT Faculty of Engineering Technology KU Leuven
prof. dr. ir.	Pieter Simoens Department of Information Technology - INTEC Faculty of Engineering and Architecture Ghent University
prof. dr. ir.	Guy Torfs Department of Information Technology - INTEC Faculty of Engineering and Architecture Ghent University
em. prof. dr. ir.	Erik D'Hollander Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University
prof. dr. ir.	Dirk Stroobandt, advisor Department of Electronics and Information Systems - ELIS Faculty of Engineering and Architecture Ghent University

Dankwoord

Eerst en vooral wil ik Karel Bruneel bedanken voor het overdragen van zijn enthousiasme waarmee hij over zijn - op het eerste gezicht - exotische technieken vertelde. Hij heeft me warm gemaakt voor een thesis rond dynamische herconfiguratie van FPGAs en later ook voor een doctoraat. Tijdens mijn doctoraat spendeerde hij soms meer tijd in mijn kantoor dan in het zijne. Karel Bruneel, samen met Tom en Brahim, leerden Karel Heyse en mij de kneepjes van het vak kennen. Ik apprecieer ook de beschikbaarheid van Karel Heyse als collega-student. Hij was altijd bereid om klankbord te spelen. Ik wil ook Dries bedanken voor de goeie samenwerking en de steun op het einde van mijn doctoraat. Hij was er al bij als masterstudent toen hij een project deed aan onze onderzoeksgroep, daarna als thesisstudent en nu ook als doctoraatsstudent. Karel Heyse en Dries hebben ook geholpen bij het nalezen van deze thesis. Graag wil ook de rest van het Hardware and Embedded System team bedanken voor de discussies en samenwerking. Ik apprecieer ook enorm de vrijheid die mijn promoter Dirk mij gegeven heeft tijdens mijn doctoraat. Ook ben ik erg dankbaar voor de tijd en moeite die de leden van de examencomissie besteed hebben in het nalezen van een eerdere versie van mijn boek. Het heeft de kwaliteit van deze thesis verhoogd.

I am especially grateful towards professor Steve Wilton for making the trip from Vancouver to Ghent to be in my examination board.

De mensen uit het reslab wil ik ook bedanken, waaronder Aaron, Sander, Francis, Pieter, Tim, Jonas, Lio, Jeroen, Ira, Tom, ... Ik keek altijd uit naar de avonden die we gevuld hebben met spelletjes, van Age of Empires II tot Game of Thrones. Daarnaast waren er ook de geanimeerde discussies tijdens de lunch. Het begon altijd met de jacht op goed eten in centrum Gent, waar de nodige onderhandelingen op zijn plaats waren. In het bijzonder bedank ik Aaron. Hij is een goede vriend en was een compagnon de route sinds de beginjaren van onze studies aan de Universiteit. We vertrokken samen op uitwisseling naar Taiwan, begonnen na onze studies een doctoraatsstudie en liepen bijna gelijktijdig een stage in Silicon Valley.

My gratitude also goes to Alireza Kaviani for the chance he gave me to do

an internship at Xilinx' CTO Lab in San José. I learned a lot about FPGAs during my stay. I also want to thank my two other colleagues at Xilinx. Henri Fraissé and Pongstorn Maidee accepted me in their team and helped me guide through Vivado's source code, which is a complex maze.

Zo wil ik ook Karel Bruneel bedanken omdat hij mij onderdak heeft verschaft in San Francisco.

Voor de rest wil ik ook mijn trouwe vrienden bedanken, waaronder Stijn, Elke, Benjamin, Sofie, Dries, ... De vrienden van de zwemclub en de waterpolo, Brecht, Jorley, Willem, Dirk, Jens, Fien, Tim zorgden altijd voor geanimeerde gesprekken na training.

Een speciaal plaatsje wil ik hier reserveren voor mijn vriendin Kaat. Zij is mijn nummer 1. Alhoewel ze de afgelopen maanden af en toe heeft moeten plaats maken voor mijn nummer 2, mijn macbook, hoop ik dat ze me dat niet al te kwalijk zal nemen. Ze was altijd bereid in de bres te springen als ik een deadline had. Ik neem er ook graag de luidruchtige schoonzussen, Sara, Liesbet en Hanne bij en de rest van de gezellige schoonfamilie.

Last but not least, dank aan mijn ouders voor alle kansen die ik gekregen heb, en hun onvoorwaardelijke steun. Ze staan samen met mijn zus en broer altijd voor mij klaar, hoe hectisch het soms ook is.

Elias Vansteenkiste
December 19, 2016

Samenvatting

Field-Programmable Gate Arrays (FPGA's) zijn programmeerbare, digitale chips die ingezet kunnen worden voor verschillende doeleinden. Ze worden gebruikt voor toepassingen waarbij hogere prestaties vereist zijn dan de prestaties die door de goedkopere microprocessors geleverd kunnen worden. Typische vereisten zijn een hoge doorvoer, korte wachttijden en een laag stroomverbruik. Een voorbeeld van een toepassing die vaak wordt uitgevoerd met een FPGA is het routeren en filteren van pakketjes in de internet-infrastructuur. In de internet-infrastructuur moeten de pakketjes verwerkt worden aan hoge doorvoersnelheden en met een minimale vertraging. Om dit te kunnen realiseren bestaan FPGA's uit een groot aantal blokken die georganiseerd zijn in een rooster. Sommige blokken zijn flexibel, anderen zijn gespecialiseerd in het uitvoeren van een specifieke functie. Al deze blokken kunnen verbonden worden om een grotere functionaliteit uit te voeren.

De ontwerper beschrijft de versneller op een hoog niveau. Een FPGA-configuratie wordt dan gecompileerd door gespecialiseerde software. Zodra de configuratie gecompileerd is, kan de ontwerper controleren of de configuratie aan de applicatie-eisen voldoet. Als de configuratie aan de vereisten voldoet, dan is het ontwerp klaar. In het geval dat de vereisten niet voldaan zijn, moet de ontwerper zijn beschrijving veranderen zodat de eigenschappen van het ontwerp verbeteren. Daarna moet de FPGA-configuratie andermaal gecompileerd worden en moet de ontwerper opnieuw controleren als aan de eisen van de applicatie voldaan is. Dit langzame proces heet de FPGA-ontwerpcyclus en wordt typisch vele keren doorlopen. Een belangrijk knelpunt in de ontwerpcyclus is de uitvoeringstijd van de FPGA-compilatie. FPGA-ontwerpen zijn steeds groter (of complexer) geworden volgens de wet van Moore. Grotere ontwerpen hebben meerdere uren nodig om gecompileerd te worden. Een belangrijk doel van het werk in dit proefschrift is **het verkorten van de ontwerpcyclus door de FPGA-compilatie te versnellen**.

FPGA-compilatie is opgedeeld in verschillende deelproblemen: synthese, packing, plaatsing en routering. Elk deelprobleem wordt behandeld door een ander ontwerptool. De ontwerpbeschrijving wordt

eerst gesynthetiseerd en afgebeeld op de primitieve blokken die beschikbaar zijn op de FPGA. Het resultaat is een netwerk van primitieve blokken. Tijdens packing worden de primitieve blokken geclusterd, waardoor we een netwerk van complexe blokken verkrijgen. De complexe blokken in het netwerk worden toegewezen aan een fysieke locatie op de FPGA tijdens plaatsing terwijl de schakeling op de FPGA geoptimaliseerd wordt voor de toepassingsvereisten. Na plaatsing worden de connecties tussen de blokken gerouteerd.

Plaatsing en routering zijn de meest tijdrovende stappen van de FPGA-compilatiecyclus. De uitvoeringstijd van de packing stap is minder kritisch, maar het beïnvloedt de uitvoeringstijd en de kwaliteit van de plaatsing en routering, daarom hebben we ons gefocust op het versnellen en verbeteren van de packing, plaatsing en routering. De traditionele algoritmes ontwikkeld voor deze problemen zijn niet geschikt voor processors met meerdere kernen, die in het afgelopen decennium de norm geworden zijn in computersystemen. We introduceren nieuwe packing- en plaatsingtechnieken die ontwikkeld zijn voor het uitvoeren op processors met meerdere kernen.

De Packing stap is geïntroduceerd voor het compileren van ontwerpen die geïmplementeerd moeten worden op moderne FPGAs met een hiërarchische architectuur. Er zijn twee populaire technieken voor packing: kiem-gebaseerd en partitionering-gebaseerd. Een kiem-gebaseerd algoritme clustert het ontwerp in één keer en kan daardoor gemakkelijker verzeild geraken in een lokaal minimum. Het is ook moeilijk om te implementeren zodat het gebruik kan maken van meerdere processorkernen. Een kiem-gebaseerd algoritme is wel goed in het opleggen van architectuurbeperkingen. Partitioneringgebaseerde algoritmes produceren een hogere kwaliteit omdat ze de natuurlijke hiërarchie van het ontwerp behouden. Het is ook gemakkelijker een meerdradige implementatie te maken van een partitioneringgebaseerde algoritme. In tegenstelling tot de kiem-gebaseerde algoritmes is het echter wel moeilijk om de architectuurbeperkingen op te leggen. We combineerden deze twee benaderingen om het beste van beide werelden te krijgen.

Bij plaatsing van de blokken in het ontwerp wordt voor ieder blok een fysieke bloklocatie op de FPGA toegewezen. Conventionele analytische methodes plaatsen een ontwerp door het minimaliseren van een kostfunctie die een schatting van de post-routeringprestatie voorstelt. Helaas is het niet mogelijk om alle architectuurbeperkingen door een analytisch oplosbare kostfunctie te beschrijven, daarom wordt het probleem opgelost in meerdere iteraties. In elke iteratie wordt een kostfunctie aangepast aan het resultaat van de vorige iteraties en de archi-

tectuurbeperkingen. Daarna wordt de functie opnieuw geminimaliseerd. De minimalisering omvat een tijdrovend proces: het oplossen van een lineair systeem. Experimenten tonen aan dat het niet nodig is om een hoge nauwkeurigheid te hebben voor de tussentijdse resultaten. In onze nieuwe plaatsingstechniek volgen we de snelst afdalende gradient. Dit is sneller dan het oplossen van een lineair systeem. Het maakt het ook mogelijk om blokniveauparallellisme toe te passen.

In de routeringsstap vindt de router een pad voor elk net in het ontwerp. Een net bestaat uit meerdere verbindingen vanuit dezelfde signaalbron, dit is typisch een uitgangspin van een blok. Paden voor verschillende netten kunnen geen draden delen of dit zou leiden tot een kortsluiting. Conventionele routeringsalgoritmen lossen dit probleem op door een mechanisme toe te passen waarbij netten meermaals worden opgebroken en opnieuw gerouteerd, terwijl de kost van draden verhoogd wordt als die door meerdere netten gebruikt worden. Hierdoor lost de congestie geleidelijk op en krijgen we een routering zonder kortsluitingen. In onze aanpak herrouteren we connecties in plaats van netten. Dit stelt ons in staat om enkel connecties die gecongesteerde draden gebruiken opnieuw te routeren (in plaats van volledige netten) wat veel tijd bespaart, zeker voor de netten met veel connecties.

Tijdens het ontwikkelen van de nieuwe ontwerptools ontdekten we een ander belangrijk probleem in de academische FPGA-gemeenschap. Onderzoek naar FPGA-ontwerptools of -architecturen wordt meestal uitgevoerd met behulp van een academisch raamwerk waarvan de broncode vrij te verkrijgen is, omdat academici geen toegang hebben tot de broncode van commerciële FPGA-ontwerptools en -ontwerpen. We hebben een populair academisch raamwerk met een commercieel raamwerk van één van de belangrijke FPGA-fabrikanten vergeleken. We hebben het verschil in resultaten gemeten en we vonden een grote kloof op het vlak van compilatietijd en kwaliteit van het eindresultaat. De snelheidsprestaties van de ontwerpen gecompileerd door het academisch raamwerk waren 2x slechter dan wanneer ze werden gecompileerd door het commercieel raamwerk. Een tweede doel van dit proefschrift is **het bewust maken van de kloof tussen commerciële en academische resultaten en die kloof proberen te verkleinen**.

Om de kloof te verkleinen introduceren we nieuwe technieken om de runtime en de kwaliteit van de FPGA-ontwerptools te verbeteren, in lijn met onze eerste doelstelling. Een groot deel van het verschil is te verklaren door de geavanceerdere commerciële FPGA-architectuur. Daarom onderzochten we nieuwe FPGA-architecturen met kleine logische poorten in het interconnectienetwerk. We dimensioneerden de transistoren in deze architecturen en we hebben nieuwe ontwerptools

ontwikkeld voor deze architecturen om de prestaties te evalueren.

Een derde doelstelling is **het verhogen van de efficiëntie van FPGA-ontwerpen**. De efficiëntie van FPGA-ontwerpen kan verhoogd worden door gebruik te maken van de runtime-herconfigureerbaarheid van FPGA's. Een FPGA-configuratie kan gespecialiseerd worden voor de eisen van de applicatie terwijl de applicatie op de FPGA uitgevoerd wordt. Gespecialiseerde configuraties zijn sneller en kleiner. We hebben bijgedragen aan een automatische toolflow die geparametriseerde configuraties produceert. Tijdens de uitvoering worden deze geparametriseerde configuraties geëvalueerd om gespecialiseerde configuraties te verkrijgen zonder de tijdrovende compilatie van het ontwerp opnieuw uit te voeren. We hebben nieuwe plaatsings- en routeringstechnieken ontworpen die de herconfigureerbaarheid van de interconnectieschakelaars in de FPGA uitbuiten.

In het kort: deze thesis draagt bij tot nieuwe ontwerptools, architecturen en het verkleinen van de kloof tussen commerciële and academische ontwerptools.

Summary

Field-Programmable Gate Arrays (FPGAs) are programmable, multi-purpose digital chips. They are used to accelerate applications in case a higher performance is required than the performance delivered by the cheaper microprocessors. Typical requirements are high throughput, low latency and low power consumption. An example of an application that is often implemented with an FPGA is packet routing and filtering in the internet infrastructure where packets have to be processed at high throughputs and with a low latency. To realize this functionality, the FPGA consists of an array of blocks. Some blocks are flexible, others are specialized in executing a specific function. All these blocks can be connected which each other to form a more complex functionality.

The designer describes the accelerator in a high level description language and is compiled by specialized software to an FPGA configuration. Once compiled the designer checks if the application requirements are satisfied. If the requirements are met, the design is finished. If the requirements are not satisfied, the designer has to change his description and recompile the design to recheck the constraints. This slow process is called the FPGA design cycle. It is typically performed multiple times. An important bottleneck in the design cycle is the FPGA compilation runtime. FPGA design sizes have grown following Moore's law. Large designs take multiple hours to be compiled. An important goal of the work in this thesis is to **shorten the design cycle by speeding up FPGA compilation**.

FPGA compilation software is divided in several subproblems: synthesis, packing, placement and routing. Each subproblem is handled by a different design tool. The design description is first synthesized and mapped to the primitive blocks available on the FPGA. The result is a network of primitive blocks. During packing the primitive blocks are packed into more complex blocks. The complex blocks in the network are assigned to a physical location on the FPGA during placement while optimizing the circuit on the FPGA for the application requirements. After placement the connections between the blocks are routed by setting switches in the interconnection network of the FPGA.

Placement and routing are the most time consuming steps of the FPGA compilation flow. Packing requires less runtime but it influences the runtime and quality of the placement and routing process. So to reduce the compilation runtime we focused on new techniques to improve packing, placement and routing. The traditional algorithms designed for these problems are not suited to exploit processors with multiple cores, which have become a commodity in the last decade. We introduce new packing and placement techniques that have been developed with a multi-core environment in mind.

Our new packing technique observes that modern FPGAs have a hierarchical structure to improve area (cost) and delay. On each hierarchical level there are a number of equivalent blocks which can be connected by a routing network. This hierarchical structure is the main reason why a packing phase has been introduced in the compilation flow. In our approach we want to better take the natural hierarchy of the design into account during packing. There are two common approaches to the packing problem: seed-based packing and partitioning-based packing. Seed-based packing packs the design in a single pass. It is prone to local minima and difficult to adapt to be able to exploit multiple processor cores, but it handles architectural constraints well. Partitioning-based packing produces better quality designs because it preserves the natural hierarchy of the design. It is also easy to execute in multiple threads. However it is difficult to handle architectural constraints. We combined these two packing approaches to get the best of both worlds.

In placement the packed blocks in the design are assigned to a physical block onto the FPGA. Conventional analytical placement places a design by analytically solving the minimization of a cost function, which represents an estimate of the post-route performance. Unfortunately it is not possible to put all the architectural constraints in one analytically solvable cost function. So the problem is divided in multiple iterations. In each iteration the cost function is adapted to the result of the previous iterations and is minimized again. The minimization encompasses the runtime intensive solving of a linear system. Experiments show that it is not necessary to have the high accuracy of the intermediate solutions. In our approach we use a steepest gradient descent based optimization which is faster than solving a linear system and still produces the same quality placements. It also allows to exploit block level parallelism.

In the routing step the router finds a path for each net in the design. A net consists of several connections coming from the same signal source. Paths for different nets cannot share wires or this would lead

to a short circuit. Regions where paths want to share wires but can't are called congested regions. Conventional routing algorithms solve this problem by a negotiated congestion mechanism in which nets are ripped up and rerouted multiple times while increasing the cost of congested wires. In this way the congestion is gradually solved. In our approach we rip up and reroute connections instead of nets. This allows us to only reroute the congested connections which saves a lot of runtime, certainly for the nets with a lot of connections.

In the process of improving the design tools, we discovered another important problem in the academic FPGA community. Research on FPGA design tools or architectures is typically performed with an open source framework, because the commercial FPGA design tools are proprietary and closed source. We compared the popular academic framework with the commercial framework of one of the important FPGA vendors. We measured the gap and found it to be significant in terms of compilation runtime and quality of the end result. The speed-performance of the designs compiled by the academic framework were 2x slower than if they were compiled by the commercial framework. So another goal of this thesis is **to raise awareness and reduce the gap between commercial and academic results**.

To reduce the gap we introduced new techniques to improve the runtime and quality of the FPGA design tools, which aligns with our first objective (to shorten the design cycle by speeding up FPGA compilation). A large part of the gap is also because of a more advanced commercial FPGA architecture. We investigated new FPGA architectures that have small logic gates in the routing network. We sized the transistors in these architectures, developed new compilation tools for these architectures and evaluated their performance.

A third objective of this thesis was **improving the efficiency of FPGA designs**. The efficiency of FPGA designs can be improved by exploiting the runtime reconfigurability of FPGAs. An FPGA configuration can be specialized for the runtime needs of the application while the FPGA is executing. Specialized configurations are faster and smaller. We contributed to an automatic flow that produces parameterized configurations. These parameterized configurations are evaluated at runtime to get a specialized configuration without the runtime intensive recompilation of the design. We developed placement and routing tools that exploit the reconfigurability of the routing switches in the FPGA.

We conclude with emphasizing that this thesis contributes to new design tools, new architectures, and the reduction of the gap between commercial and academic tools.

Contents

List of Acronyms

ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
AIG	And-Inverter Graph
AP	Analytical Placement
ARM	Advanced RISC Machines
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
BDD	Binary Decision Diagram
BLE	Basic Logic Element
BLIF	Berkeley Logic Interchange Format
BRAM	Block RAM
CAD	Computer-Aided Design
CAM	Content-addressable Memory
CGRA	Coarse-Grained Reconfigurable Array
CLB	Configurable Logic Block
CM	Configuration Manager
CMOS	Complementary Metal–Oxide–Semiconductor
CPD	Critical Path Delay
CPU	Central Processing Unit
CTO	Chief Technology Officer
CV	Computer Vision
CW	Channel Width

DAO	Depth-optimal Area Optimization
DCS	Dynamic Circuit Specialisation
DDR SDRAM	Double data rate synchronous dynamic random-access memory
DPR	Dynamic Partial Reconfiguration
DRAM	Distributed RAM
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FB	Functional Block
FET	Field-Effect Transistor
FF	Flip-Flop
FIFO	First In, First Out
FinFET	Fin Field Effect Transistor
FIR filter	Finite Impulse Response filter
FM	Frequency Modulation
FPGA	Field-Programmable Gate Array
GB	GigaByte
GP	General Purpose
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HP	High-Performance
HPC	High-Performance Computing
HPWL	Half-Perimeter Wire Length
HWICAP	Hardware ICAP
HLS	High-Level Synthesis
IBM	International Business Machines Corporation
ICAP	Internal Configuration Access Port
IO	Input/Output

IOB	Input/Output Block
IP	Intellectual Property
ISE	Xilinx Integrated Synthesis Environment
JIT	Just-In-Time
K-LUT	K-input LUT
KU	Kintex UltraScale
LAB	Logic Array Block
LC	Logic Cluster
LD	Logic Depth
LI	Local Interconnect
LIFO	Last In, First Out
LVDS	Low-Voltage Differential Signaling
LUT	LookUp Table
MAC	Multiply-Accumulate Unit
MB	MegaByte
MM	Multi-mode
MCW	Minimum Channel Width
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MUX	Multiplexer
NFA	Non deterministic Finite Automaton
NIDS	Network Intrusion Detection System
NMOS	N-channel MOSFET
NRE cost	Non-Recurring Engineering cost
OS	Operating System
PAL	Programmable Array Logic
PaR	Place and Route
PCIe	Peripheral Component Interconnect Express
PE	Processing Element

PI	Primary Input
PLL	Phase Lock Loop
PMOS	P-channel MOSFET
PO	Primary Output
PPC	Partial Parameterized Configuration
PR	Partial Reconfiguration
RAM	Random-Access Memory
ROM	Read-Only Memory
RCP	Representative Critical Path
RRG	Routing Resource Graph
RTL	Register-Transfer Level
RTR	Run-Time Reconfiguration
RTL	Register-Transfer Level
SA	Simulated Annealing
SB	Switch Block
SDC	Synopsys Design Constraints format
SRAM	Static Random Access Memory
SRL	Shift Register LUT
STA	Static Timing Analysis
TCAM	Ternary Content-Addressable Memory
TCON	Tuneable Connection
TH	Treshold
TLUT	Tuneable LUT
TPaR	Tuneable Place and Route
TWL	Total Wire Lenght
TRCE	The Timing Reporter And Circuit Evaluator tool from Xilinx
VCGRA	Virtual CGRA
VHDL	Hardware Description Language

VPR	Versatile Place and Route
VTB	Verilog-To-Bitstream
VTR	Verilog-To-Routing
WL	Wire Length
WNS	Worst Negative Slack
XDL	Xilinx Design Language

1 Introduction

This thesis starts with an introduction to the FPGA by answering some frequently asked questions. Next the fundamental problems related to FPGA compilation and architectures that are addressed in this thesis are described. This is followed by our contributions that help towards solving the problems. In the last sections we describe the structure of this thesis and list the publications about the work in this dissertation.

1.1 Introduction to FPGAs

What is an FPGA? A Field Programmable Gate Array is a type of programmable, multi-purpose digital chip. They are programmable in the 'field' after they are manufactured. An FPGA essentially consists of a huge array of gates which can be programmed and reconfigured any time, anywhere. However, "A huge array of gates" is an oversimplified description of an FPGA. A modern FPGA consists of an array of programmable blocks. Some of those blocks are very flexible. They contain look-up tables which can perform simple Boolean logic operations, registers to temporarily store results and resources that connect the lookup tables and registers. Other blocks are specialized in a specific task such as Digital Signal Processing (DSP) blocks, memory blocks, high speed communication resources, ... The blocks are embedded in an interconnection network, which can be programmed to connect the blocks together to make a circuit of your choice.

Why would we use an FPGA? An FPGA is used to accelerate an application that requires intensive computations, high throughput, low latency calculations or has a stringent power budget. An FPGA is flexible and is built to exploit the parallel nature of the problem. How much parallelism is used to implement the application is completely up to the designer. The application design architect can tailor a custom processor to meet the individual needs of the application.

How does it work? How is an FPGA used? The typical work environment for an FPGA is depicted in Figure 1.1. The hardware/application designer writes his application accelerator or process kernel in a high level description language such as VHDL, Verilog, OpenCL, ... Subsequently the design description is compiled by specialized software to an FPGA configuration bitstream on the workstation of the application designer. The compilation software is typically divided in different steps which are being handled by different tools. The compilation tool flow is also called Computer Aided Design (CAD) or Electronic Design Automation (EDA) tool flow. We use the former term in what follows. The software tools in the CAD tool flow are an important subject in this thesis. The majority of the chapters describe improvements and speed-up techniques made to the most time consuming steps of the compilation tool flow. After the compilation, the design is then typically tested on a printed circuit board which contains the target FPGA. The FPGA configuration bitstream is sent to the program interface of the test board and the FPGA is programmed with the configuration bitstream.

How does it compare to other popular digital chips? Other important popular digital chips are microprocessors, Graphical Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs):

- **Microprocessors** perform tasks by splitting them up in small and simple operations and perform these operations sequentially in one or more threads, depending on the number of cores in the microprocessor. Each of these simple operations (instructions) is executed by specific hardware on the microprocessor chip.

- **GPUs** contain a large array of processors specialized for multiply and accumulate (MAC) operations and distributed memory to support this. They are specially designed to support video processing and graphics rendering, but they are also used for other applications that need a lot of MACs, such as training convolu-

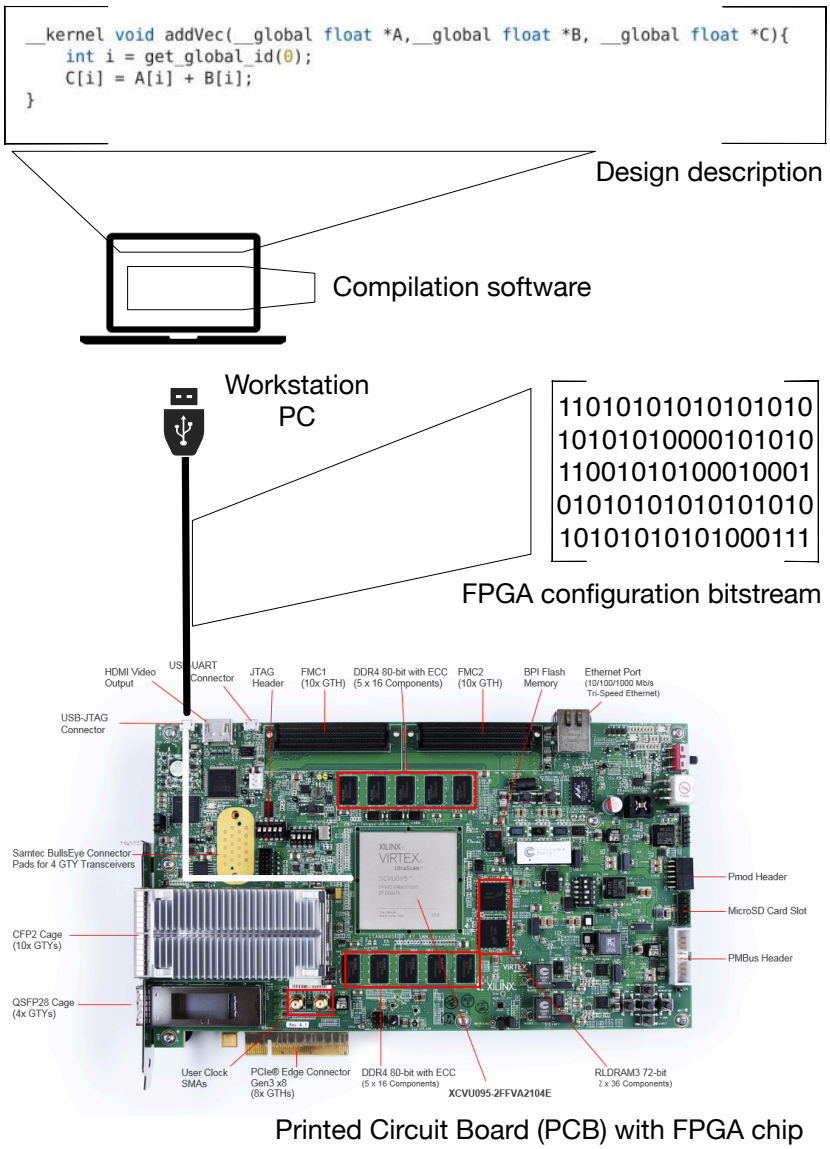


Figure 1.1: An overview of a work environment for an FPGA

Table 1.1: Comparison of typical microprocessor, FPGA, ASIC and GPU designs. Partly reproduced from [56].

	Microprocessor	FPGA	ASIC	GPU
Example	ARM Cortex-A9	Virtex Ultrascale 440	Bitfury 16nm	Nvidia Titan X
Flexibility during development	Medium	High	Very high	Low
Flexibility after development[1]	High	High	Low	High
Parallelism	Low	High	High	Medium
Performance[2]	Low	Medium	High	Medium
Power consumption	High	Medium	Low	High
Development cost	Low	Medium	High	Low
Production setup cost[3]	None	None	High	None
Unit cost[4]	Medium	High	Low	High
Time-to-market	Low	Medium	High	Medium

[1]E.g. to fix bugs, add new functionality when already in production

[2]For a sufficiently parallel application

[3]Cost of producing the first chip

[4]Cost of producing each chip after the first

tional networks, digital signal processing, ...

- **ASICs** are single purpose chips. They are manufactured to only perform one big function. Any digital circuit can be baked into the silicon during production, but they cannot be reprogrammed. They are typically fast and low power, like FPGAs, but each chip can only perform the one function that is baked into the silicon during production and cannot be reprogrammed.

An overview and comparison of the properties of these chips with the properties of the FPGA is summarized in Table 1.1

FPGAs are typically used when the application requirements are not met by the cheaper microprocessors. They have a vastly wider potential to accelerate applications than the microprocessor and they excel in power consumption. Only ASICs can achieve higher speeds, lower power consumption and lower unit costs, because they are specially made for the application and don't have the overhead sustained by the programmability of the FPGA. However, for low and medium volumes ASICs are too expensive, because producing a custom silicon chip has a large upfront cost due to the high cost of design and production setup (e.g. photomasks). FPGAs are mainly used for small to medium volume products and ASICs for very high volume products. ASICs also lack flexibility, once they are produced they can't change their functionality

During development an ASIC is specialized for the application. Each wire and transistor is placed specially for the application and it is

therefor the most flexible of accelerators. The FPGA is the second most flexible during development. The application has to be implemented by connecting low level generic programmable blocks and specialized blocks on the FPGA. They can be interconnected in various patterns. Microprocesors are less flexible during development, because the application has to be executed by splitting the task up in simple operations that are executed sequentially in one or a few threads. The development of the accelerator is restricted mostly when developing for GPUs. GPUs typically contain from several hundred up until several thousand cores. However, each processing core is only capable of executing a small subset of basic operations, most notably the multiply and accumulate operations. GPUs are very specialized accelerators, they are focused on MAC-heavy applications that require a high throughput and low latency. In an ASIC and an FPGA the dataflow can be specialized for the application, in a GPU you are restricted by different aspects, such as the available memory caches and the 16/32/64 floating point operations. Additionally FPGAs serve a broader spectrum of applications than the GPU.

FPGAs, GPUs and microprocessors have a lower commercial risk and a faster time-to-market than ASICs [142]. Mistakes made during development can easily be fixed after development by reprogramming the device. There is also the possibility to add new functionality in future upgrades. The reprogrammability extends the time a product stays relevant, because features can be changed according to the changing demand. For this reason, some products ship with an FPGA/GPU/microprocessor that is over-dimensioned, to allow for future upgrades. For ASICs there is almost no flexibility after development. Development mistakes that make it into an ASIC require a very expensive silicon respin or even a product recall. Microprocessor solutions have the lowest time-to-market, because of its ubiquity and the well developed compilation and debug tools. Developing for FPGAs and GPUs is more complex, which results in a higher time-to-market The ASIC takes the cake for time-to-market with its complex and slow development process.

Microprocessors have only limited capabilities to exploit the parallellism of an application. They typically only have a 1 to 6 cores. GPUs have typically much more with up to 3840 cores for the recent Nvidia Quadro P6000 GPU. For FPGAs and ASICs, the number of processing units is completely up to the designer. It can be adapted to the needs of the application.

For a same technology node an ASIC will have a higher performance compared to the FPGA in terms of area, speed and power con-

sumption, because the functionality is hard-wired and there is no programmability overhead. However, many new ASIC designs do not use the latest process technology, because they are way more expensive than older ones, whereas FPGA vendors do [1, 4]. Because of this, the speed, area and power gap is smaller between FPGAs compared to a functionally equivalent ASIC in an older process technology. For a MAC heavy application that requires high throughput, the GPU probably will have the upper hand in comparison to the FPGA, but where the FPGA excels is the power performance. FPGAs even outperform the GPU in terms of energy efficiency for the MAC intensive evaluation of convolutional networks [81]. The microprocessor typically has the lowest performance in terms of area, speed and power consumption.

An ASIC is has the highest development cost. A lot of man hours and high license costs for the EDA tools. It also has a large production setup costs, which is the cost to produce the first chip. The FPGA does not have production setup costs and it has a lower development cost than an ASIC, but it has a higher development cost than developing an accelerator for GPU or CPU, because of the license costs and the more time consuming design cycle. A downside of both the FPGA and the GPU is the relatively high unit cost, which is typically higer than the omnipresent microprocessor. An ASIC has relatively the lowest unit cost.

Who uses FPGAs? What are the important applications of the FPGA? The most important applications implemented on the FPGA are packet routing, switching and filtering in the internet infrastructure. Wired and wireless communication has grown to over half of the FPGA business with important customers as Cisco, TE connectivity, Juniper networks and many more. With such an important share of the revenue, it has driven innovation in FPGAs to support this application domain.

Another important application domain is high performance computing, with important customers as IBM for example. Datacenters prefer FPGAs to perform some tasks over other processing units, because of their superior performance per power unit and the flexibility to reprogram the FPGA at any time. Microsoft is a pionier in this aspect. Other important application domains are video processing and sensor processing, for example low latency virtual reality rendering and seismic imaging software. FPGAs are also favoured in embedded systems, because of their low power signature.

FPGAs are also used by engineers in the design of ASICs. The application is first prototyped, debugged and tested with the help of an

FPGA. Radiation upsets are emulated with an FPGA. Test vectors are calculated by injecting faults in the design. Possibly the first generation of a product is sold with an embedded FPGA. Once the major problems have been ironed out, the hard-wired version of the design is produced and embedded in the second generation of products. An example is the Lattice Semiconductor LFXP2-5E low-cost non-volatile FPGA that was embedded in the motherboard of Apple's 2011 Macbook pro to switch the LVDS display signal between the two GPUs. In the 2012 versions it was replaced by the Texas Instrument's dedicated ASIC, HD3SS212. Another company using the same strategy is Nokia.

New application fields are being unlocked as we write. One example is accelerating inference by evaluating convolutional nets on the FPGA.

Who produces FPGAs? The main FPGA vendors are Xilinx and Altera, now part of Intel. They are both based in Silicon Valley. They design and sell FPGAs, but they outsource the manufacturing to specialized silicon foundries, such as Taiwan Semiconductor Manufacturing Company, Limited (TSMC) or Intel. Other smaller FPGA companies focus on niche markets, such as Lattice with its low power and low cost FPGAs and Microsemi with its non-volatile low power FPGAs.

How much does an FPGA cost? The cost of an FPGA is largely dependent on the size of the chip, i.e. how many programmable blocks and input/output interfaces are available on it. The price range of one FPGA unit varies a lot between 1 EUR for low end, smaller and older FPGAs to a few 10,000 EUR for the high end, large flagship devices of the newest technology node.

The second aspect of an implementation that affects its cost is the clock frequency. The clock frequency is the drum beat that defines the rate at which computations are performed. It is determined by the electric delay of the hardware and depends on the configuration of the FPGA. If a design does not meet minimal performance requirements, this can be solved by redesigning it using more resources (e.g. more parallelism or pipelining) or choosing a more expensive FPGA with lower electric delay (higher speedgrade or newer technology node).

1.2 Introduction to the Research

There are a few fundamental problems and opportunities we try to address in this thesis. We will mainly discuss the problems in this section,

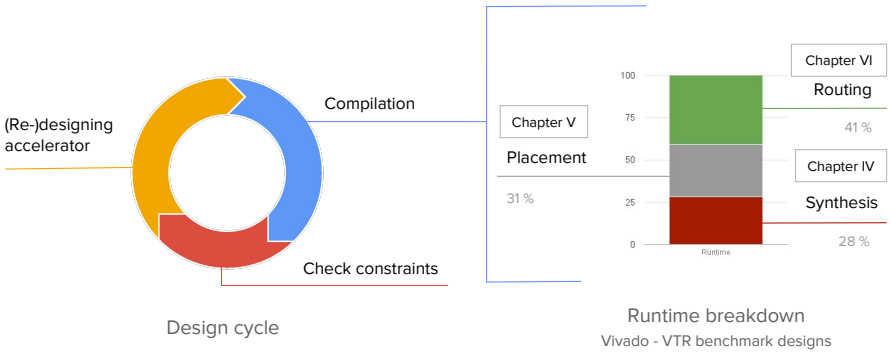


Figure 1.2: The FPGA design cycle and the runtime breakdown of the FPGA compilation.

the solutions we investigated are described in the next section.

1.2.1 The Slow FPGA Design Cycle

The design cycle for FPGA design is illustrated in Figure 1.2. The hardware engineer designs the accelerator by describing the different modules of the accelerator. Once the design has been described, the design is compiled to an FPGA configuration. The compiler tries to meet the design constraints. However, it is possible that the design constraints are too stringent for the given design description and FPGA architecture. So after the compilation is finished, the designer has to check if the FPGA configuration meets all the application constraints. Typical constraints are maintaining a certain throughput, an upper limit for latency, lowest cost (area) and a small power budget. In case the compiled design meets the constraints, the cycle is finished. In case it doesn't meet the constraints, the engineer has to change his design to obtain different characteristics after compilation. Depending on the gap between the obtained performance and the goal, there a number of options: make fundamental changes to the algorithm, target a different FPGA or make smaller changes, such as properly pipelining for example. The design has to be recompiled and the constraints have to be rechecked. This cycle typically has to be performed numerous times. We want to shorten the design cycle as much as possible, because a slow design cycle means high engineering costs and a slow time-to-market.

There are two important approaches for shortening the design cy-

cle. On the one hand we can try to increase the productivity of the engineer by increasing the ease of use. For example, an active research field with this aim is the high-level synthesis efforts. They make it easier for the designer to describe the application by using a high level language: C, SystemC or OpenCL. On the other hand the compilation runtime should be as short as possible. For large designs the compilation is the bottleneck of the design cycle. It can easily take a few hours to compile a design. For example the *mes_noc* design from the Titan benchmark suite with 549K blocks requires 4h to be compiled by the Altera's Quartus compilation tool flow and 10h with an academic compilation tool flow for a single threaded execution. The size of commercial designs easily surpasses 500K blocks.

Reducing the compilation time also improves the ease of use, because the engineer can use the compilation flow for trial and error approaches as is common in the software programming domain. In this thesis we focus on reducing the compilation time by improving the compilation steps.

FPGA Configuration Compilation

Generating the optimal FPGA configuration is nearly impossible, because the solution space is very large. To simplify the problem, the compilation is split up in several steps. In the first step the design description is synthesized and mapped to the available functional block types on an FPGA, resulting in a network with functional block instances. We call this step synthesis. Next, each block in the network is assigned to a physical block location on the FPGA, which is called the placement step. Finally, the connections between the blocks are routed by deciding which switches need to be set in the interconnection network. Even the subproblems in each of the compilation steps are hard to solve. Generating the optimal solution for one step would take a very long time. This only worsens for larger designs, so the FPGA vendors and academic community try to find heuristics that generate a near optimal solution in a reasonable timeframe. Each compilation step has its influence on the end result and influences the runtime of the other steps downstream. The bar chart in Figure 1.2 shows the runtime breakdown for the different compilation steps. The current synthesis, placement and routing approaches account for an equal part of the total compilation runtime. In this thesis we look at each of these steps in a dedicated chapter and propose new techniques to reduce the runtime and improve the quality of the result.

1.2.2 The Gap between Academic and Commercial Results

It is hard to make conclusions about research work around new FPGA compilation techniques or new FPGA architectures, because everyone is working with a different framework. A framework includes the target FPGA, the compilation tools and the benchmark designs. The FPGA vendors keep the details of their architecture secret. They specialize their FPGA compilation tools to their architectures and the compilation tools are closed source. This makes it hard for academic researchers to benchmark their new approaches in a commercial framework. There are academic frameworks available but they lag behind the commercial frameworks in almost every aspect, which makes it hard to estimate the value of new techniques.

1.2.3 Improving the Efficiency of FPGAs

As Moore's law is ending and technology process scaling is slowing down [4, 119, 139], it is imperative to find new ways to improve the FPGAs performance. This includes investigating new architectures and different techniques to use the FPGA more efficiently. The main objective is to reduce the cost of FPGA design, while improving the performance.

Architecture

In the past years the continuous race towards the next smaller technology node pushed FPGA architects towards designing architectures that are performing well when scaling down and are easy to adapt to the new process technology node. Incremental changes to the architecture and tools were preferred above drastic changes. One example is changing the ratio between specialized blocks and generic blocks in the FPGA. Another example is changing the size of specialized memory blocks.

As the advantage of newer process technology diminishes, newer more exotic architectures can become more interesting to further push the performance forward.

Partial Reconfiguration

New techniques are emerging that try to exploit the "hidden" features of the current FPGAs to improve the efficiency. One example is partial and runtime reconfiguration. The configuration memory of an FPGA has to be loaded with a configuration bitstream at start-up before the

FPGA can start to execute. Modern FPGAs allow parts of the configuration memory to be rewritten at runtime, thus changing the function of these resources. This can be done without affecting the operation of other parts of the FPGA. Two types of reconfiguration can be distinguished. They are illustrated in Figure 1.3.

Modular Reconfiguration is a technique in which a region of the FPGA is reserved to implement several predefined circuits, one at a time, and it is possible to switch between the predefined circuits on the fly using partial reconfiguration (Figure 1.3a) [12, 141, 146]. Without partial reconfiguration, all of the circuits would have to be implemented in separate regions of the FPGA, each using its own set of resources. This would result in a resource cost that is many times larger. This type of partial runtime reconfiguration is becoming popular in datacenters. It reduces the cost when larger FPGAs in the cloud can run two or more independent accelerators concurrently.

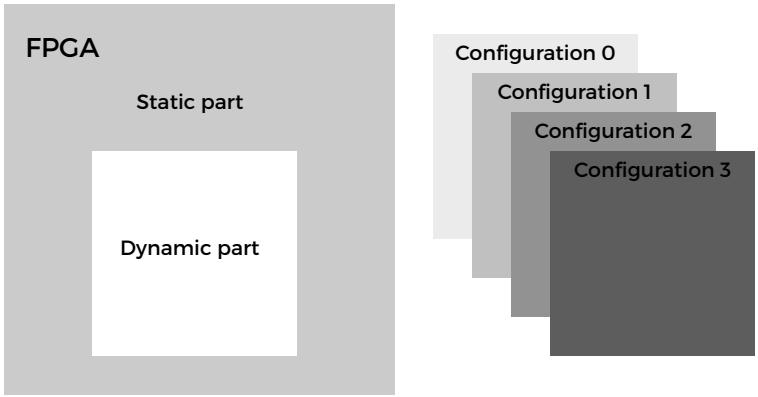
Micro Reconfiguration Partial reconfiguration can also be used to reconfigure very small parts of the FPGA, such as a single logic or routing resource (Figure 1.3b). This is called micro reconfiguration. Micro reconfiguration is used to slightly tweak a circuit, for example the coefficients of a digital filter or ease the transition between different modes.

Problems The partial and runtime reconfiguration techniques have not found their way in a lot of commercial applications because the techniques perform badly in terms of ease of use. There is a lack of good compilation tools and the process currently requires a lot of manual work. In this work we mainly focused on micro reconfiguration.

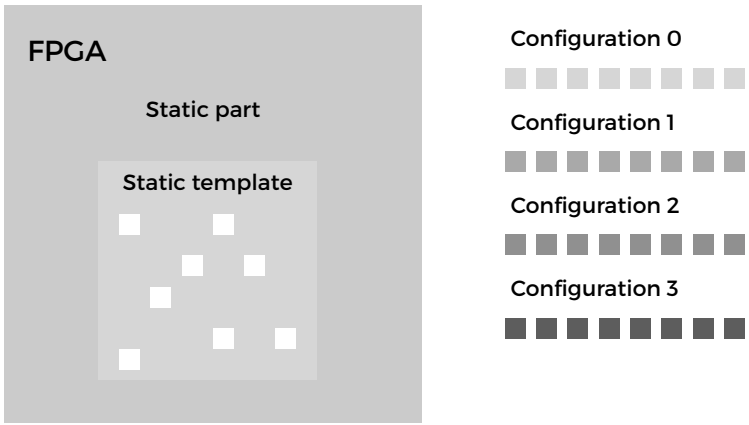
1.3 Contributions

In this dissertation we contributed towards solving the problems mentioned in the previous section. These are our main contributions:

Quantifying the Gap between Academic and Commercial Results It is difficult to assess how the new ideas on compilation tools and architectures presented in academic conferences and publications would perform in a commercial framework. Academic work is benchmarked against the well known compilation tools, architectures and benchmark designs in open source frameworks. There is a danger in only using open source frameworks to evaluate new tool and architectural ideas,



(a) Modular reconfiguration.



(b) Micro reconfiguration

Figure 1.3: The two partial and runtime reconfiguration techniques

it creates an academic bias. We measured the gap between academic and commercial results and found it to be substantial. To reduce the gap we focused mainly on new FPGA compilation techniques and architectures. There are already other academic researcher working on trying to reduce the gap between commercial and academic benchmark designs [106].

New FPGA Compilation Techniques Many of the old compilation techniques are designed for single core processors. It is hard to adapt these old techniques in order to exploit the acceleration potential of the multiple cores in the modern workstations. We investigated the main runtime consuming parts of the old techniques and propose new compilation techniques that can accelerate these parts by exploiting the multi-core processor environment. We propose new pack, placement and routing techniques that improve the runtime and quality. These efforts contribute to a shorter FPGA design cycle and a smaller divide between the academic and commercial results We also made our new compilation tools available to the academic community in an open source project [136]. In contrast to the other open source projects in the academic community, which are mainly implemented in C [1, 88], the new compilation tools are implemented in Java. Java is a platform independent high level programming language. This makes it easier for other researchers to adapt the compilation tools to suit their research objective.

Developing New Compilation Techniques for Micro Reconfiguration To exploit micro reconfiguration, we propose to dynamically specialize the accelerator for the needs of the application. To make it easier for the designer we automatically produce a static template and parameterized configurations for the micro parts in the configuration. This is only possible because of the newly proposed dynamic circuit specialization compilation flow. We contributed to this compilation flow with new place and route techniques.

Investigating new FPGA Architectures To improve the efficiency of FPGAs we investigated a range of new architectures. We focused on FPGA architectures with small logic gates introduced in their interconnection network. To test these architectures we developed new compilation techniques and sized the architecture with the help of an electronic circuit simulator.

1.4 Structure of the Thesis

This thesis is organized as follows: in the background section (Chapter 2) we give an overview of the current state of the FPGA architecture and describe the different steps in the CAD tool flow that generate FPGA configurations, synthesis, technology mapping and packing, placement and routing. A historic context of the FPGA is also described in this chapter to put the work in this thesis in perspective. In Chapter 3 we investigate the performance gap between the results obtained by academic and commercial research frameworks. A research framework includes the FPGA CAD tool flow, a target FPGA architecture and benchmark designs. A research framework allows researchers to make conclusions about new techniques, algorithms or architectures. The gap indicates that research conclusions in the academic and commercial world can differ, which negatively impacts the whole FPGA ecosystem.

In Chapters 4, 5 and 6 we give more detailed background on the packing, placement and routing problem respectively and describe new algorithms applied to these problems. A hierarchical multi-threaded partitioning algorithm for packing is described in Chapter 4. A steepest gradient descent based algorithm for placement is explained in Chapter 5 and in Chapter 6 we introduce a connection-based routing algorithm with a more fine grained negotiated congestion mechanism which allows to save routing runtime.

In Chapter 7 we describe the placement and routing algorithms we developed for compiling parameterized FPGA configurations for the dynamic reconfiguration of the FPGA's routing network. We also investigated a new FPGA architecture with logic gates in the routing nodes and in Chapter 8 we describe the sizing results, the technology mapping and packing algorithms we developed to test the architecture.

1.5 Publications

Journal Papers

- Dries Vercruyce, **Elias Vansteenkiste** and Dirk Stroobandt. "How preserving Design Hierarchy during Multi-threaded packing can improve Post Route Performance". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, In review.

- Tom Davidson, **Elias Vansteenkiste**, Karel Heyse, Karel Bruneel and Dirk Stroobandt. "Identification of Dynamic Circuit Specialization Opportunities in RTL Code". *ACM Transactions on Recon-*

figurable Technology and Systems, Vol. 8, Issue 1, No. 4, 2015, 24 pages

- D. Pnevmatikatos, K. Papadimitriou, T. Becker, P. Böhm, A. Brokalakis, Karel Bruneel, C. Ciobanu, Tom Davidson, G. Gaydadjiev, Karel Heyse, W. Luk, X. Niu, I. Papaefstathiou, D. Pau, O. Pell, C. Pilato, M.D. Santambrogio, D. Sciuto, Dirk Stroobandt, T. Todman and **Elias Vansteenkiste**. "FASTER: Facilitating Analysis and Synthesis Technologies for Effective Reconfiguration". *Microprocessors and Microsystems*, Volume 39, Issues 4–5, June–July 2015, Pages 321–338.

- **Elias Vansteenkiste**, Brahim Al Farisi, Karel Bruneel and Dirk Stroobandt. "TPaR : Place and Route Tools for the Dynamic Reconfiguration of the FPGA's Interconnect Network". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 33, Issue 3, 2014, Pages 370–383.

Conference Papers with International Peer Review

- **Elias Vansteenkiste**, Seppe Lenders and Dirk Stroobandt. "Liquid: Fast Placement Prototyping Through Steepest Gradient Descent Movement". In *26th International Conference on Field Programmable Logic and Applications, Proceedings (FPL2016)*, 2016. Pages 49-52

- Dries Vercruyce, **Elias Vansteenkiste** and Dirk Stroobandt. "Runtime-Quality Tradeoff in Partitioning Based Multithreaded Packing". In *26th International Conference on Field Programmable Logic and Applications, Proceedings (FPL2016)*, 2016. Pages 23-31

- **Elias Vansteenkiste**, Alireza Kaviani and Henri Fraisse. "Analyzing the divide between FPGA academic and commercial results". In *International Conference on Field-Programmable Technology, Proceedings (ICFPT2015)*. Pages 96-103 (nominated for Best Paper Award)

- Berg Severens, **Elias Vansteenkiste** and Dirk Stroobandt. "Estimating Circuit Delays in FPGAs after Technology Mapping". In *25th International Conference on Field Programmable Logic and Applications, Proceedings (FPL2015)*, 2015, Pages 380 - 383

- Alexia Kourfali, **Elias Vansteenkiste** and Dirk Stroobandt. "Parameterised FPGA Reconfigurations for Efficient Test Set Genera-

tion". In *International Conference on ReConFigurable Computing and FPGAs, Proceedings (ReConFig2014)*, 2014, 6 Pages

- Brahim Al Farisi, **Elias Vansteenkiste**, Karel Bruneel and Dirk Stroobandt. "A Novel Tool Flow for Increased Routing Ronfiguration Similarity in Multi-mode Circuits. In *IEEE Computer Society Annual Symposium on Very-Large-Scale Integration, Proceedings (VLSI2013)*, 2013. Pages 96-101

- **Elias Vansteenkiste**, Karel Bruneel and Dirk Stroobandt. "A Connection-based Router for FPGAs". In *International Conference on Field-Programmable Technology, Proceedings (ICFPT2013)*, Pages 326 - 329.

- Karel Heyse, Tom Davidson, **Elias Vansteenkiste**, Karel Bruneel and Dirk Stroobandt. "Efficient Implementation of Virtual Coarse Grained Reconfigurable Arrays on FPGAs". In *23rd International Conference on Field Programmable Logic and Applications, Proceedings (FPL2013)*, 2013, 8 pages

- Karel Heyse, Tom Davidson, **Elias Vansteenkiste**, Karel Bruneel and Dirk Stroobandt. "Efficient Implementation of Virtual Coarse Grained Reconfigurable Arrays on FPGAs". In *50th Design Automation Conference (DAC2013)*, 2013, Pages 1-8

- **Elias Vansteenkiste**, Karel Bruneel and Dirk Stroobandt. "Maximizing the Reuse of Routing Resources in a Reconfiguration-aware Connection Router". In *22nd International Conference on Field Programmable Logic and Applications, Proceedings (FPL2012)*, 2012, Pages 322 - 329

- **Elias Vansteenkiste**, Karel Bruneel and Dirk Stroobandt. "A Connection Router for the Dynamic Reconfiguration of FPGAs". In *Lecture Notes in Computer Science: 8th International Symposium on Applied Reconfigurable Computing (ARC2012)*, 2012, Pages 357 - 364

2 Background

This chapter contains background information about FPGAs. The architecture of an FPGA and the CAD tools that are used to compile a design to an FPGA configuration are described first. Subsequently the history of the FPGA is described to put the work in this thesis in perspective and explain the current state of the FPGA architecture and tools. The goal of this chapter is to provide a solid foundation for the following chapters.Throughout the background you will find references to all the other chapters in this thesis.

2.1 FPGA Architecture

In this section, a basic overview of a typical FPGA architecture and the state of commercial FPGA architectures is described. An FPGA consists of a large number of functional blocks embedded in a programmable interconnection network. The functional blocks can roughly be divided into a number of categories: Input/Output blocks, hard blocks and soft blocks. We start with the low level building blocks of the FPGA and follow a bottom-up approach to explain the architecture. The description starts with the basic building blocks and ends with the high-level overview of the FPGA architecture.

2.1.1 Low Level Building Blocks

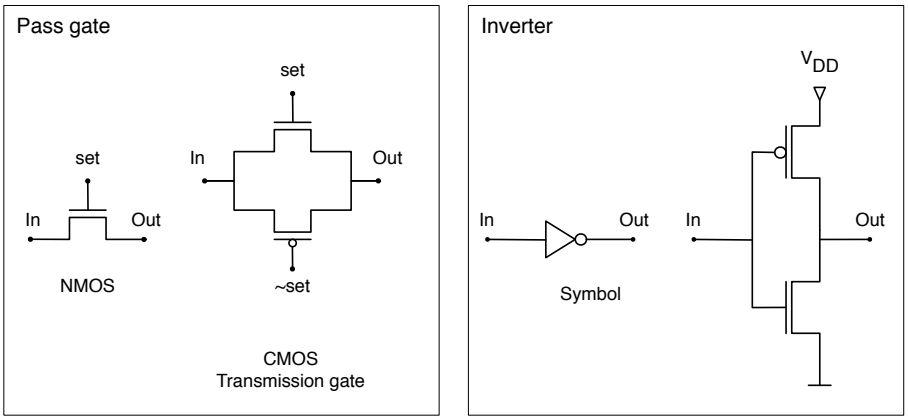


Figure 2.1: The schematics for the pass gate and inverter. Both are basic building blocks of an FPGA.

Pass Gates are basic switches, with three pins, the input, output and on/off pin. If it is turned on, it passes the logic level from the input pin to the output pin. If the switch is turned off, then the input and output pin are disconnected by a high impedance. It is a basic element used to build multiplexers and SRAM cells. Pass gates can be implemented using an n-type MOSFET (NMOS) or p-type MOSFET (PMOS). NMOS pass gates are good at passing a logic low signal but they are bad at passing a logic high. Similarly PMOS pass gates are good at passing a logic high but bad at passing a logic low. This causes problems for chained pass gates. The logic signal degrades at each stage and needs to be restored. Instead of restoring the signal, another common solution is to combine the advantages of the NMOS and PMOS pass gate by connecting them in parallel. This is called a transmission gate. The downside here is that two transistors are used to produce a tranmission gate. Figure 2.1 shows the symbol used for a NMOS pass gate and the schematic for the transmission gate.

Inverters have only two pins. The inverter outputs the complement of the input signal. The schematic of an inverter is depicted in Figure 2.1. The input signal drives the gates of an NMOS and a PMOS. The NMOS is switched to pass on the logic low level if the input signal is high and the PMOS passes the logic high level if the input signal is low. The inverter is the preferred circuit to strengthen signals because the NMOS and PMOS are switched in the way their advantages are ex-

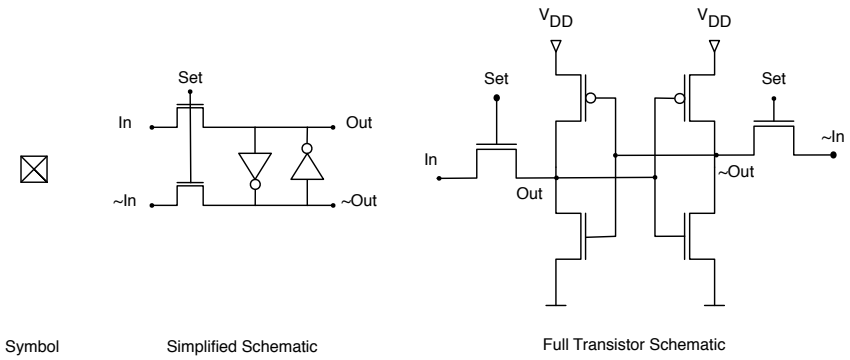


Figure 2.2: Schematics for the SRAM cell.

ploited. The PMOS is good at passing a logic high and the NMOS is good at passing a logic low. Inverters are used to strengthen a weakened signal and reduce the rise/fall times for signals that have to drive a large downstream capacitance. Typically two or more inverters are cascaded to form a chain buffer.

Static Random-Access Memory Cells (SRAM cells) are able to store a single bit. Figure 2.2 shows schematics for an SRAM cell. They consist of two inverters connected in a ring. Feedback ensures that the value is stored as long as V_{DD} is high. Two pass gates are used to enforce the input signal and its inverse to the nodes in-between the inverters. The drivers of the input signal have to be stronger than the inverters used in the cell. The nodes in-between the inverters are also used as the output pins of the SRAM cell. SRAM cells are the basic element of the configuration memory. The layout of the SRAM cell is highly optimized because FPGAs contain so many, e.g. Xilinx' flagship FPGA, the Virtex UltraScale 440 contains close to one billion SRAM cells. In the following sections is described how SRAM cells are used to configure the interconnection network and the soft blocks.

Multiplexers (MUX) select one of their input pins depending on the selection signal and pass the logic value of the selected input pin to the output pin. N:1 multiplexers have N input signals, one output signal and at least $\lceil \log_2 N \rceil$ selection signals. Multiplexers are completely built up by pass gates. In Figure 2.3 the typical symbol of a multiplexer and two common transistor-level implementations are shown. For the

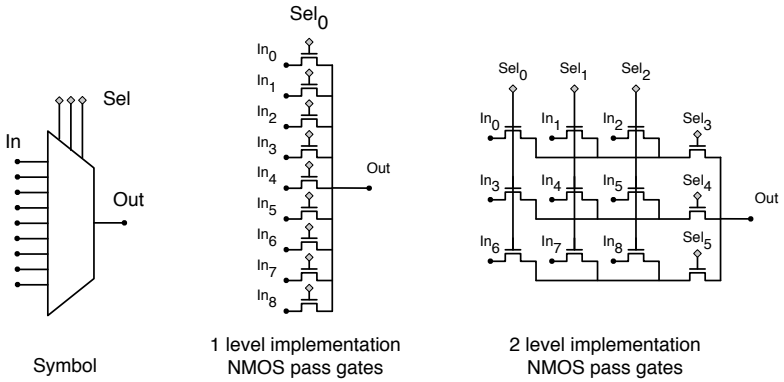


Figure 2.3: Different transistor schematics for multiplexers.

sake of simplicity only NMOS pass gates are used in the schematics, because of the more compact symbol. For smaller multiplexers the 1-level implementation is the fastest implementation because the input signals only have to pass a single NMOS gate. For larger multiplexers, the capacity and the resistance of the summing node becomes too high, which makes the 1-level implementation slower. Another downside of the 1-level implementation is that the number of selection signals is equal to the number of inputs, which will be an important disadvantage for routing multiplexers, because the selection signals in routing multiplexers are provided by SRAM cells and consequently more silicon area will be required. The 2-level implementation is a good trade-off. The signal has to pass two NMOS gates and encounters a larger propagation delay, but the number of selection signals is now between $2 \cdot \lfloor \sqrt{N} \rfloor$ and $2 \cdot \lceil \sqrt{N} \rceil$, which is much more scalable for larger multiplexers. Multiplexers are extensively used in the routing network and are an important part of the LookUp Table implementation.

LookUp Tables - LUTs implement arbitrary Boolean functions of their inputs using a truth table stored in configuration memory. The truth table contains the desired output value for each combination of the input values. In Figure 2.4 a schematic of a 4-input LUT is depicted. A LUT is typically built up by a 2:1 multiplexer tree and it acts as a large $2^k : 1$ multiplexer with k the number of LUT inputs. The SRAM cells provide the multiplexer inputs and the LUT input pins provide the selection signals. Each selection signal is shared between all the 2:1

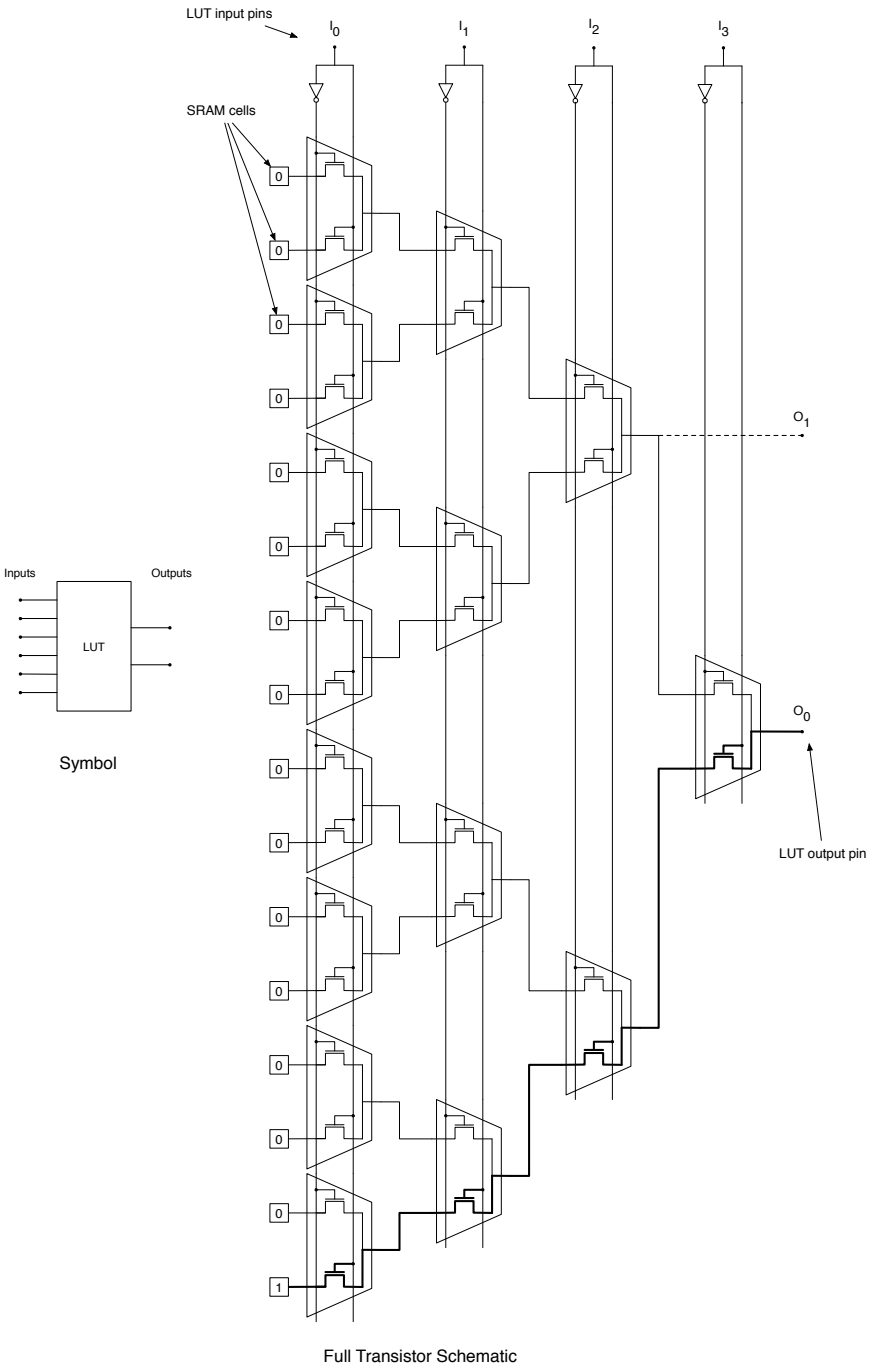


Figure 2.4: Schematic of a 4-input LUT. The basic building block is a 2:1 multiplexer. The SRAM cells in this example are configured so that the LUT implements a 4-input AND gate.

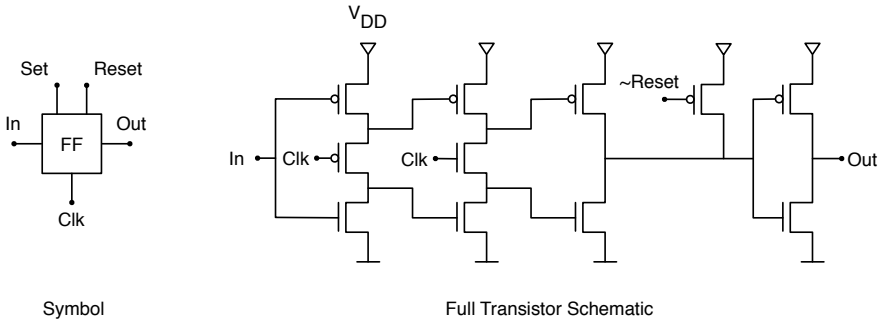


Figure 2.5: Symbol and schematic for a positive edge triggered D-type flip-flop.

MUXes of each stage. Normally the signal is buffered after passing 2 or more pass gates, but the buffers are omitted for clarity. The optimal position of the buffers and the number of buffer stages depend on the process technology anyway. The LUT input pins are logically equivalent, but there exist typically faster and slower pins. In our 4-LUT example, the I_0 pin is the slowest input pin and I_3 pin is the fastest.

In the example in Fig. 2.4 two 3-inputs LUTs can be combined in one 4-input LUT if they share two input signals. LUT implementations that are capable of combining LUTs with shared inputs are called fracturable LUTs. The main FPGA vendors introduced LUTs with more inputs and more sharing options. Altera uses 8-input fracturable LUTs since the introduction of Stratix II family in 2004. The 8-input fracturable LUTs can implement a selection of 7 input functions and two 6 input LUTs with 4 shared signals. Xilinx moved from 4-input LUTs to 6-input LUTs with the introduction of their Virtex 5 architecture in 2006. The sharing options are more limited since two 5-input LUTs have to share all inputs to be implemented in the same LUT.

Flip-flops - FFs are memory elements that are designed to transfer the logic value at the input pin to the output pin after one clock cycle. They can be used to implement sequential logic. In Figure 2.5 a positive edge triggered D-type flip-flop is depicted. This is the most common type of flip-flop used in digital design. The flip-flops in commercial FPGA architectures are more versatile. The flip-flop takes the logic value at the input pin on the positive edge of the clock signal and makes it available

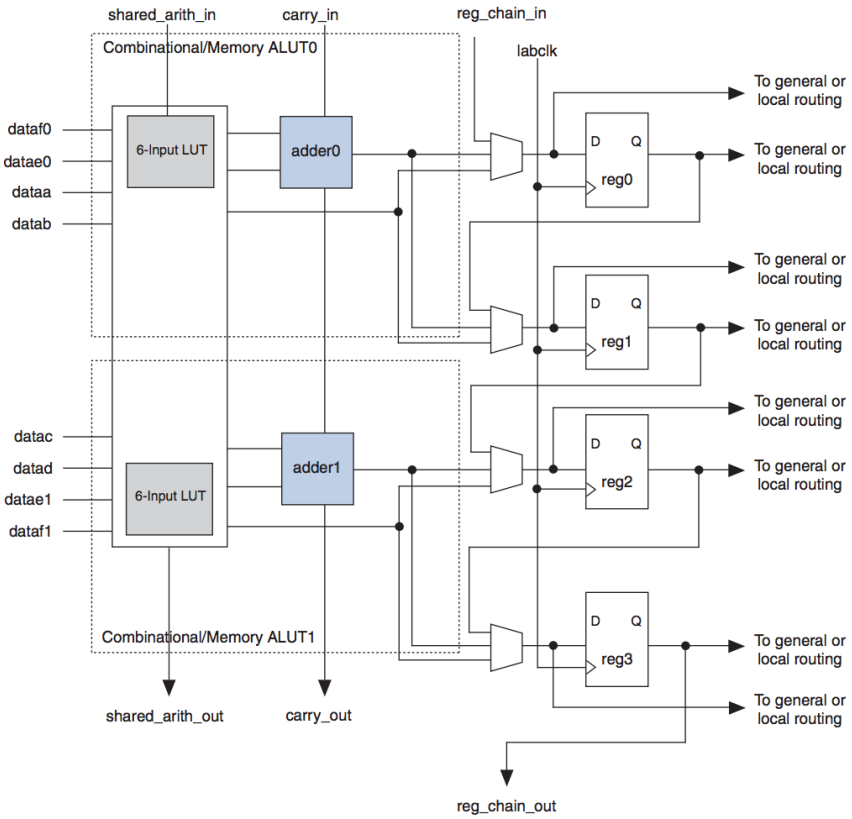
at the output pin on the next rising edge of the clock. Flip-flops in commercial devices are more complex. They can be programmed to trigger on the negative edge by inverting the clock signal, but they can also be programmed to act as simple set-reset latches. Additionally setting and resetting can be done synchronously or asynchronously. We refer the reader to the manuals of the FPGA vendors for more details [147, 13].

2.1.2 Basic Logic Element (BLE)

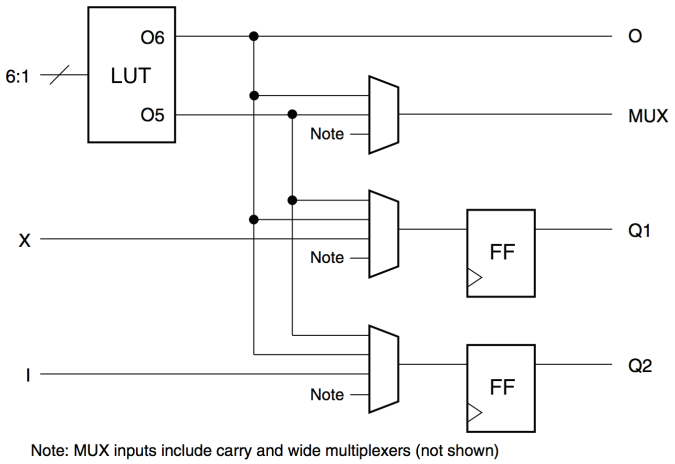
A basic logic element contains typically one or two LUTs, one or two Flip-Flops, adder or carry chain logic gates and multiplexers to connect the blocks. The majority of the BLE inputs are connected directly to the inputs of the LUT. There is also typically an input that bypasses the LUT and is directly fed to the FF input MUX. The output of the LUT is connected to the FF input MUX, in that way the output of the LUT can be stored. The FF input MUX selects which signal is fed to the FF. The output of the FF and the LUT are outputs of the BLE. Figure 2.6 shows schematics for the basic logic element architecture used by the main FPGA vendors.

2.1.3 Soft Blocks

A typical soft block is depicted in Figure 2.7. A soft block is very flexible and is able to implement almost any function of its inputs. They typically contain a number of BLEs and a number of MUXes. The MUXes are switched to form the local interconnect crossbar which can be configured to connect the soft block inputs and BLE outputs to the BLE inputs. There is a lot of symmetry in this architecture. The position of BLEs and the LUT input pins can be swapped without changing the functionality of the soft block. This can be exploited to reduce the connectivity of the local interconnect crossbar, so typically the local interconnect crossbar is not a fully connected one. At the moment of writing FPGAs generally contain 10,000s-100,000s of soft blocks. Soft blocks are also called Configurable Logic Blocks (CLBs) in Xilinx' architectures and Logic Array Blocks (LABs) in Altera's architectures. In this thesis we mainly use the CLB abbreviation. Soft blocks commonly contain extra resources to implement wide multiplexers and carry logic or adders. Another feature is that the SRAM cells in the LUTs of the soft block can be configured to implement memories, such as synchronized RAM and shift registers.



(a) Schematic of the Adaptive Logic Module in Altera's Stratix V architectures.



(b) Schematic of the basic logic element in Xilinx' UltraScale architectures.

Figure 2.6: Schematic of the Basic Logic Elements in the FPGAs from the main FPGA vendors. Schematics are taken from the respective manuals [13, 147].

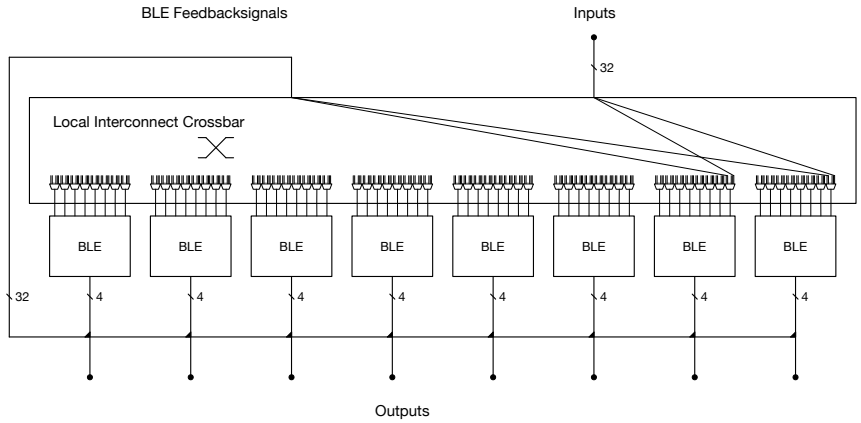


Figure 2.7: A simplified schematic of a soft block. Soft blocks are also called Configurable Logic Blocks (CLBs) in Xilinx' architectures and Logic Array Blocks (LABs) in Altera's architectures.

2.1.4 Hard Blocks

Hard blocks are optimized for a specific function or task that is common in FPGA designs. They are smaller, faster and consume less energy compared to their implementation in soft blocks. We describe the important ones:

DSP Blocks contain one or more adders and multipliers to accelerate the common multiply-accumulate operations heavily used in digital signal processing applications such as video and audio processing. The DSP blocks also contain registers for pipelining the input and output. DSP blocks are capable to combine different smaller width multiplications/additions. Commonly they can also be used to implement bitwise logic functions, pattern detection, barrel shifters, wide counters, wide XOR and wide-bus multiplexing. We refer the reader to the manuals and white papers of the FPGA vendors for more detailed information about the architecture and features of the DSP blocks in commercial devices [9, 108, 148].

RAM Blocks are specialized large memories. They have one or more data ports, address ports and clock inputs. RAM blocks support memory with different aspect ratios, different data and address widths. They can be cascaded to form larger memories. Dual port RAM blocks have two independent data and address ports and two clock inputs. Both data ports access the same memory. RAM blocks are frequently

used to implement FIFO and LIFO buffers and have some extra hardware features to support these use cases. The size of RAM blocks varies amongst vendors and device families, Xilinx' UltraScale architecture has 36 Kb and 288 Kb blocks. Altera has 9 Kb blocks and 144 Kb blocks in their Stratix IV devices but only 20 Kb blocks in their Stratix V devices. We refer the reader to the manuals of the FPGA vendors for more detailed information about the features of the RAM blocks in commercial devices [149, 13].

Other Blocks typically found in FPGA devices are clock management resources and processor blocks. Clock management resources are an essential part of the FPGA. They generate the clock signal with a PLL, distribute the clock signal to all registers and take care of the clock skew effect. It additionally allows to use different clock domains. Some FPGAs contain hard microprocessor blocks. They are more powerful and efficient than soft-core microprocessors implemented on the fabric of the FPGA. PowerPC processors have been embedded in Xilinx' Virtex families since the introduction of the Virtex-II Pro family. The fully fledged ARM processors are embedded in Altera's System-on-a-Chip (SoC) devices and Xilinx' Zynq devices.

2.1.5 Input/Output Blocks

The Input/Output Blocks (IOBs) are connected to the external pins of the chip and thus allow communication with the outside world. It is difficult to scale the I/O pads and pins down at the same rate as has been done for transistors. This problem leads to the number of I/O pins being an important part that determines the area of the die and consequently influences the price setting of FPGAs. The solutions for this I/O bandwidth constriction is the introduction of higher speed I/O transceivers with serializer/deserializer infrastructure. There are also specialized blocks to implement different IO protocols, such as PCIe, Ethernet, ...Some FPGAs have embedded ADC's to capture analog input signals.

2.1.6 High-level Overview

In the beginning FPGA devices only contained simple soft blocks and IOBs. FPGA architects designed one tile for the soft block and replicated the tile to make the masks for the whole die. The soft blocks in these older devices are organized in a large array, hence the "Array" in the acronym FPGA. Nowadays an FPGA is a heterogeneous device

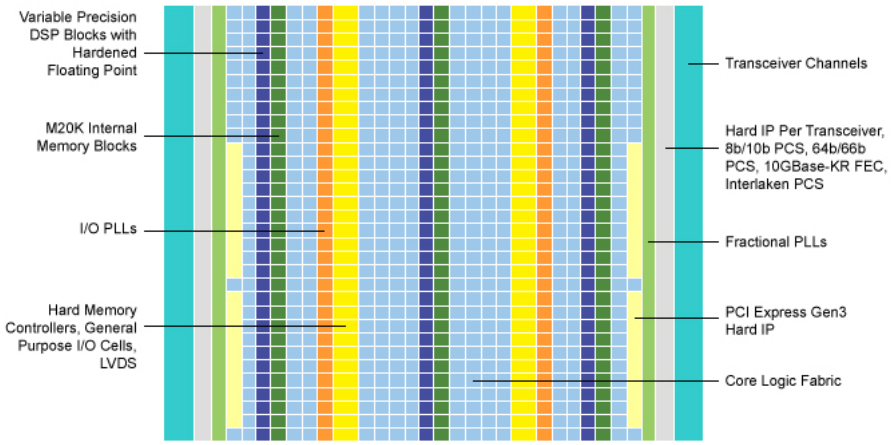


Figure 2.8: A high level overview of the block organisation of Altera's Arria 10 architecture.

Table 2.1: Flagship devices of the latest device families for the main FPGA vendors.

Property/Device	GX 5500	GX 2800	XCVU13P	ZU19
FPGA vendor	Altera	Altera	Xilinx	Xilinx
Device Family	Stratix 10	Stratix 10	Virtex UltraScale+	Zynq UltraScale+
Technology Proc.	14nm	14nm	16nm	16nm
IO Blocks GP/HiS	1,640/72	1,160/144	832/128	310/572
Total Mem. Size*	166 Mb	244 Mb	503 Mb	80 Mb
Soft Blocks**	187K LABs	93K LABs	216K CLBs	65K CLBs
DSP blocks**	3,960	11,520	12,288	1,968
Embedded μ-proc.	yes	yes	no	yes

GP = general purpose
HiS = High Speed
* Total memory size includes distributed memory and RAM blocks
** DSP blocks and Soft blocks are different for each vendor and family. Here follow the main differences, keep in mind that there are other important differences. DSP blocks in Altera's Stratix 10 devices contain 18x19 multipliers. DSP blocks in UltraScale devices contain 18x27 multipliers. CLBs contain 8 6-LUTs and LABs contain 20 6-LUTs.

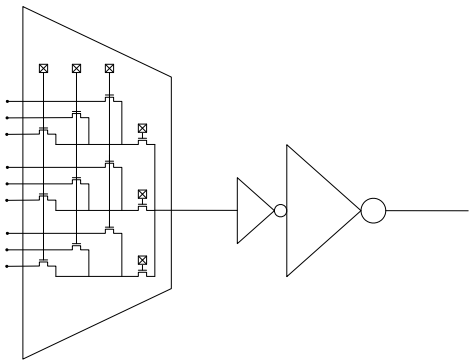


Figure 2.9: A 9:1 routing multiplexer consisting of 6 SRAM cells, a 2-level mux and a 2-level buffer.

and contains several types of blocks. Typically flagship FPGA devices contain an embedded micro processor, several hundreds of IOBs, 100K or more soft blocks, several thousand DSP blocks, several hundred Mb of memory spread out in RAM blocks and distributed memory. In Table 2.1 the properties are listed for the devices from the main FPGA vendors that we consider flagship devices. The blocks are still organized in arrays but some columns contain different types of blocks. Figure 2.8 shows an overview of a fabric fragment of Altera's Arria 10 device family, which clearly demonstrates the column based layout.

2.1.7 Programmable Interconnection Network

To execute a larger functionality the functional blocks in an FPGA have to be able to connect. The inputs and outputs of the functional blocks can be connected to each other using the FPGA's routing network. To provide interconnectivity the routing network is built up of a lot of routing multiplexers and wiring in-between the routing multiplexers. The wiring is partly realised in the metal layers that run on top of the functional block tiles. A typical routing multiplexer is shown in Figure 2.9. A routing multiplexer is a multiplexer with SRAM cells connected to the selection inputs. The selected signal is then typically buffered and distributed to other routing multiplexers and input pins of the functional blocks. The routing multiplexers are the subject of investigation in Chapter 8. We investigated the performance of the architecture if we insert a NAND gate in the buffer after the multiplexer.

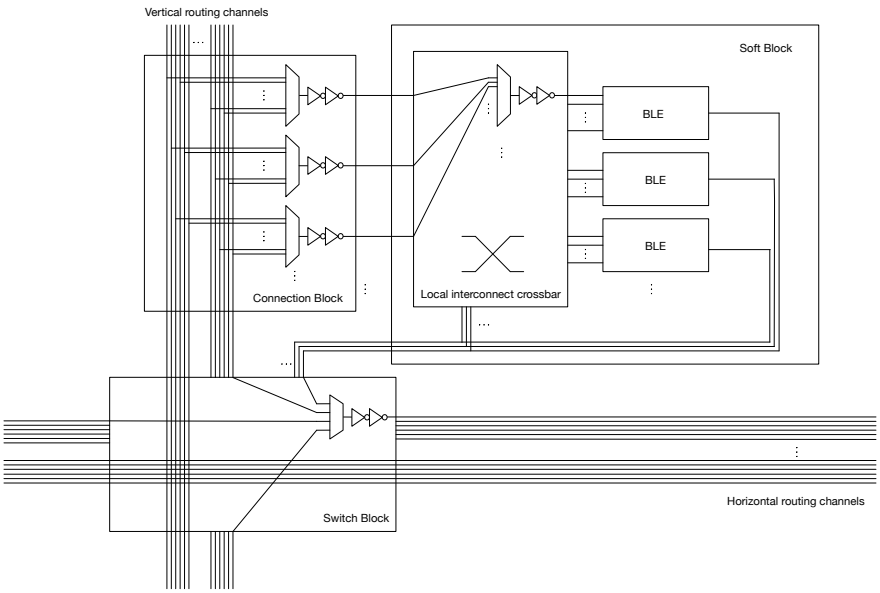


Figure 2.10: The routing network fragment of one tile.

At a higher level, the routing network is typically organized in connection blocks, switch blocks and routing channels, as depicted in Figure 2.10. The wiring is bundled in vertical and horizontal routing channels. The width of the routing channel indicates the number of wires in the channel, which can easily surpass 200 in commercial devices. A majority of the wires span multiple blocks. The length of wires typically varies and spans between 1 and 24 functional block tiles. In commercial devices the routing network also has diagonal wires. Switch blocks are located at the intersection of vertical and horizontal routing channels and provide a limited crossbar functionality. Switch block routing multiplexers are configured to select a signal from one of the wires in the other channels or from the outputs of the functional block. The selected signals are strengthened and driven onto the wires. Connection blocks hook the functional block up to the routing network by routing multiplexers that connect the wires inside the routing channels to the inputs of the functional block. A functional block tile typically contains a part of the routing network, such as its connection block, a switch block and wires going to other tiles. In that way the routing network can be built up by stitching tiles together. The local interconnect crossbar of the soft logic block is typically also considered to be routing infrastructure. The clock signal however is distrubuted to the flip-flops in the CLBs by a dedicated routing tree network. They are different from regular

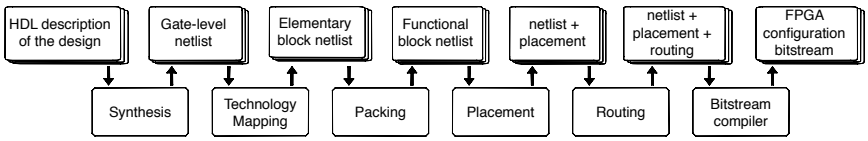


Figure 2.11: Tool flow for FPGA configuration compilation.

routing resources because they are optimised for clock signals, e.g. to minimize clock skew.

The routing network constitutes a very significant part of the silicon area of an FPGA. We sized a soft logic block tile of the basic architecture used in [30] and the switch block contributes 34% of the area, the connection block 20 % and the local interconnect crossbar 15%. Together these account for 69% of the soft block tile area.

The collection of SRAM cells from the routing network and the functional blocks form the configuration memory. In our explanation we ommitted the special infrastructure to read and write the configuration memory. SRAM cells are volatile and, they have to be configured again every time at boot-up.

2.2 FPGA CAD Tool Flow

The functionality of an FPGA is not fixed during the production process and can be changed by writing a different configuration to the configuration memory. This flexibility leads to a substantial reduction of the economic risk of developing hardware accelerators. Unfortunately the flexibility of the FPGA comes at a price. Each time the application developer wants to test his design, a new FPGA configuration has to be compiled. Finding the configuration bits that define the FPGA's functionality is done in an automated way by the FPGA CAD tool flow. The compilation/translation of a high-level description of the application to an FPGA configuration is typically divided in several steps: synthesis, technology mapping, packing, placement and routing (Figure 2.11). A short description of each step is given in the following sections. In each step NP-hard problems have to be solved. The main FPGA device manufacturers and the academic community have developed heuristics to approximate an optimal solution, unfortunately it remains a computationally intensive process. The most time-consuming steps are the placement and routing steps. Murray et al. report a placement runtime of 7 hours for the one million block sized *bitcoin_miner* design, which is part of the Titan23 benchmark suite [106].

Table 2.2: Density and cost scaling of low cost Kintex UltraScale FPGAs. Prices in EUR were obtained from Octopart for one item of the cheapest edition of each FPGA (August 2016).

Device	Price	CLBs	IOBs	DSP Slices	Total Mem.(Mb)
KU025	850	1818	456	1152	17
KU035	977	25391	760	1700	25
KU040	1270	30300	760	1920	28
KU060	2290	41460	912	2760	47
KU085	3480	62190	996	4100	71
KU095	4243	67200	1014	768	64
KU0115	5175	82920	1216	5520	94

2.2.1 Optimization Goals

In each compilation step the tools can pursue different optimization goals. Depending on the constraints given by the designer the tools optimize the FPGA configuration for the application. The most important optimization goals are listed here.

Power Consumption Low power applications such as wearable devices, space satellites, ... have stringent power consumption requirements. The CAD tools can optimize the FPGA configuration for lower power consumption by reducing the switching activity in the design based on typical design usage. Making frequently used connections shorter helps reducing the power consumption of the design. Reducing the size of the device also helps because smaller FPGAs typically consume less energy. Lowering the clock frequency or partially switch off unused parts of the design can also help but that is considered to be the area of the designer. The designer can adapt the design or constraint file respectively if the applications allow it.

Speed High speed applications in the area of wired/wireless communication, high performance computing and real time applications benefit from achieving high clock frequencies and high throughput. The CAD tools optimize the FPGA configuration to be run at higher clock frequencies. To increase the maximal clock frequency the CAD tools try to shorten the critical path, which is the slowest combinational path in the design from one sequential block to another. A sequential block is a Flip-flop or register in the DSP/RAM blocks.

Cost (Area) Cost is a factor that impacts every application implemented on FPGAs. A smaller FPGA is always cheaper, this applies to each FPGA family. An FPGA family is a series of FPGA devices from the same company, made in the same technology node and typically optimized for the purpose (High performance/Low Cost/Low power consumption). As a demonstration the prices for Xilinx' Kintex UltraScale devices are listed in Table 2.2. Depending on the application the critical resource usage may be the soft, IO, RAM or DSP block usage. Tools are typically trying to reduce soft and hard block usage as much as possible.

Since the routing infrastructure is a major part of the chip, minimizing the routing resource usage of a design can greatly reduce the FPGA production cost. Efficient compilation tools can help to reduce the silicon area and the number of metal layers needed to produce the FPGA. The routing resource usage metrics used by the tools are the total wirelength and routing channel width.

2.2.2 Overview of the Tools

In the following paragraphs we introduce each of the steps in a typical FPGA tool flow.

Synthesis

In the synthesis step the design, described in an HDL (Hardware Description Language), is translated to a gate-level logic circuit (Figure 2.12a), which means a Boolean network of simple logic gates (AND, NOT, flip-flop, ...) (Figure 2.12b). Some optimisation steps can be applied to the resulting Boolean network to improve the quality of the result of the next step, technology mapping. These steps try to reduce the number of simple logic gates in the Boolean network or reduce its logic depth. Some specific structures such as multipliers or adders may directly be synthesised into predefined implementations obtained from a library using LUTs, carry chains, DSP blocks or RAM blocks.

Technology Mapping

During technology mapping, the Boolean network generated by the synthesis step is mapped onto the resource primitives available in the target FPGA architecture. The basic primitives of most FPGAs are considered to be the LUT and the FF. The LUT is limited by the number of inputs it has. Other primitives, such as DSPs, are typically inferred directly from HDL and not mapped during technology mapping.

The result of this step is a netlist with elementary blocks (LUTs, FFs, DSPs, etc.) (Figure 2.12c). The technology mapper tries to minimize the depth, i.e. the number of LUTs on the longest path from input to output, and area, i.e. the total number of LUTs, of the netlist.

Packing

During the packing step, LUT primitives, flip-flops, memory slices and DSP slices from the mapped netlist are clustered into functional blocks, taking the connectivity possibilities of the functional block and the design constraints into account. The packing algorithms try to pack the primitives with critical connections together in one block, because intra block connections are typically faster. In order to minimize the routing congestion, packing tools also try to pack closely connected primitives together. This minimizes the number of connections between functional blocks that have to be realised by the routing network. Additional constraints may have to be taken into account: e.g. on Xilinx FPGAs only one set of clock, clock enable and reset signals is allowed per CLB. The result of the packing step is a netlist of functional blocks (Figure 2.12d). In Xilinx' latest FPGA tool flow, Vivado, this step is being merged with the placement step [145]. In Chapter 4 we propose a new multi-threaded packing algorithm in which the netlist of functional blocks is partitioned while trying to preserve the design hierarchy.

Placement

During the placement step, functional block sites on the FPGA are assigned to implement the instances of the functional block primitives in the packed netlist (Figure 2.12e). Placement takes into account the timing, wirelength and congestion of the design [145], e.g. by placing strongly connected blocks or blocks with critical connections in-between closer together. The placement problem is a NP-complete problem. Approximate solutions are found by placement heuristics. Simulated annealing is the conventional heuristic used to produce placements. Simulated annealing for the larger designs can take several hours. In the last decade analytical placement techniques have become more popular to speed up the FPGA placement process. Analytical placement techniques produce a placement in two steps, a global placement prototyping step and a refinement step. In Chapter 5 we propose a new Steepest Gradient Descent based technique for global placement prototyping.

Routing

During the subsequent routing step, the routing resources are assigned to realise each net as described in the mapped circuit netlist without causing short-circuits, see Figure 2.12f. The FPGA's interconnection network not only requires the larger portion of the total silicon area in comparison to the logic available on the FPGA, it also contributes to the majority of the delay and power consumption. Therefore it is essential that routing algorithms assign the routing resources as efficiently as possible. Additionaly the routing solution of the design determines the maximum clock frequency of the circuit. Routing therefore has to take into account the timing constraints and the contributions of all nets to the critical path, but also the availability of the scarce routing resources. Unfortunately the routing problem is a NP-complete problem and in most cases the most time-consuming physical design step. Routing for the larger designs can easily take multiple hours. For a given architecture and design it is not always sure if a routing solution can be found. If a routing solution is found, the routing infrastructure settings are extracted and used to compile a FPGA configuration bitstream. The conventional methods to solve the routing problem are based on the PATHFINDER algorithm [100]. The PATHFINDER algorithm rips up and reroutes nets iteratively until no more congestion exists. The cost of overused routing resource is increased in each iteration. To reduce the routing runtime, only the congested nets are ripped up and rerouted. In Chapter 6 we propose a more refined version of the PATHFINDER algorithm, which is called the connection router. The connection router is able to reduce the runtime by partially rerouting nets. To achieve this, the main congestion loop of the connection router rips up and reroutes only the congested and critical connections, which allows the router to skip congestion free parts of large fanout nets and consequently converges much faster to a solution.

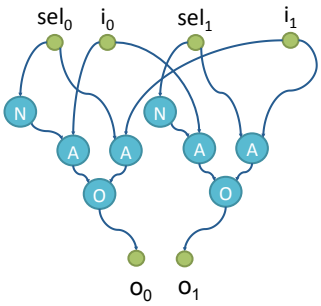
2.2.3 Compilation Runtime

Since the introduction of high-level synthesis [2] and the emergence of new markets [123], more and more engineers with a software background attempt to accelerate applications with an FPGA. They are used to gcc-like compilation times and their design methodologies are adapted to these short compilation times. In order to fix bugs and measure the performance of their design, the compilation is performed numerous times to evaluate if the design meets the constraints of the application. Hence, they cannot accept the long compilation times that are common in FPGA design. This is an important reason for research

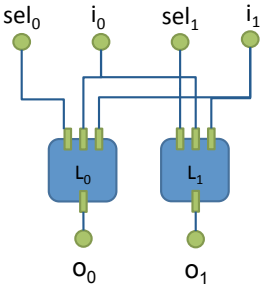
```
entity crossbar is
port(
  sel : in  std_logic_vector(1 downto
        0);
  i   : in  std_logic_vector(1 downto 0)
        ;
  o : out std_logic_vector(1 downto 0);
);
end crossbar;

architecture behavior of crossbar is
begin
  o(0) <= i(to_integer(sel(0)));
  o(1) <= i(to_integer(sel(1)));
end behavior;
```

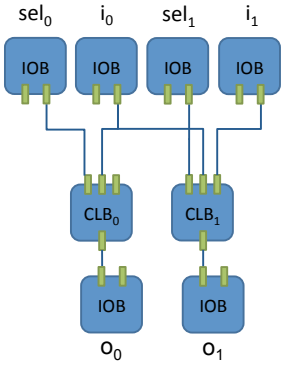
(a) HDL design.



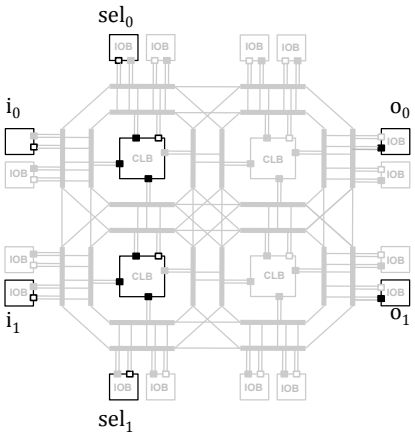
(b) After Synthesis.



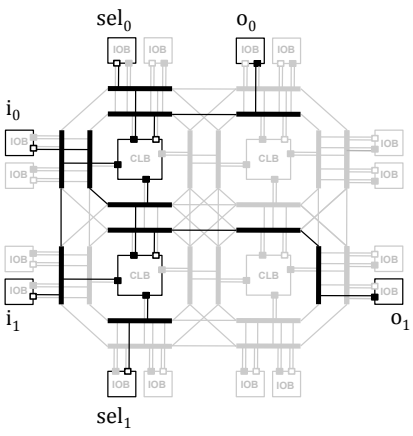
(c) After Technology Mapping.



(d) After Packing.



(e) After Placement.



(f) After Routing.

Figure 2.12: Intermediate results of the different steps of the FPGA tool flow.

into speeding up the CAD algorithms. The most problematic runtime consuming steps of the FPGA design compilation are placement and routing.

Additionally we still observe an increase in both the size of applications and target devices following Moore's law [117]. Moore's law states that for integrated circuits the density of transistors at which the cost per transistor is the lowest increases at a rate of factor two per year. The increase in size for designs and target FPGAs makes it hard to keep the CAD tools scalable in terms of runtime and memory requirements. To overcome this problem, device manufacturers and academics developed multi-threaded versions of the previously serial implemented algorithms [18, 51, 63, 85]. Although seemingly successful, the algorithms cope with problems such as serial equivalence, being deterministic and the fact that algorithms are not designed to scale in terms of number of threads.

In this thesis we describe novel concepts and improvements for packing, placement and routing algorithms in Chapters 4, 5 and 6 that improve the compilation runtime - solution quality tradeoff.

2.2.4 Related Work

Open-source projects such as VTR [88] and RapidSmith [77] aim to build academic FPGA tool flows. VTR consists of the ODIN II synthesis tool, the ABC logic optimization and technology mapping framework and VPR the Versatile Place and Route framework. VPR was designed to investigate FPGA architectures and tools. VPR has shown some flexibility to investigate architectures, but it serves less for prototyping CAD algorithms because it is implemented in C and is only recently being ported to C++. C/C++ are low level languages, making it easier to build faster tools, but it requires more work and time to prototype new CAD algorithms. The algorithmic improvements described in Chapters 4, 5 and 6 to the packing, placement and routing steps are implemented in an abstract and heavily object-oriented fashion in Java, making it easier to try out new algorithms. In the future we plan to combine the tools and release them in one framework [136], making it easier for academics to try out new CAD algorithms.

In Chapter 3 we describe a considerable gap between the quality of results obtained by the VTR framework and the Xilinx' framework, which influences research conclusions made in the academic community.

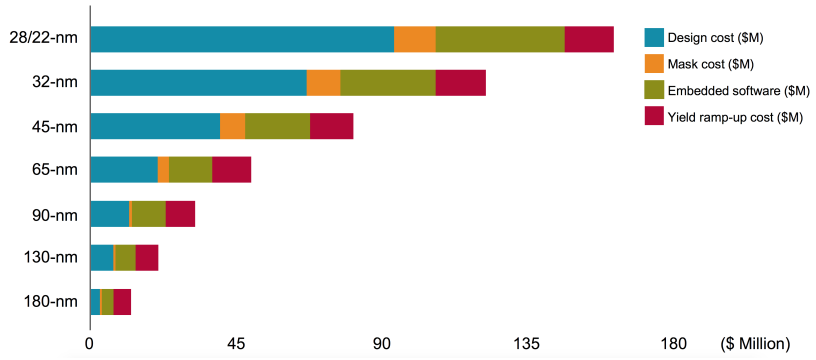


Figure 2.13: Initial chip design cost, by process node, worldwide [62] The design cost is the cost to design the chip in the specific technology process node.

2.3 The History of the FPGA

In this section we give a historic context of the FPGA that allows to put the work in this thesis in a larger timescale perspective. We largely follow the three ages of FPGA as suggested by Steve Trimberger in [127, 128]. Steve Trimberger is a research fellow at Xilinx and an important voice in the academic community on reconfigurable architectures.

2.3.1 FPGA versus ASIC

An application-specific integrated circuit (ASIC) is a chip customized for a particular application, rather than intended for a general-purpose use. In the 1980s FPGAs performed poorly on all important aspects in comparison with ASICs, such as price per unit, low power, high capacity and high speed. Despite the poor performance they were still becoming popular because of the high non-recurring engineering (NRE) cost involved with producing an ASIC. Producing an ASIC requires to not only design the transistor level circuit, but also design the masks required to produce the silicon wafers. The NRE cost is a huge threshold for new companies that want to accelerate their application. The FPGA vendors can amortize the NRE costs of the ASIC development over all their FPGA customers. Additionally FPGAs allow a faster time-to-market, because the FPGA customers save time by avoiding the low level engineering work involved with producing an ASIC.

The bar chart in Figure 2.13 shows that the NRE cost for ASICs increased as the process technology scaled down, which pushed the cost crossover point favourably towards FPGAs. Today, ASIC customers use an older process technology, which reduces the NRE cost but also

decreases the gap in performance and per unit cost gain. Additionally the FPGA vendors reduce the risk for developing a hardware accelerator by raising the abstraction level for designing hardware and eliminating low level design problems. Bugs could be corrected by reconfiguring the FPGA in contrast to ASICs, for which new masks need to be produced, making it a costly and timing intensive design cycle.

2.3.2 Age of Invention (1984-1991)

In the age of invention of FPGAs cost containment was critical and the most important parameter determining the cost was silicon area. Every advantage in process technology and architecture was exploited to reduce the area of the die. FPGAs are more scalable in size than other programmable architectures such as Programmable Array Logic (PALs). A PAL is a programmable AND array which generate product terms and a fixed OR gate that combines the terms. The number of programmable points of a PAL architecture grows with the square of the number of inputs which puts PALs at a big disadvantage.

Cost efficient programming technologies such as antifuse architectures were popular with the largest FPGA at the time being an antifuse device, the Actel 1280. They only used one transistor, but could only be programmed once. SRAM cells based architectures were at a disadvantage, because SRAM cells require 6 transistors and are volatile. Xilinx used 4-input LUTs but FPGA architects looked at finer grained architectures to increase the efficiency with smaller LUTs or NAND/XOR gates.

As cost was really the number one issue and consequently silicon area too, the interconnect wiring was designed as efficient as possible with only short wires. Early FPGAs were starved for interconnect and consequently notoriously hard to route.

The Rise of FPGA design Tools

Automated design tools such as placement and routing were not yet considered essential in the age of invention, but each FPGA vendor had his own architecture, which prevented universal FPGA design tools. For customers it was hard to tell if a function would fit the FPGA. The performance of the design was hard to predict and dependent on the placement of the logic and routing of the inter-block connections. To aid the customer to get the most out of the FPGA, the FPGA vendors started developing their own CAD tools. Here lays an important reason for the gap between commercial and academic research results described in Chapter 3.

2.3.3 Age of Expansion (1992-1999)

FPGA startup companies were fabless manufacturers, they designed and sold FPGAs but outsourced the fabrication of the FPGAs to specialized manufacturers, the semiconductor foundries. As a result they could not use the leading edge process technology, but in the 1990s they became the process leaders as foundries realized the value of using the FPGA as a process-driven application. Foundries were able to build **SRAM** FPGAs as soon as they could yield transistors and wires in the new technology. So FPGA companies caught up on the recent technology process and could ride the technology wave fuelled by Moore's law. Additionally the introduction of chemical-mechanical polishing allowed stacking more metal layers on the die, which made interconnect much cheaper. This had several effects. The area became less precious. The device capacity of the FPGA increased exponentially and applications became too large for manual design, so FPGA CAD tools became the main way to program FPGAs. The FPGA vendors choose automation-friendly architectures with regular and plentiful interconnect resources to simplify algorithmic decision-making.

Cheaper wire also admitted longer wire segmentation, interconnect wires that spanned multiple logic blocks. Wires spanning many blocks effectively make physically distant logic logically closer, which improved performance.

The survivors in the FPGA business were those that leveraged process technology advancement to enable automation. An important example is the emergence of LUT as logic cell of choice. A LUT implements any function of its inputs which made technology mapping easier and the LUT's logic equivalent pins reduced the complexity of the placement and routing problem.

The FPGA vendors were/are locked in a race to the next process technology node. Only architectures that were the easiest to adopt to the next process technology survived. An example is the emergence of SRAM as technology of choice. It contains more transistors than antifuse, flash, ... but it can be built with only transistors and wires. The other technologies were only qualified months or years later, after the first introduction of the process technology node.

FPGA Capacity Bell Curve

The bell curve in Figure 2.14 shows the histogram of distribution of sizes of ASIC applications. In the age of expansion FPGA capacity grew following Moore's law. The growth in FPGA capacity was faster than the FPGA application size, which allowed FPGA vendors to address

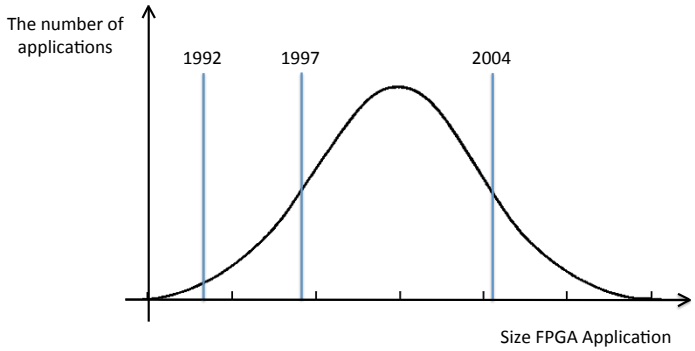


Figure 2.14: The growth of the FPGA addressable market. Reproduced from [128]. The size of the bell curve changes over time, but the shape stays constant. The vertical lines indicate the year when the FPGA became large enough to cover the left part of the bell curve starting from the line.

customers with larger applications. A small increase in capacity admitted a large number of new applications which further fuelled the growth of FPGA business. Until the end of the age of expansion the FPGA vendors successfully absorbed nearly the entire low-end of the ASIC business.

2.3.4 Age of Accumulation (2000-2007)

In the 2000s FPGAs became larger than the typical problem size, which put them over the top of the bell curve of the typical applications size and this leads to diminished returns for the number of new applications that could be implemented by increasing the FPGA capacity. FPGAs could contain large designs which implemented complete subsystems. The cost and power became more important.

The winning application of the FPGA was found in the communications industry, with customers as Cisco. The sales to the communication industry amount to half of the total FPGA business. This large market segment drives FPGA vendors to customize their FPGAs for the communication industry, which lead to the introduction of high-speed flexible IO and source-synchronous transceivers, thousands of dedicated high performance multipliers, the ability to make wide data paths and deep pipelines for switching large amounts of data with-

out sacrificing throughput. Soft logic IP was developed to save design effort, most notably the communication standard protocols, bus protocols, microprocessors and other function generators.

The Dennard scaling states that power density stays constant as transistors get smaller. The Dennard scaling ended in the mid 2000's. A new technology generation still gave improvements in capacity and cost. Power also improved but with a clear tradeoff against speed performance. To increase the performance the FPGA vendors relied more on hardening commonly used functions.

The high capacity FPGAs were too expensive for the customer that only needed a low capacity low cost device. So new FPGA families were dedicated to this market segment, with low cost and low capacity FPGAs, such as the Xilinx' Spartan and Altera's Cyclone family.

2.3.5 Current Age

In the current age ASICs are only feasible for the very high volume products, which are microprocessors, memories, cell phone processors and FPGAs.

The slowing process technology improvements push the FPGA architects to investigate the viability of novel FPGA circuits and architecture, without degrading the ease of use. In Chapter 8 we investigate a new architecture for the routing switches.

FPGAs are becoming complete systems-on-a-chip, but suffer from programming complexity. They require a large design effort to be used effectively. Designers looking for hardware acceleration are attracted to multicore systems, such as Graphics Processing Units and software programmable Application Specific Standard Products (ASSPs), that provide pre-engineered systems with software to simplify the mapping problems onto them, e.g. Lasagne and TensorFlow simplify mapping neural network training procedures to GPUs [41, 72]. The designers sacrifice some of the flexibility, performance and power efficiency for ease-of-use and fast prototyping.

Two important directions can help to ease the design effort. The first one is the elevation of the abstraction level of programming hardware. New High-Level Synthesis (HLS) tools are released by the main FPGA vendors [10, 144]. Xilinx introduced a python library called PYNQ to program their SoC Zynq devices [92]. The second one is speeding up the design cycle. Large designs can take up to hours to compile, which makes it hard for the designer to apply trial and error approaches for meeting the design constraints. In Chapters 4, 5 and 6 we present new

techniques to improve the runtime of the most time consuming compilation steps of the FPGA tool flow.

FPGAs still have an important edge over GPUs. They are much more flexible and they serve a broader range of applications. They excel in the speed performance versus power consumption metric, which is a big advantage in datacenter and wearable application domains. A convolutional neural network showcase in a Microsoft research whitepaper shows that an Nvidia Tesla K20 GPU consumes 235 W to evaluate a seven layer network on the Imagenet 1K dataset at 376 images per second. Using FPGAs the power consumption is reduced with an order of magnitude to 25 W at a slightly reduced throughput of 233 images per second [109].

The inclusion of ADCs in the Xilinx' Virtex 7 family and the new Xilinx Ultrascale+ Zynq SoC devices with embedded GPUs suggests that the FPGA SoCs move from a pure hardware accelerator to a prototyping platform.

2.3.6 Current State of FPGA Vendors

The current FPGA market can be considered dominated by a duopoly. Xilinx and Altera have product lines which are similar. They, respectively, hold 50% and 39% of the market share [105]. Intel bought Altera in 2015 for 16.7 billion USD. The press release states that the two main reasons were datacenter integration and a better market position in respect with Xilinx because Intel has its own semiconductor device fabrication division. The remaining market share is divided by Microsemi and Lattice. Both companies focus on low power and low cost. Microsemi focuses on FPGAs with non-volatile configuration memory.

3 The Divide between FPGA Academic and Commercial Results

The pinnacle of success for academic work is achieved by having impact on commercial products. In order to have a successful transfer bridge, academic evaluation flows need to provide representative results of similar quality to commercial flows. A majority of publications in FPGA research use the same set of known academic CAD tools and benchmarks to evaluate new architecture and tool ideas. However, it is not clear whether the claims in academic publications based on these tools and benchmarks translate to real benefits in commercial products.

In this chapter we compare the latest Xilinx commercial tools and products with these well-known academic tools to identify the gap in the major figures of merit.

3.1 Introduction

Commercial Field-Programmable Gate Arrays (FPGAs) have been rapidly growing in both capacity and performance, opening the door to a large number of applications. Advances in process technology along with FPGA CAD tools and architecture have enabled this growth. Further advances in both tools and architecture are required to sustain this growth. Potentially, academic research efforts in these areas could contribute to this success by identifying promising tool or architecture ideas. This is especially important as FPGAs serve a wider range of ap-

plications compared to ASIC or ASSP counterparts in semiconductor business. FPGA architecture and tool ideas that are seeded from the FPGA academic community have decreased significantly over the last decade.

The few that are proposed do not offer significant benefits when incorporated and evaluated in a commercial framework. If this trend continues, the academic work in this area might become irrelevant. This will adversely impact both FPGA industry and academic community, as the products can no longer leverage the broader academic ecosystem. We claim one of the main reasons of this trend to be the significant performance gap between the academic and commercial framework. We try to examine this claim by comparing the most prevalent academic architecture tools with Xilinx Vivado used for UltraScale devices [150, 89]. When academic tools lag behind the state of the art by a large amount, it is easy to show improvement, but those improvements do not translate to any benefits for commercial tools and devices. After identifying the gap, we investigate the historical reasons for how the divide came to be, we try to provide guidance on how to reduce it and also provide a few rules of thumb for assessing the merits of academic work.

The work described in this chapter is largely done at the CTO Research lab of Xilinx during a six month internship. The research was done in close collaboration with Alireza Kaviani, principal engineer at Xilinx and head of the next generation FPGA architectures group and Henri Fraisse, senior staff software engineer at Xilinx. The work was presented at ICFPT2015 and is described in [134].

3.2 Background and Related Work

The most popular academic open source tools used for FPGAs are Versatile Place and Route (VPR) [89] and ABC logic optimization and technology mapping [1]. There is also a front-end synthesis tool, called ODIN II [110], which takes a Verilog design and performs RTL elaboration. A recent academic framework, called Verilog-To-Routing (VTR), combines ODIN II, ABC and VPR to offer a complete unified flow for FPGA compilation [61]. We chose this well-known academic framework as our academic reference because it is the most flexible framework available. It gives researchers control over every part of the framework from architecture to tools and benchmark designs. The front-end synthesis step produces a Berkeley Logic Interchange Format (BLIF) file, which is read by ABC to perform logic optimizations and technology mapping to LUTs. VPR then packs the LUTs and FFs into

CLBs, places the CLBs and routes the whole design. There are three main components in an evaluation framework: the target FPGA architecture, the CAD tools and the benchmark designs.

There are two works that have also raised the issue of the gap between commercial and academic results. The first one provides the ability to compile designs for commercial devices using a VTR-to-Bitstream (VTB) flow presented in [61]. The VTB flow translates the mapped and placed circuits to an XDL description, a format provided by Xilinx ISE tools. These XDL text descriptions are translated to binary files and subsequently the design is routed with Xilinx ISE's PAR. VTR supports FPGA routing but is unable to model complex routing structures that exist in Xilinx FPGAs. New advances in the VTB project [60] enables routing designs on Xilinx' older architectures, such as the Virtex 6. The routed designs are analysed by Xilinx ISE's TRCE static timing analyser to get reliable timing information. Our analysis is different in several aspects: we use the most recent commercial and academic tools (in contrast with a decade old commercial tools) and show that the divide is actually much wider now. The focus of VTB is to realize a design on a commercial device and they achieved that goal. However, XDL and the relevant flow are no longer supported by Xilinx and the proposed flow will unfortunately not work with the latest products, such as UltraScale. A second work [106] focuses on addressing the mismatch in benchmark designs by providing larger designs for the open source community. They contributed 23 large benchmark designs and 20 mid-size designs. They identify the critical path delay gap as 50%, but they use a commercial tool for synthesis providing a hybrid evaluation flow and this gap is only measured on the benchmarks that did not fail. Their hybrid flow was only able to place and route 13 of the 23 large designs. They are also comparing to older 40nm products from Altera. This framework can easily be used to test advancements in place and route tools. In contrast to previous work, we focus on a new comparison to identify the gap for the most recent products and tools and show that it is much wider than stated in the literature. If the quality of academic tools is inadequate such that the required figures of merit are not met, there is little value in implementing those designs on commercial products using the VTB flow. We also address the area-efficiency and runtime scaling gaps. We then take a deeper dive in one of the academic tools, ABC logic optimization, to show that it is possible to achieve quality results on par with commercial tools with some effort.

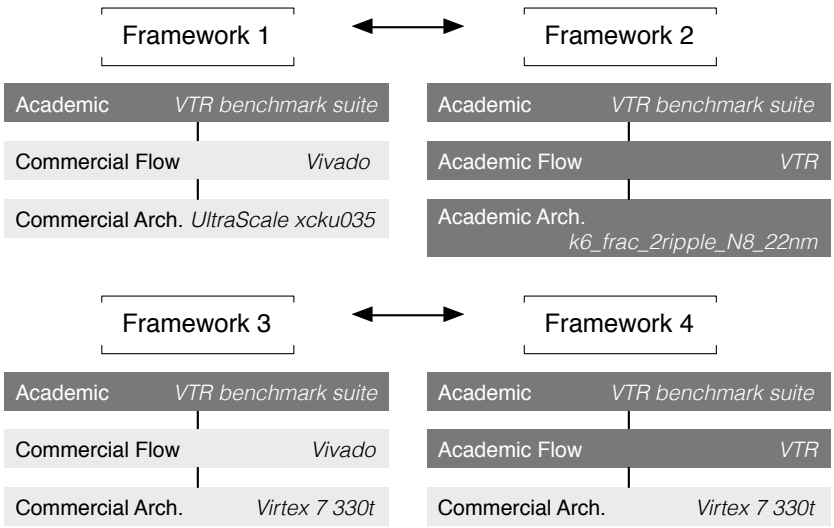


Figure 3.1: The four frameworks used to compare commercial and academic flows

3.3 Commercial and Academic Tool Comparison

In order to make a fair comparison, we should use the same benchmark designs and the same target architecture. Unfortunately, VTR is not designed to compile for commercial architectures. First, we compare Vivado and VTR for the architectures available for the latest comparable process technology nodes, Framework 1 vs Framework 2 in Figure 3.1. Later, we will use the VTB flow to target a commercial architecture and assess if the gap diminishes, Framework 3 vs Framework 4 in Figure 3.1.

3.3.1 Evaluation frameworks

We selected the smallest 20nm Ultrascale Kintex device (xcku035) with the largest package (ffva1156) and the fastest speed grade to match that of academic architectures. Together with the Vivado 2014.3 tool flow, we call this the commercial implementation. The academic target device is the most advanced architecture closest to 20nm available in VTR. We choose *k6_frac_2ripple_N8_22nm*, because it performed best in terms of speed-performance of all the architectures available in VTR. We will call this architecture VTR-22nm from here on. The original architecture was sized for a 22nm high performance process and we needed to resize the transistor-level circuit for this architecture so that both com-

mercial and academic devices are optimized for the same nominal operating voltage (0.95V). We used an automatic transistor sizing tool [30] and 22nm predictive technology models optimized for high performance [155]. It is worth noting that the process technology for xcku035 is a low power process technology, and hence our speed-performance results for the academic flow will be somewhat optimistic. The VTR-22nm architecture contains carry chains, fracturable 36x36 multiplier blocks, and fracturable 32Kb memory blocks. Each CLB contains 8 fracturable LUTs similar to that of the xcku035, but contains only one flip-flop per LUT and no distributed memory capabilities. The routing architecture was kept simple with only length-four wires. We refer to the VTR-22nm architecture together with VTR tool flow revision 4591 as the academic implementation and use it as a reference in this section. As benchmark suite we used the 19 designs available in the VTR framework. The results for the most important figures of merit (area, maximum clock frequency) are listed in Table 3.1 and will be discussed in the following subsections.

3.3.2 Speed-performance

Vivado is designed to compile a design for a set of known constraints and not to find the highest possible operating frequency for a given design. To find the maximum clock frequency we started with constraining the designs with a clock period that could be easily met. Subsequently the data path delay of the most critical path in the clock domain was used as a new constraint for the clock period. We repeated this process until Vivado just failed the constraint with a violation of less than 1ns. Another approach could be to constrain the design with an unrealistic clock period like in VTR, for example 1ns, but Vivado would recognize that it could never meet the constraint and it would exit early. Therefore the latter approach is not an option for the commercial flow. As noted in Table 3.1, the maximum clock frequency for all benchmark designs is higher for the commercial implementations compared to the academic implementations. The geomean of the maximum clock frequencies of the commercial implementations is 2.24 times higher than that of the academic implementations. We believe this 2.24X divide in quality of results is an important result. It indicates why many academic FPGA architecture and tool improvements cannot translate to realistic benefits for FPGA industry. This wide gap includes architecture and tool differences, but excludes differences caused by benchmark designs and process technology. Referring to previous work [60], we may also estimate that 50% of this divide is due to synthesis and the rest is

Table 3.1: Overview of the post-routing results for the VTR benchmarks

	Academic (VTR - k6_frac_2ripple_N8_22nm)						Commercial (Vivado – UltraScale)								
	Area				Fmax	Runtime	Area					Fmax		Runtime	
Benchmarks	CLB	Mult	Mem	Norm*	(Mhz)	(min)	CLB	DSP	BRAM	Norm*	rel	(Mhz)	rel	(min)	rel
bgm	4259	11	0	4424	52	10.4	2150	22	0	2260	0,51	183	3,52	6.3	0.61
blob_merge	717	0	0	717	96	4.7	1437	0	0	1199	1,67	364	3.79	2.8	0.60
boundtop	280	0	1	300	146	4.5	813	0	1	836	2,79	367	2,52	2.8	0.63
diffeq1	33	5	0	108	64	4.3	78	9	0	123	1,14	135	2,11	1.6	0.36
diffeq2	21	5	0	96	81	4.3	34	9	0	79	0,82	149	1,84	1.5	0.35
LU8PEEng	2645	8	45	3665	16	8.5	2534	16	23	3143	0,86	24	1,51	5.2	0.61
LU32PEEng	8794	32	168	12634	16	25.1	8867	64	136	12315	0,97	23	1,46	9.2	0.37
LU64PEEng	17028	64	340	24788	16	59.4	15574	128	188	20538	0,83	26	1,63	18.5	0.31
mcml	8137	27	159	11722	27	32.1	6988	104	154	11050	0,94	55	2,01	13.2	0.41
mkDelayW∼	755	0	43	1615	117	5.2	140	0	27	761	0,47	645	5,51	1.9	0.37
mkSMAda∼	210	0	5	310	158	4.6	193	0	3	262	0,85	491	3,11	2.3	0.50
or1200	308	1	2	363	102	4.7	365	4	1	408	1,12	176	1,73	1.9	0.40
raygentop	266	7	1	391	148	4.6	390	9	0,5	446,5	1,14	469	3,18	1.9	0.41
sha	244	0	0	244	179	4.6	212	0	0	212	0,87	299	1,68	2.2	0.47
stereovision0	1195	0	0	1195	245	4.9	1013	0	0	1013	0,85	635	2,59	2.6	0.52
stereovision1	1916	46	0	2606	149	5.6	2511	0	0	2511	0,96	337	2,27	4.1	0.73
stereovision2	3290	201	0	6305	100	7.9	2213	270	0	3563	0,57	136	1,36	4.6	0.58
stereovision3	22	0	0	22	270	4.4	30	0	0	30	1,36	474	1,76	1.3	0.29
Geomean											0.95		2.24		0.46
Geo. Std. Dev.											1.52		1.45		1.30
Total						199.8								83.9	

* Normalized Area: the total area expressed in terms of CLB tiles, it includes the DSP/Mult and Mem/BRAM usage, see equation (1) and (2)

from the place and route portion of the flow and the architecture difference. We will discuss this further in the following sections.

3.3.3 Area-efficiency

Comparing the area-efficiency between commercial and academic implementations is more difficult because of the different hard blocks in the target architectures. The VTR-22nm architecture contains fracturable 36x36 multiplier blocks and fracturable 32 Kb memory blocks. The Ultrascale fabric has versatile DSP48E2 blocks that can implement 27x18 multiplications, 48-bit addition/subtraction, XOR, and some additional functionality. It also contains fracturable 36Kb block RAMs. Comparing the multiplier logic consumption for the academic and commercial implementations, we find the commercial DSP usage to be about twice the amount of academic multiplier block usage. This corresponds with the size of the respective multipliers. The only exception is *stereovision1*. The default behaviour of Vivado is to implement the divisions in *stereovision1* without DSP blocks in contrast with VTR. This leads, however, to an increased LUT count for Vivado, but a faster circuit. Vivado chooses the most delay-optimal implementation if there are enough resources available. Academic implementations typically use more memory blocks than the commercial ones. The Ultrascale fabric has slightly larger memory blocks, but that is not the main reason. The VTR benchmark designs contain a lot of shallow memories and Vivado implements these shallow memories with distributed memory. To overcome the issue of comparing resource usage for different types of hard blocks, we define a normalized area measure. The measure is expressed in terms of CLB tiles and encompasses the CLB count and all hard block occurrences:

$$Area_{norm,academic} = n_{CLB} + k_{mult} \cdot n_{mult} + k_{mem} \cdot n_{mem} \tag{3.1}$$

$$Area_{norm,commercial} = n_{CLB} + k_{DSP} \cdot n_{DSP} + k_{BRAM} \cdot n_{BRAM} \tag{3.2}$$

For the academic area constants k_{mult} and k_{mem}, we use the minimum transistor width count as reported in the architecture description for each type of hard block. We compare it to the area used for one CLB tile in the newly sized academic architecture [88]. Taking the interconnect area into consideration, we set the constants to $k_{mult} = 15$ and $k_{mem} = 20$. For the commercial area constants k_{DSP} and k_{BRAM} a similar approach to the academic calculations is used, but we scale the block areas based on multiplier bits and memory bits. Each DSP pair and memory block pair has a height of 5 CLBs, as seen in Figure 3.2

Figure 3.2: RAMB and DSP height vs CLB height, image taken from Vivado Design Editor

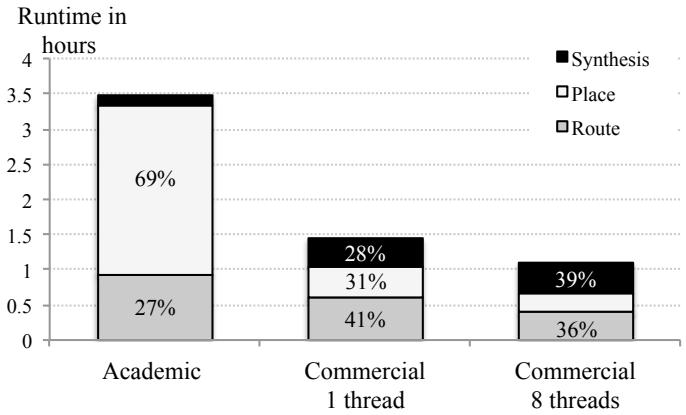


Figure 3.3: Total Runtime for compiling the VTR benchmark designs for a single run

from Vivado design editor. This results in the following area constants, $k_{DSP} = 5$ and $k_{BRAM} = 23$. The hard block occurrences and normalized area is reported for each design in Table 3.1. The commercial implementations use on average 96% of the normalized area used by academic implementations. This gap is significant, but not a showstopper in contrast with the other figures of merit we investigated.

3.3.4 Runtime

The benchmark designs were compiled on a workstation with a 3.4 GHz quad-core Intel Core i7-3770 processor and 32 GB memory. In Table 3.1, the runtime for each benchmark design is reported for the commercial and academic flow. VTR can only operate in a single-threaded

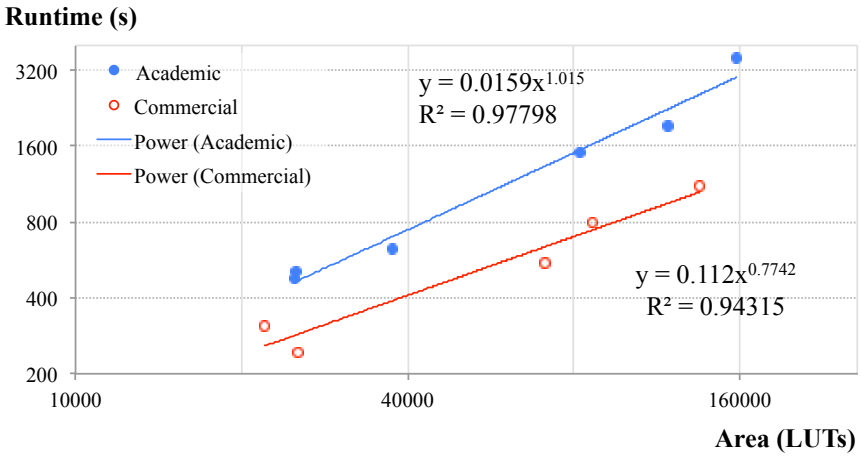


Figure 3.4: Runtime scalability of the Academica and Commercial CAD tools with respect to the size of the design

mode, so to be fair we compare it to Vivado restricted to only run a single thread. Even in single-threaded mode Vivado is on average 2.2X faster than VTR. This runtime gap is consistent with a geometric standard deviation of 1.3. All benchmark designs compile faster when using Vivado with runtimes ranging from 73% to 35% of the runtime of VTR. Figure 3.3 shows the total runtime to compile all VTR benchmark designs for both the commercial and academic flow and a breakdown for each major step in the compilation. As a reference we also included a run in which we let Vivado run unrestricted. On our test machine this mode used 8-threads, which decreased the total runtime from 1.45h to 1.1h. This is not a huge decrease mainly because we only deal with smaller benchmarks for which the runtime is so small that Vivado cannot fully take advantage of multithreading. Overall we see the same picture for the total runtime as for the separate benchmark designs. The total runtime for Vivado is a little under one hour and half and a little over three hours for VTR. So Vivado compiles the benchmarks in less than half the time VTR does. In Vivado the runtime is more equally divided between the three major steps, synthesis, placement, and routing than in VTR. In VTR, synthesis takes only 4% of the total runtime. The placer in VTR is responsible for biggest chunk of runtime (69%) followed by the routing step (27%).

We also considered the runtime scalability with respect to the size of the benchmark designs. We choose the LUT count as area figure and we only selected the designs with more than 20K LUTs to minimize the impact of the nonrecurring runtime cost. The runtime for smaller de-

signs is often dominated by fixed portions, such as reading or writing the files. The fixed portions don't have a significant impact on scalability. The data points for each flow are plotted in Figure 3.4 and fitted using a power regression model. The runtime scaling gap widens proportional to the equation $0.14 \cdot x^{0.24}$, where x represents the number of LUTs, so for each 60K LUT increase the runtime gap doubles. We predict that the runtime gap will increase in the same fashion beyond the 160K mark. This makes the compile time of academic flows impractical for today's FPGA application sizes, because they easily surpass the 160K LUT mark. The small number of designs shown in Figure 3.4 might not be sufficient for calculating a statistically valid scaling factor. However, the trend of growing runtime gap, which is the main message of this section, will hold as we add larger and more designs to the graph. Out of this comparison, placement is clearly a main cause of the gap in runtime scalability. At its core, the VTR placer still uses simulated annealing which is more runtime intensive and does not scale as well as the analytical placement techniques used in the Vivado placer. The academic framework could benefit greatly from an open-source analytical placer. We built an open source analytical placement tool and designed a new fast placement prototyping algorithm. They are described in Chapter 5. The academic framework also spends little time on synthesis in comparison with the commercial framework. Hung has investigated and reported his findings in [60]. His results indicated that a large part of the quality gap is caused because by the academic synthesis tools.

3.3.5 Using VTR for a Commercial Target Device

VTR is not designed to map to commercial devices but we made an attempt to use the VTB flow introduced in the previous work. In [61], the authors present a VTR-to-Bitstream (VTB) flow that enables users to map to Virtex 6 devices. Vivado does not support the older Virtex 6 devices. We extended VTB to target the Virtex 7 vx330t device with the help of the authors of [61]. We compared the frameworks 3 and 4 as shown in Figure 3.1. VTB and Vivado target the same commercial device. We choose to target the fastest speed grade and the largest footprint. This resulted in Vivado implementations that consume 25% less area and are able to operate at 2.1X higher operating frequencies on average than VTB implementations. This is more or less in line with the gap reported for the commercial versus academic comparison. The only major difference was that the VTB was remarkably slower than Vivado by a factor of 5.5X. There is only a very slight reduction

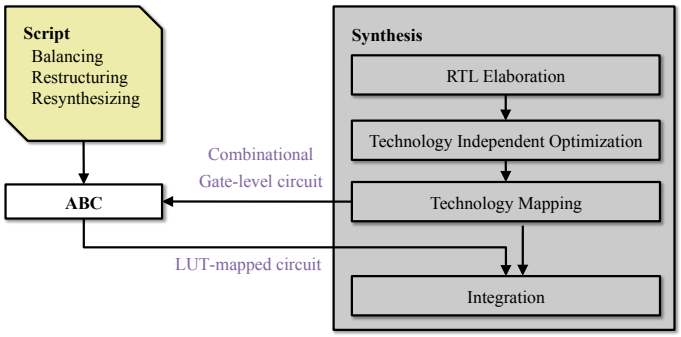


Figure 3.5: Hybrid flow for logic optimization

in speed-performance gap if we compare Vivado versus VTB targeting a commercial device (2.1X) and the commercial versus fully academic comparison (2X). We attribute this reduction in the gap to a better architecture, but conclusions are difficult here because VTB is not designed to fully exploit this commercial architecture, so the actual architecture gap could be much wider. Our initial intention for using VTB was to identify which part of the gap can be attributed to the architecture and which part to the tools. We believe the large quality gap in the tools may be misleading and hence we defer making solid conclusions on the architecture gap to future work.

3.3.6 The Reasons for the Divide

In Chapter 2 we described the three ages of the FPGA. During the first age, the age of invention, architecture efficiency was the highest good. There was no uniformity in FPGA architectures. This forced the FPGA vendors to own their own tools, because research into new architectural ideas is faster if you have in-house tools. The advantage was that it prevented the commoditization of FPGAs. A big disadvantage of the closed source tools is that the academic researchers could not build on the advances made in the industry. Additionally commercial designs are proprietary and difficult to access for common academic researchers, which makes it hard for academics to benchmark new innovative architecture and tool ideas. This leads to a one way street in which the industry cherry picks the advances in academic research, but academics have to start from scratch every time again. This is the historical reason for the growing divide between the academic and commercial results. Nowadays the closed source nature of commercial tools is further promoted by the protection of vendor's IP design cores. IP

design cores are common basic building blocks for a design which enables hardware designers to faster prototype their design.

3.4 Hybrid Commercial and Academic Evaluation Flow

We described the gap between academic and commercial tools for FPGA design implementation in Section 3.3. However, the main advantage of open-source academic tools is that they are easier to change and augment toward a research goal. The tools are often data-driven and skip unnecessary detail, helping the researcher conclude faster. The question we are trying to answer in this section is how to combine the credibility of commercial tools with the flexibility of academic tools to reach pragmatic architectural or tool conclusions. In contrast to the previous section, we use commercial tools as our baseline for assessing a new tool flow. We created a hybrid evaluation flow using Vivado and ABC [1], which is a well-known academic tool for logic optimization and technology mapping. The advantages of such a hybrid flow are two-fold: 1) we can accurately quantify the quality of logic optimization; 2) we can quickly evaluate architecture ideas or opportunities in commercial tool optimization. Even if such evaluation flow helps us detect failures for certain ideas, it will prevent researchers from investing unnecessary additional time. Figure 3.5 summarizes the hybrid flow that we created. The key to creating such a flow is identifying the best interception points to exit and re-enter the commercial Vivado flow. The Vivado synthesis tool processes the design in three steps: elaboration, architecture-independent optimizations, and technology mapping. During RTL elaboration, common data path operations such as additions and storage elements such as memory blocks are identified and inferred. Architecture-independent optimizations include constant propagation, operation sharing, strength reduction; expression optimization, finite state machine encoding/minimization, generic restructuring and don't-care optimizations. During technology mapping the optimized design is mapped onto target architecture structures, such as DSP blocks, adders with dedicated carry-chains, BRAMs, LUTs and FFs. In the new hybrid flow, the combinational portion of the logic gate network is cut out and written to a BLIF file. ABC reads in the BLIF file and performs logic optimizations and mapping as stated in the script given by the user. The script may contain commands to restructure and balance the logic network and to perform different mapping algorithms. After ABC optimizations, the circuit mapped to LUTs

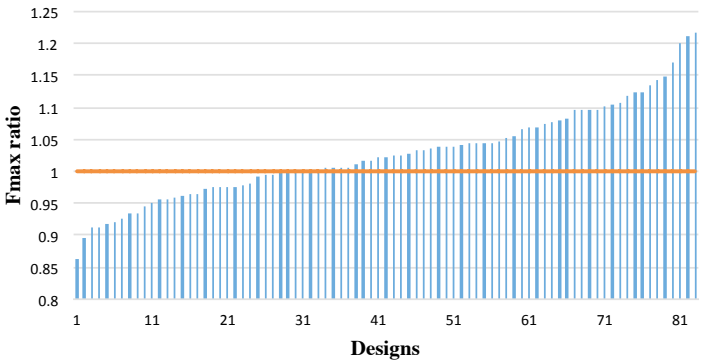


Figure 3.6: Maximum clock frequency ratio for the hybrid flow versus the Vivado baseline

is stitched back into the design in Vivado. This new flow replaces the commercial technology mapping and optimization with that of the academic flow. Initially, this new hybrid flow with ABC performed worse than the baseline Vivado flow in all figures of merits: performance, area and runtime. However, the differences were all within 2% except for runtime. This indicates that the significant performance gap we noted earlier is not due to the logic optimization and technology mapping portion of the flow. After a number of iterations and modifications to both ABC and the script, we managed to show some improvements compared to baseline Vivado. According to our results 2.5% increase in maximum clock frequency along with 1.8% decrease in area was achieved on average for more than 80 commercial benchmark designs, as summarized in Table 3.2. These modest average improvements compared to Vivado were achieved at the expense of additional runtime in ABC.

Figure 3.6 depicts the maximum clock frequency ratio for each design in the commercial suite, providing a more detailed view. Fmax improved for roughly 70% of the designs and up to 20% in the best cases. The main reason for improving the quality was less emphasis on early depth reduction. Initial scripts aggressively reduced the depth of the deepest paths in the designs, which led to worse post routing results. This is expected because at the time of technology mapping there is too much uncertainty to predict the real critical path after routing. The critical path could be dominated by routing and aggressive depth reduction will adversely affect the final results. The ABC script that produced the best results is available in [129]. It contains multiple LUT mapping iterations interleaved with sum-of-product balancing.

Table 3.2: Summary for the hybrid flow vs vivado results. The percentages indicate relative improvement for the geomean of the ratios.

	Early depth	**Fmax**	**Area (CLBs)**
Whole suite	-16 %	+2.5 %	-1.8 %
High depth	-24 %	+ 5 %	-3 %
Low Depth	-13.5 %	+0.4 %	-1 %
Arithmetic	-7 %	+1.1 %	-1 %

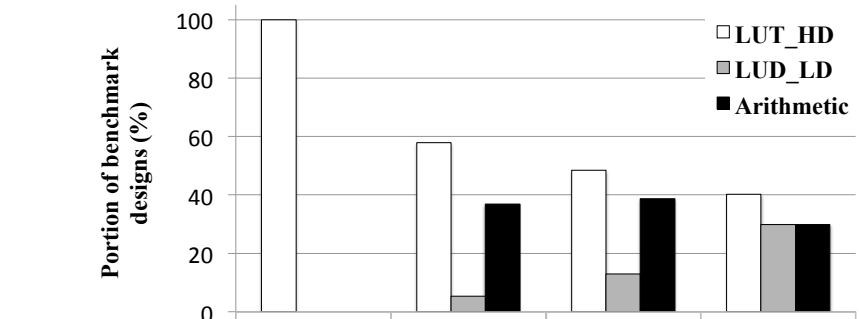
The hybrid flow helped us find the right balance between the area and depth reduction by focusing on average depth reduction and observing post-routing results from the commercial tool. It is worthwhile to note that even the initial results from the hybrid flow (before our optimizations) were within a few percentages of the baseline Vivado flow. This indicates that the synthesis gap observed in previous work is not due to logic optimization portion of the academic flows. We can make two high level observations using our hybrid flow. First, the fact that we could reach the quality of commercial tools and even improve the results for some designs shows the potential for academic tools if used in a correct framework. The second high level conclusion from this exercise was that depth reduction does not translate to post routing improvement directly. A good rule of thumb to estimate post-routing benefits of the academic work that claim improvements by reporting depth reduction is to divide the gain by an order of magnitude. We further investigated this by focusing on depth-oriented designs and confirmed that ABC indeed improves the results by 5% on average on these designs. This is a significant improvement even for commercial products and we will elaborate on this classification more in the next section. We also used this evaluation flow to dismiss some of the published architecture ideas and tool optimizations quickly without additional expensive investments in changing the commercial flow. For example, previous work has suggested using cascaded LUTs [115, 101] as potential FPGA architecture improvements, because they improved area and the depth of the circuit. Since these ideas are often implemented in ABC framework we used our hybrid flow to evaluate some of them. We found that conclusions that are mainly based on early depth reduction will not hold after routing the designs. Another example is the and-inverter cones [111]. In this case the authors further investigated their claims in a second publication [154] and came to the conclusion that the observed benefits after technology mapping did not translate to post-routing improvements. We experienced it ourselves in the work done on logic in routing nodes explained in Chapter 8.

3.4.1 Benchmark Design Suites

An important aspect of any evaluation framework is the benchmark designs. In this section we focus on highlighting the differences between academic and commercial benchmark suites.

Academic benchmark designs Unfortunately it is hard to separate benchmark designs from the framework they were written for, so we inspect the benchmark designs with their framework in mind. Typically used in academia are the well-known evaluation frameworks such as the VPR framework [89], the VTR framework [88] and the recent Titan framework [106]. The Versatile Place and Route (VPR) framework consists of 20 large benchmark designs synthesized by the Microelectronics Centre of North Carolina (MCNC). VPR is used as place and route tool and a homogeneous LUT-only architecture at 48nm technology node as a target architecture. The MCNC benchmarks are still quite popular [5, 40, 50, 116] and the VPR framework is still maintained as part of the VTR framework. The Verilog-To-Routing framework (VTR) [88] includes several benchmark designs described in Verilog. There is a range of architectures that can be targeted in VTR and researchers can add or tweak their own architecture. We used the most advanced architecture available, called *k6_frac_2ripple_N8_22nm*. Researchers working on applications or tools will probably not change the default architectures provided in the VTR framework. The Titan framework [106] consists of 23 large benchmarks and 20 mid-sized benchmarks. They are synthesized with Quartus II and VPR is used for backend of the flow to map to an architecture closely matching the Altera Stratix-IV architecture [38]. They used identical hard blocks, but the routing architecture was only modelled approximately. Unfortunately VPR does not succeed at routing 13 of the 23 large designs because of memory requirements or routing congestion as also reported by Murray et al. in [106].

Comparing with commercial benchmark designs We profiled more than 80 industry benchmark designs in order to understand the differences with academic designs. The commercial benchmark designs were part of the Xilinx Quality of Results benchmark design suite. They are proprietary designs from Xilinx' most important customers. The academic designs are much smaller compared to the industry benchmark designs we used, which typically have more than 100k LUTs. The other noticeable difference is that the majority of VTR benchmark designs are I/O-bound. They also have fewer memory and DSP components



Benchmark suite	MCNC20	VTR	Titan	Xilinx QoR
CAD tool flow	VPR	VTR	QuartusII & VPR	Vivado
Target Architecture	Homogeneous 48nm	VTR-22nm	Stratix IV*	Ultrascale

*hybrid architecture

Figure 3.7: Benchmark Suite Profiles. For each framework, the benchmark designs are classified in three categories depending on the paths and the type of instances in the critical zone of the circuit, LUT dominant and High Depth (LUT_HD), LUT dominant and Low Depth (LUT_LD) and Arithmetic dominant.

compared to industrial designs. All these differences may contribute to misleading academic conclusions in the academic frameworks. Some of these differences such as size of the benchmark designs are already highlighted in previous work [106].

In this work, we highlight and analyse another subtle, yet important difference that may skew the academic conclusions. Figure 3.7 depicts how the depth profile differs for various academic and industrial designs compiled within their respective framework. For each framework, the benchmark designs are classified in three categories depending on the paths and the type of instances in the critical zone of the routed circuit. For the commercial framework, each benchmark category contains around the same number of designs. For the VTR and Titan benchmark designs, the category of designs with a LUT dominant and shallow critical zone is under-represented. Lastly, the MCNC20 benchmark suite contains only LUT dominant designs with deep critical zones. It is clear from this comparison that the academic benchmark suites contain relatively much more designs with a high depth critical zone than the industrial benchmark suite.

We have a certain bias because we only looked at important benchmark designs Xilinx obtained from his customers. Analyzing bench-

mark designs from other important FPGA vendors such as Altera is difficult without working at the company. However, Xilinx is the market leader and accounts for more than the half of the total FPGA business, so we think our results can be generalized.

A large number of academic publications, especially those that use ABC, make conclusions based on depth improvement after technology mapping. Therefore, it is important to understand how depth reduction correlates to the end performance improvement after routing. In the next subsection we dig deeper into this depth profiling to understand the trends.

Depth Classification of the Benchmark Designs: Discussion and Trends Our depth classification is based on the profile of the critical zone in the commercial benchmark designs. We define the critical zone as all the paths in the design with 5% worst negative slack. The slack indicates whether timing is met along a timing path. A positive slack indicates that the signal can get from the startpoint to the endpoint of the timing path fast enough for the circuit to operate correctly. A negative slack means that the signal is unable to traverse the combinational path fast enough to ensure correct circuit operation. The worst negative slack is the most negative slacks of the design.

Taking into account the type of instances in the critical zone, we observe that for 68% of the designs the critical zone is dominated by LUT instances. The most occurring instance type is CARRY blocks for 20% of the designs. The DSP blocks dominate the critical zone for the 12% remaining designs. The average logic depth of the DSP dominated designs is typically lower than the carry dominated designs. We group both CARRY and DSP dominated designs in the same class, the arithmetic designs, because they show similar behaviour regarding our analysis. The remaining designs with a critical zone dominated by LUTs are further divided into 2 groups. We take into account the average depth of the paths in the critical zone for the LUT dominant circuits. 29% of the designs have an average logic depth smaller than or equal to 2. This class contains heavily pipelined designs with critical zone dominated by routing and net delays. We also refer to this group of applications as low depth. The other group contains benchmark designs with an average logic depth higher than 2. We now revisit the results of our hybrid flow explained in Section 3.4 in the context of this logic depth classification. The results are summarized in Table 3.2. The hybrid flow augmented with ABC has an average 5% higher maximum clock frequency and uses on average 3% less CLBs for the high depth, LUT-dominant designs. LUT-dominant, low depth designs show no

significant improvement in performance, but they show a 1% area reduction. For the arithmetic dominant circuits the new flow produces solutions with 1.1% higher clock frequency and 1% lower CLB usage. Our results show that ABC advantages for performance are mostly applicable to a third of the designs which have critical paths with a lot of logic levels. This is in line with the academic literature where most of ABC work is focused on depth reductions. However, the FPGA application trends are in the direction of highly pipelined designs with lower depth. Therefore, these advantages will be less pronounced in the future. The representativeness of academic benchmark suites could be improved by adding low-depth designs. Another important observation is that depth reduction no longer translates to significant post-routing delay improvement in commercial frameworks.

3.5 Concluding Remarks

We examined the divide between the quality of the FPGA configurations produced by the commercial and academic frameworks to show that it has grown beyond acceptance. For example, the speed-performance quality gap is more than 2.2X. This makes it hard to assess the merits of academic results, because it is much easier to improve something that is so far from optimal. On the other hand, we showed that it is still possible to use academic tools in a credible framework that is a hybrid with an academic logic optimization and technology mapping tool and the rest of the commercial Xilinx framework. Our results showed that close to 5% improvement is possible on average for designs with high depth paths in the critical zone. This work also highlighted a trend in industrial applications towards low depth, highly pipelined designs. Designs with shallow LUT dominated critical zones are under-represented in the academic frameworks. This further emphasizes the need for updating benchmark designs and suggests that academic tools need to focus on other optimizations such as retiming instead of early depth reduction. Academic contributions in the area of FPGA architecture and tools are still possible, but only if the wide divide highlighted in this work is addressed or academic work is done in the context of intercepting a commercial framework at the right access points in the flow. Such effort requires joint cooperation and involvement of academic and commercial interested parties. Commercial parties are often questioning the return of investment on such efforts due to the significant gap. On the other hand, some academics dismiss the importance of the quality gap as the responsibility of the industry. This is leading to a tentative stale-mate and the solution requires contribu-

tion from both parties. Industry needs to provide easier interfaces at appropriate interception points for their tools. Academics need to build hybrid flows that use commercial frameworks with the exception of the portion under investigation. In Chapters 4, 5 and 6 we use the hybrid Titan framework to evaluate our new tool improvements. Other academic FPGA work in the areas of applications or where commercial tools are evolving such as high-level synthesis is still relevant if quality of results is properly maintained. This may also imply combining them correctly with the relevant commercial framework and collaboration between industry and academic ecosystems. We also encourage academic researchers to use commercial tool flows and architectures as a baseline when possible. The evaluation framework we used is available online on GitHub [129]. It includes the VTR benchmarks partly rewritten to enable compilation with Vivado and a collection of scripts to derive the statistics used in this chapter. The gap found in this chapter has grown steadily since the invention of the FPGAs and we tried to reduce the gap by improving the tools in Chapters 4, 5 and 6. Since we are aware of the gap, we try to evaluate our innovative tool ideas with hybrid flows as much as possible, see Chapters 4, 5 and 6. This is however not always possible as is the case for the architectural research in Chapter 8 and the research in the dynamic reconfiguration of routing network in Chapter 7. The architectural research requires changes in technology mapping, packing and the architecture. Architectures are difficult to change in commercial frameworks. First and foremost because academic researchers don't have access to the source code of the commercial frameworks. However, even for the researchers at FPGA vendors, it is a time consuming job to test out new architectural ideas, because the commercial frameworks are highly optimized for a few families of architectures with the same fabric. The dynamic reconfiguration of the routing network requires changes in the benchmark design, all the steps of the compilation tool flow at the same time and access to lower level details of the architecture. For these two chapters we use academic frameworks to evaluate our results.

4

Preserving Design Hierarchy to Improve Packing Performance

In this chapter we introduce a new superior packing algorithm, which combines the two most popular approaches in conventional packing. In the introduction we explain why the packing step is introduced in the compilation flow and which metrics a packing algorithm should optimize. The introduction ends with a summary of the most important results.

4.1 Introduction

The early FPGA architectures have a flat architecture where the functional blocks contain only one programmable lookup table and a flip-flop. The functional blocks in modern architectures have a hierarchical structure to improve area and delay [96]. On each hierarchical level a number of equivalent blocks are available which are connected by a local interconnection network. The routing pathways on the lowest hierarchical level are short and hence very fast while the top level connections in the interconnection network are slower. Due to this hierarchical structure a packing step is introduced in the FPGA CAD tool flow. During packing, all low level primitives in the circuit are packed into the high level functional block types available in the FPGA architecture. These clusters are then placed and routed on the highest hierarchical level.

Several optimization criteria are used during packing such as total wirelength, area, critical path delay and power consumption. They are evaluated after the design is routed. Total wirelength (TWL) is a good measure for the number of routing resources required and it is also closely related to power consumption. The total wirelength is reduced when less connections are required between the clusters. The area is determined mainly by the number of high level clusters in the packing solution. However, if a design is too tightly packed it can lead to routing problems. Congestion prone designs require extra wire tracks in the routing channels, that leads to more area and metal layers or it forces the FPGA designer to move towards a larger more expensive chip. Minimizing the critical path delay is obtained by using the fast connections on the low hierarchical levels for the critical connections in the design. Because these fast connections are short, they also have a lower capacitance and therefore they should be favoured for connections with a high switching activity if reducing power consumption is an objective.

Packing can greatly influence the end result. We introduced two partitioning based packers that improved the quality of the routed design by preserving the design hierarchy. The best performing packing approach, called MULTIPART, reduces the total wirelength with 28% and the critical path delay with 8% on average compared with the conventional seed-based packing approach in VPR for the Titan23 benchmark suite, which targets Altera's Stratix IV FPGA. This is an important result, because it closes a major part of the gap between commercial and academic results in terms of total wirelength. Murray et al. reported a total wirelength gap of 2.19x. The improvements in MULTIPART reduce this gap to 1.58x. An additional advantage of partitioning based packers is the opportunity to implement a multi-threaded version of the algorithm. Our multi-threaded implementation has an average runtime speedup factor of 3.6 on a CPU with four cores compared with the conventional approach in VPR.

The results are reported in a conference publication [138]. The work in this chapter is done in collaboration with Dries Vercruyce. At the moment of writing, he is a Phd. student in our research group, the Hardware and Embedded Systems group. We start this chapter with describing related work and defining the fundamental problems in the current packing approaches.

4.2 Related Work

Existing packing approaches can be divided into three main classes, seed based, depth-optimal, and partitioning based packers. Seed based packers are fast, because they pack the circuit in a single pass. They produce tight packings but they are unable to escape from local minima. They lack a global overview, because they use a bottom up approach.

For each new functional block a seed block is selected with a certain optimization criterium in mind. An affinity metric between the seed block and its surrounding blocks is calculated. The block that scores the highest on this affinity metric is packed into the cluster. This is repeated until the cluster is full. Several cost functions for the affinity metric are proposed to improve the quality for a specific optimization criterium. A well-known seed based packer is T-VPack [98] and the newer version AAPack [90] which is used in the VTR tool flow [87]. This packer is used as a baseline in our experiments. The main objective of this packer is minimizing the critical path delay. A routability driven affinity metric is proposed in iRAC [120] and T-RPack [19]. Un/DoPack [126] and T-NDPack [84] try to increase routability with depopulation based clustering methods. Depopulation resolves congestion problems in the routing network by preventing that functional blocks are filled to their full capacity, thereby increasing total area because the functional logic is spread across the FPGA. P-T-VPack [75] and W-P-T-VPack [44] incorporate switching activities of the nets in the affinity metric to reduce power consumption at the cost of an increase in channel width and critical path delay. W-MO-Pack [114] and YAMO-Pack [74] are multi-objective packers that incorporate several criteria in the affinity metric to obtain good quality for all optimization criteria.

Due to synchronization problems, it is difficult to apply multi-threading to the seed based packing approaches. Each thread would have its own seed and picking blocks from the neighbourhood of that seed could conflict with the neighbourhood of the seed from other threads that run concurrently. To our knowledge, no multi-threaded seed based packer is proposed yet.

Depth-optimal methods such as TLC [37], RCP [39], and MLC [125] try to duplicate netlist primitives on the critical path to optimize the depth of all clusters. Although these methods reduce the critical path delay, they lead to an increase in total area because logic duplication is hard to control.

Chen *et al.* [28] proposed a very fast clustering method to pack the Titan benchmarks in limited time. It uses structure-aware packing and datapath extraction to handle different structures of heterogeneous

components and reduce the critical path delay respectively. Although Chen et al.'s packing approach obtains good results for runtime and total wirelength, this method leads to a large increase in the number of required functional blocks.

In partitioning based packers, the clusters are determined by performing a hierarchical partitioning of the circuit. Hierarchical partitioning is a top-down approach. First the circuit is bi-partitioned, the partitioning algorithm minimizes the number of connections cut by splitting the circuit in two. The two parts of the circuit are then further bi-partitioned independently. The resulting parts are again bi-partioned and this repeats itself recursively until the parts contain less primitives than a predefined limit.

Fully partitioning based packers are proposed in Marrakchi *et al.* [99], PPack [47], and PPack2 [48]. These partitioning based methods obtain good quality results for the total wirelength but still have some fundamental problems that we solved in our packing approaches. The first fundamental problem is due to the difficulty to impose constraints during partitioning. It is not possible to control the number of pins on each partitioned subcircuit, while functional blocks have a fixed number of input pins. Partitioning tends to minimize this number, but there is no guarantee that every subcircuit will have less than the allowable number of pins. Furthermore, it is hard to control the number of blocks in each subcircuit. It is likely that there are too many or too few blocks in a subcircuit compared to the number of available positions in a functional block. Due to these problems a constraints enforcing post processing step is required which results into a loss of quality because the natural hierarchy of the circuit is disturbed. In Marrakchi *et al.* blocks are swapped between the clusters until all constraints are met. In PPack and PPack2 an architecture without an input bandwidth constraint is used, which is far removed from commercial FPGA devices. But even with this unrealistic architecture, it is required to swap blocks between the clusters to limit the number of blocks in each subcircuit. Next to the difficulty to impose constraints during partitioning, large packing runtimes are expected because many subcircuits have to be partitioned. Each time a subcircuit on a certain hierarchy level is cut in half, it leads to two new subcircuits on the next hierarchical level. In total, the number of required cuts to fully partition a circuit, is approximately equal to the number of blocks in that circuit. In PPack2, a runtime overhead of 10x is reported for the MCNC-20 benchmark circuits when compared to VPR4.3. Luckily partitioning based packing offers an opportunity for multithreaded parallelism. Once a subcircuit is split in half, then the two new independent subcircuits on the next hierarchical level can be

partitioned concurrently, by spawning a seperate thread for each subcircuit.

HDPack [26] solves some of the problems while preserving the reduction in total wirelength. HDPack uses a seed based clustering method but incorporates physical information from a global placement in the affinity metric. This global placement is obtained by partitioning the circuit to a certain hierarchical level. All blocks in the subcircuits on this level have a physical location on the architecture assigned. The physical information leads to a reduction of 20% in total wirelength and a reduction of 6% in critical path delay when compared to T-VPack for the MCNC-20 benchmark designs. However, these reductions come at the cost of additional runtime to partition the circuit.

4.3 Heterogeneous Circuit Partitioning

Digital designs are build-up hierarchically to cope with the increasing vastness and complexity of applications. The application is divided into several subproblems, which are in turn divided into smaller subproblems. Typically the interconnection complexity of these subproblems is high while there are only a small number of connections between them. This hierarchical structure of applications is exploited in our packing tool to improve the properties of the routed design. In the partitioning phase, the circuit is split into a set of subcircuits. A design is recursively bi-partitioned while minimizing the number of connections to be cut. In this way an optimal Rent characteristic [76] is obtained and the natural hierarchy of the circuit is preserved [48]. This method works as follows. First the circuit is split into two parts. The amount of cut edges is minimized and the difference in size between these parts is limited by an unbalance factor, U:

$$\frac{Size_{s1}}{Size_{s2}} < U \tag{4.1}$$

This leads to two subcircuits on the first hierarchy level of the partition tree. The two subscircuits are further bi-partitioned which results in subcircuits on the next hierarchy level. The number of hierarchy levels a circuit has, is approximately equal to $log2(N)$, with N the number of blocks in the circuit. The hMetis tool [67] is used because it is fast and able to generate partitions with a high quality.

4.3.1 Balanced Area Partitioning

Rent's rule states that the relation between the number of terminals, n_t, (cut nets) and the number of internal blocks, n_b, is a power law over

the hierarchical levels of a recursively partitioned circuit. The circuit has to be hierarchically partitioned while minimizing the number of cut nets. Rent's rule emerges because of the natural hierarchy present in the circuit. The natural hierarchy is introduced by the human hardware designers.

$$n_t = t \cdot (n_b)^{rent\_exponent} \tag{4.2}$$

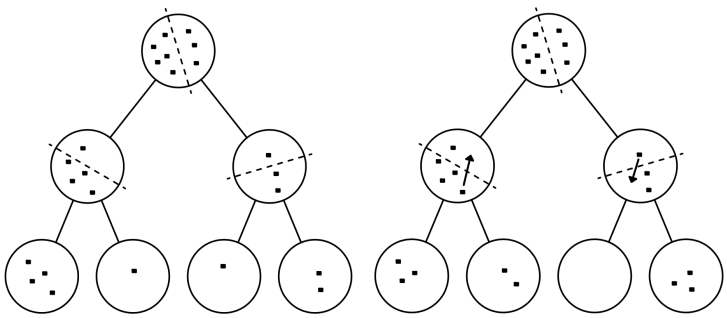
In the experiments section, we use heterogeneous architectures and benchmark applications, so the architectures contain different block types. Different block types have very different sizes in terms of area. RAM blocks are large in comparison to Flip-flops (FF). It is apparent that for such large hard blocks the number of block terminals will be much larger than for a FF, which only has 1 input and 1 output. We advocate that these large blocks should be seen as blocks on a higher hierarchical level. Therefore we use the following extension of Rent's rule: instead of defining the size of a circuit as the number of internal blocks, we use the area of these circuits. The area of a FF and a LUT are based on COFFE simulations [30]. The area of the hard blocks in the VTR architectures can be found in their description. In our experiments we also use a description of the Stratix IV [80] which is made publicly available by Murray et al.[106]. The area of the hard blocks in this architecture is estimated by comparing the area of the hard blocks in the VTR architecture description and the representative area factors given in [135].

4.3.2 Pre-packing

In this step, all netlist primitives that should be packed together are clustered into a molecule. During partitioning a molecule is processed as a single block, with an area equal to the total area of all its atoms. Examples of primitives that should be packed together are DSP primitives and adder primitives that are part of a carry chain.

Carry and Share Chains

Many carry and share chains are part of a long combinational path in the circuit, leading to a large path delay in the routed design. Therefore modern architectures have short and fast dedicated connections built-in for these chains. Blocks that are separated during partitioning are located in different subcircuits and thus all connections between these separated blocks are routed with the slow interconnection network. For this reason connections between blocks that are part of a chain should



(a) Without hard block swap (b) With hard block swap

Figure 4.1: Hard block swap during circuit partitioning

not be cut during partitioning. To avoid that such connections are cut, we generate molecule blocks that contain all atom blocks in a chain.

DSP Primitives

Each DSP element in modern architectures consists of a one or more multipliers connected to an accumulator. It is possible to pack these multiplier and accumulator blocks in different DSP hard blocks, but this leads to an increase in total wirelength and path delay because the dedicated connections in a DSP block are shorter and faster than the interconnection network. To prevent that DSP primitives are scattered and thus part of different subcircuits, we also cluster them into DSP molecules.

4.3.3 Hard Block Balancing

The hard blocks are typically able to implement multiple netlist primitives. All memory slices that share the same address and control signals can be packed together in the same memory block. Similarly a DSP block is able to implement several DSP primitives. If a set of memory slices that would be packed together into a single memory block are scattered over different subcircuits after partitioning, then each subcircuit requires at least one memory block to implement these slices because no hard blocks can be shared between the subcircuits. In the worst case, this leads to a resource requirement with M memory blocks if M memory slices are scattered over different subcircuits.

The problem is solved by adding a hard block swap during recursive bi-partitioning to balance the hard block primitives between both

partitions after each cut. In Figure 4.1 an example is shown for eight memory slices that share the same address and control signals. Consider an architecture in which the memory hard blocks are able to implement three of these slices. The minimum number of required memory blocks to implement these slices is equal to three. If there is no hard block balancing during partitioning, five memory blocks are required. The first subcircuit requires two blocks, while the others require one. If hard block balancing is added, memory slices are swapped between both partitions after partitioning if this reduces the total number of required memory blocks. In this case, only three memory hard blocks are required to pack all memory slices.

The difference with the pre-packing step is that the netlist primitives in the pre-packing step are clustered to a molecule prior to partitioning. We do this because there is little or no choice for the blocks we pack together in the pre-packing step. These blocks have to be packed into the same functional block. During partitioning, we add hard block balancing because there are many candidate primitives that fit into a few hard block instances. Choosing a hard block instance for each of these hard block primitives before partitioning could disturb the natural hierarchy of the circuit because the decision can't be based on the hierarchy that is only obtained after partitioning.

The hard block swapping algorithm is defined as follows. Before partitioning we group all hard block slices that can be implemented by the same hard block type. Let's assume that one of these groups consists of P hard block primitives and that each hard block of this type can implement N slices. In total we thus require $H = \lceil \frac{P}{N} \rceil$ hard blocks of this type. When a circuit is partitioned in two parts C_1 and C_2, both parts have P_{C_1} and P_{C_2} hard block slices, with $P_{C_1} + P_{C_2} = P$. Now a total of $H_1 + H_2$ hard blocks are required, with H_1 equal to $\lceil \frac{P_{C_1}}{N} \rceil$ and H_2 equal to $\lceil \frac{P_{C_2}}{N} \rceil$. If $H_1 + H_2$ is larger than the original number of required hard blocks H then a hard block swap is added after partitioning to rebalance the hard block primitives in such a way that $H_1 + H_2 = H$. In this swap blocks are moved from C_1 to C_2 or from C_2 to C_1. To determine the direction in which hard blocks are moved, we calculate the number of moves M_{12} and M_{21} that are required to ensure that $H = H_1 + H_2$ when blocks are moved in the $C_{1\rightarrow2}$ and $C_{2\rightarrow1}$ direction respectively. The best direction is determined by the minimum of M_{12} and M_{21}. Once the direction is known, we move blocks between the subcircuits until $H = H_1 + H_2$. Hereby the moved blocks are chosen in such a way that the total number of cut edges is minimized. Each time a block is moved, the minimum cut increase is chosen by comparing all possible moves.

Once the number of hard block primitives in the subcircuit is smaller than the number of available positions in a hard block, all these primitives will be packed into a single hard block. In this case, the primitives are clustered to a molecule to prevent that they are further partitioned.

4.4 Timing-driven Recursive Partitioning

4.4.1 Introduction to Static Timing Analysis

Before describing the timing-driven version of the hierarchical partitioning method, we describe how timing properties of a circuit are analyzed. The timing properties are typically analyzed statically without simulating the full circuit. In synchronized systems, the clock signal conducts the synchronized elements such as flip-flops or latches, which copy their input to their output at the rising or falling edge of the clock signal. Logic signals are supposed to move in lockstep, advancing one stage on each tick of the clock signal. During static timing analysis the timing constraints are checked. A typical constraint is a lower limit for the clock frequency. Only two kinds of timing violations are possible:

- Setup time violation: a signal arrives too late at the input pin of a sequential element
- Hold time violation: an input signal of a sequential element changes too soon after the clock's active transition.

To analyze when violations will occur, a timing graph is constructed in which each node is a pin of a primitive block and each edge a connection between these primitives. After technology mapping little is known about the routing pathways of the connections so typically each edge has the same delay. In the packing stage a different delay can be assigned to the edges because connections on a higher hierarchical level will be slower than low level connections. The further downstream in the compilation flow, the more accurate the timing graph will become.

The arrival time of a signal is the time elapsed for a signal to arrive at a certain point, T_{arr}. The reference, or time 0.0, is often taken as the arrival time of the clock signal. Calculating the arrival time requires a forward topological traversal through the timing graph. The required time, T_{req}, is the latest time at which a signal can arrive at a node without timing violations. The timing constraints determine the required times at the sequential inputs. A backward topological traversal is required to calculate the required time at each node. The slack is the difference between the required time and the arrival time.

A positive slack at a node implies that the arrival time at that node may be increased without affecting the maximum delay in the circuit. Conversely, negative slack implies that a path is too slow, and the path must be sped up (or the reference signal delayed) if the timing constraints have to be met. The worst negative slack (WNS) indicates the critical path of the circuit. The timing violations will not improve unless this path is taken care of. To indicate how critical an edge is a criticality measure is introduced:

$$Crit(edge) = \frac{Slack(edge)}{WNS} \tag{4.3}$$

In case the only timing constraint is that the clock frequency should be as high as possible then the critical path is defined as the path with the maximum delay, D_{max}.

4.4.2 Timing Edges in Partitioning

Edges on the critical or on a near critical path should not be cut on high hierarchical levels because this leads to longer and slower connections in the interconnection network. These edges should remain uncut as long as possible. For this reason weighted timing edges are added to the circuit graph before it is passed to the partitioning tool. These timing edges avoid that the critical or a near critical path is cut when a partition is possible without cutting this path. The amount of timing edges added and the weight of these edges are important parameters to obtain good results for both the total wirelength and the critical path delay. Adding too many timing edges with too large weights results in partitions that violate the natural hierarchy of the circuit and leads to an increase in total wirelength.

A timing analysis of the mapped circuit determines where the timing edges should be added. We optimistically assume that the fastest possible connection is used between two blocks. A timing edge is added for each edge in the circuit that has a criticality larger than the threshold value. However there are two types of designs: designs with only a few long paths and circuits with a more gradual path delay distribution.

For the circuits with only a few long paths the process is straightforward. A treshold criticality is set to 0.65. All edges with a higher criticality are added, thereby assuring that they are not cut during partitioning. However, many circuits have a more gradual path delay distribution. Adding every critical connection would lead to too many timing edges in the circuit. For these circuits we avoid adding too many edges by considering only the 20% most critical edges.

We also add a weight to each timing edge, in order to differentiate between the critical and near critical edges. The larger the criticality of an edge, the larger its corresponding weight will be. hMetis only allows integer values for the weights [68], so the criticality of the edge is multiplied with a factor M and rounded to the nearest integer value. We found experimentally that 12 is an optimal value for M.

Timing Edge Weight Update

If a critical or near critical path is cut during recursive bi-partitioning, then this path should have special attention. Otherwise it could be cut several times and this would lead to multiple slow connections in the interconnection network. To prevent this, all timing edges on a critical or near critical path have a very large weight assigned once it is cut. This ensures that other uncut critical connections are cut first if a partition is not possible without cutting critical edges.

4.5 PARTSA

Our first packing algorithm we propose is PARTSA. It is a fully partitioning based packer with a simulated annealing based refinement step instead of the constraints enforcing post processing step that is common for the partitioning based packers we described in Section 4.2. In the first phase the circuit is completely partitioned to small sets of blocks. In existing packing tools, partitioning ends when the size of all sets is smaller than the available number of positions in a functional block. In this approach, an additional constraints enforcing step is required for two reasons. Firstly it is not possible to limit the number of pins on each subcircuit. Secondly it is very difficult to obtain subcircuits that exactly fill all empty positions in a functional block.

To remedy this PARTSA uses the distance from the resulting partition tree as a cost metric in a simulated annealing process. Two advantages are obtained with this method. Firstly it is possible to impose a bandwidth constraint on each functional block by adding a penalty, relative to the number of used input pins. Secondly the problem with too large or too small subcircuits is solved because the simulated annealing method fills all functional blocks until no more empty positions are left.

4.5.1 Introduction to Simulated annealing

Simulated annealing [70] is a heuristic to find the global minimum of a cost function inspired by the physical annealing of metals. Physical an-

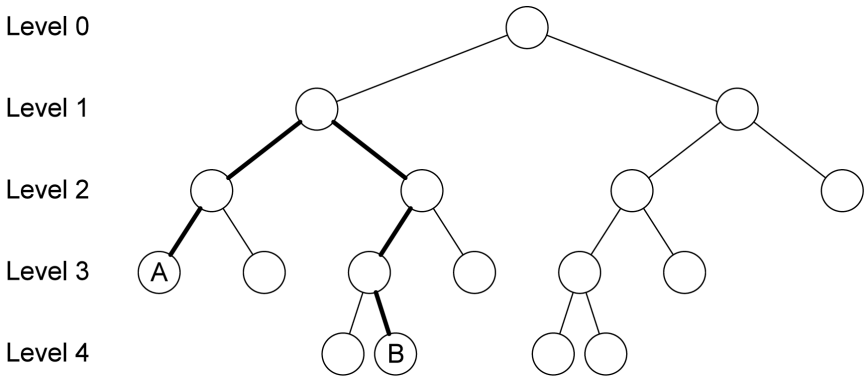


Figure 4.2: Partition tree and distance metric between two blocks

nealing of metals involves heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. To apply simulated annealing, each state in the search space should have an easily calculable cost.

Simulated annealing is an iterative algorithm in which a solution is repeatedly randomly altered and evaluated. If a proposed alteration causes the cost of the solution to drop, the newly altered solution is accepted and used for further alterations in the next iterations. To avoid getting stuck in local minima, sometimes an alteration causing a higher cost is accepted as well. The probability of such a solution to get accepted depends on how much worse the solution is and the value of the temperature (T) at the current iteration. The alteration is accepted with a probability of $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost due to the alteration. T, the temperature, controls the probability by which hill-climbing moves/swaps are accepted. Initially, T is very high so that most moves are accepted, which allows a thorough exploration of the solution space. Gradually T is decreased so that the probability by which hill-climbing moves are accepted decreases and the placement converges towards a close to optimal solution. The algorithm stops when the solution does not improve anymore.

Initialization During initialization the primitives are packed in high level functional blocks without optimization. The blocks are filled with the sets from the hierarchical partitioning. All available positions in the high level functional blocks are filled, in that way we avoid the problem of having too large or too small subcircuits after partitioning.

Alterations An alteration is randomly constructed by choosing a random primitive and a random primitive block location in one of the high level functional blocks and different from the location of first randomly chosen primitive. If there is a primitive assigned to the chosen location, then the primitives are considered to be swapped. In the other case the chosen primitive is considered to be moved to the vacant primitive location. Following an annealing schedule, alterations are performed until the total cost is minimized.

Annealing Schedule The ensemble of the following parameters is called the annealing schedule: the initial temperature, the rate at which the temperature is decreased, the number of moves that are attempted at each temperature, the way in which potential moves are selected and the exit criterion. A good annealing schedule is crucial for finding a good solution in a reasonable amount of time. We used the annealing schedule proposed in [16]. The annealing schedule is formulated in Equation (4.4) with an exponential cooling using a variable parameter γ. The goal of the variable parameter $\gamma(\beta)$ is to make the algorithm spend more time in the stage where the algorithm makes the most improvement, namely when the fraction of accepted alterations β is between 15% and 96%.

$$T_{new} = \gamma(\beta) \cdot T_{old} \tag{4.4}$$

$$\gamma(\beta) = \begin{cases} 0.5, & \text{if } 96\% < \beta \\ 0.9, & \text{if } 80\% < \beta \leq 96\% \\ 0.95, & \text{if } 15\% < \beta \leq 80\% \\ 0.8, & \text{if } \beta \leq 15\% \end{cases} \tag{4.5}$$

The initial temperature can be estimated by performing some random alterations and calculating the changes in the cost function. The standard deviation of the delta costs determines the initial temperature. Both the quality and runtime are not very sensitive to the initial temperature, because the annealing schedule adapts the temperature exponentially, if the number of accepted alterations is too high.

A detailed description of the annealing schedule falls out of the scope of this thesis. We refer the reader to [16] for more details.

4.5.2 Cost Function

The cost function used in simulated annealing is based on the partition tree distance. Additionally the annealer also imposes an input band-

width constraint to each functional block by assigning a cost, relative to the number of input pins. In this way a post-processing constraints enforcing step can be avoided.

The cost of each functional block (FB) is a weighted sum of two elements, a distance cost and a pin cost:

$$Cost_{FB} = \alpha \cdot Cost_{Distance} + (1 - \alpha) \cdot Cost_{Pin} \tag{4.6}$$

The distance cost is the sum of the distance between all pairs of blocks in the considered functional block Equation (4.7). The distance between two blocks is defined as the distance of the path between these blocks in the partition tree. In Figure 4.2, the path between block A and B is shown and has a distance equal to five. This way, blocks that are partitioned on a low hierarchy level are preferably put into the same functional block.

$$Cost_{Distance} = \sum_{b_1, b_2 \in FB} distance_{tree}(b_1, b_2) \tag{4.7}$$

If a functional block has more input pins, P_{FB}, than the threshold value, P_{TH}, then a penalty cost is added (4.8). For a P_{TH} value of 35 all functional blocks typically satisfy an input bandwidth constraint of 40.

$$Cost_{Pin} = \begin{cases} 0 & P_{FB} \leq P_{TH} \\ (P_{FB} - P_{TH})^2 & P_{FB} > P_{TH} \end{cases} \tag{4.8}$$

We empirically devised the optimal weight α in Equation (4.6) to be 0.16667.

4.5.3 Fast Partitioning

In PARTSA the circuit is completely partitioned because the information from the resulting partition tree is used in the simulated annealing step. This results in large runtimes because many subcircuits have to be cut in half. A large gain is obtained by using multithreading and avoiding partitioning on the lowest hierarchy levels.

In Figure 4.3 the cumulative partitioning runtime over all hierarchical levels is shown for *LU64PEEng*. On the high hierarchical levels only a few large subcircuits have to be partitioned. To speed up this process, multithreading is used. This is possible because once a circuit is cut in half, both subcircuits can be partitioned independently.

Firstly the circuit is split into two subcircuits. Then these are partitioned in parallel by using two threads. On the next hierarchical level four subcircuits are available which are partitioned in four threads. The

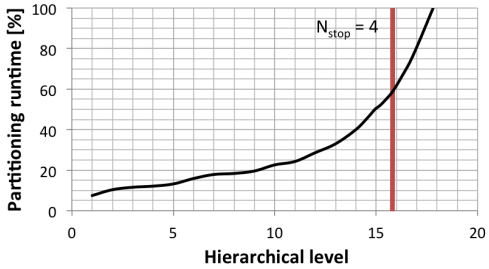


Figure 4.3: Cumulative partitioning runtime over all hierarchical levels for *LU64PEEng*

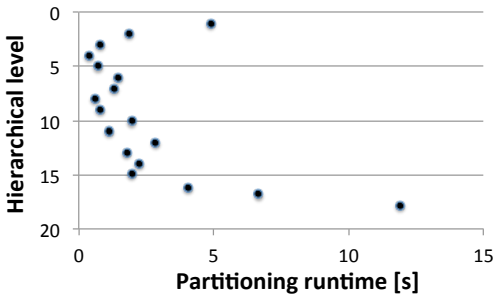
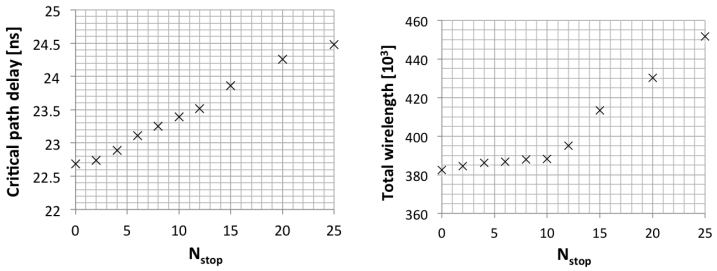


Figure 4.4: Partitioning runtime per level for a multi-threaded partitioning on a CPU with four cores. For the first four levels, the runtime decreases because there are unused threads because of the low number of subcircuits.

maximum number of threads is parameterized and is best adapted to the number of cores on the used CPU.

A problem occurs on the lower hierarchical levels. On these levels many small subcircuits have to be partitioned which leads to a large amount of overhead. For *LU64PEEng* more than 100000 subcircuits are partitioned on the lowest two hierarchy levels. This takes around 40% of the total partitioning time as shown in Figure 4.3. Even if we apply multi-threading to the partitioning problem, the runtime increase remains because of the sheer amount of subcircuits that have to be partitioned, see Figure 4.4. The solution is to stop partitioning when the size of a subcircuit is smaller than a certain threshold value N_{stop}. In this case, a subcircuit of size N with $N \leq N_{stop}$ is not further cut with recursive bi-partitioning but instead it is immediately split into N subcircuits that contain only one block. This means that recursive bi-partitioning ends when the size of a subcircuit is smaller than the threshold value resulting in less distance information in the partition tree.



(a) Critical path delay (b) Total wirelength

Figure 4.5: Clustering quality in function of N_{stop} for *bgm*

Reducing the partitioning depth leads to a disturbance in the natural hierarchy. There is no detailed distance information available beyond the threshold hierarchy level, but it is still possible to obtain high quality clustering results with the simulated annealing method. In Figure 4.5 the critical path delay and total wirelength for *bgm* are shown in function of N_{stop}. We observe that this method does not lead to a large loss in quality when N_{stop} is small. The total wirelength starts to increase rapidly when N_{stop} is larger than the number of available positions in a functional block. In our experiments we use an $N_{stop} = 4$. This leads to a loss in quality of less than 1% while reducing the partitioning runtime with 40%.

4.5.4 Parallel Annealing

To speed up the process and quality of simulated annealing not all blocks are packed with one simulated annealing step. Instead anneal roots are chosen for independent multithreaded annealing. These anneal roots are subcircuits on a high hierarchical level in the partition tree.

In Figure 4.6 a partition tree of a small circuit is shown. If the toy circuit is packed into functional blocks with two available positions, then this will result in a solution with four clusters as shown in the figure. These blocks are the result of the annealing based packer because in this case the total cost is minimized. If subcircuits S1 and S2 would have been used as independent subcircuits for simulated annealing, the same result would have been obtained because blocks from S1 and S2 are not packed into the same functional block. Doing this would increase the total cost because the tree distance between blocks in S1 and S2 is large. More in general, if two subcircuits on a high hierarchical level in the partition tree are considered, then the blocks in these sub-

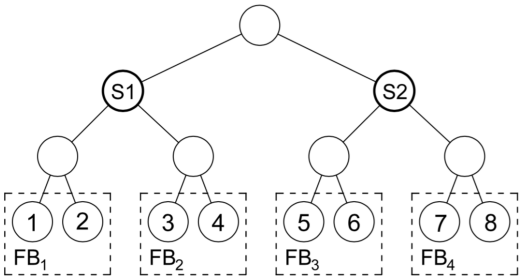


Figure 4.6: Anneal roots for fast independent multithreaded annealing

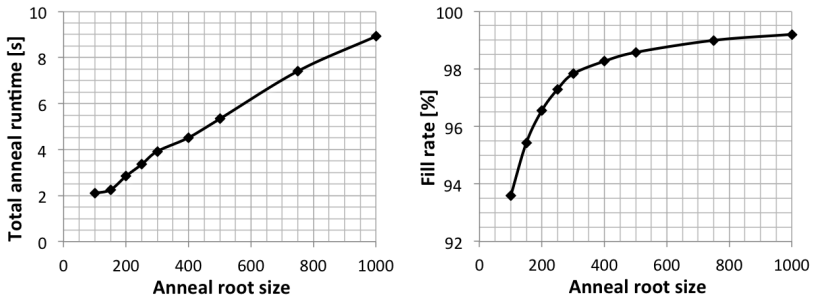
circuits will not be packed into the same functional block because this would increase the total cost. Therefore it is possible to pack the blocks from these subcircuits independently with the annealing based packer.

This method reduces the annealing runtime for two reasons. Firstly the size of the annealing problem is largely reduced because less swaps are possible in the smaller search domain. Secondly multithreading can be used because all subcircuits are annealed independently.

The drawback is that lower filling rates for the functional blocks are obtained because each anneal root possibly leads to unfilled positions. If the number of available positions in the example of Figure 4.6 would increase to three, then three clusters are required to pack the eight blocks from the circuit and one position remains unfilled. For the approach with anneal roots each subcircuit has four blocks and this leads to functional block with unfilled positions for each anneal root. For each of these anneal roots two functional blocks are required. The two functional blocks would provide six positions from which two are unused.

In Figure 4.7 the total anneal runtime and fill rate are shown in function of the anneal root size. The anneal root size is the number of primitives that is annealed in one simulated annealing process. The root size should be as small as possible to reduce the annealing runtime, but if the root size becomes too small than there is a penalty in terms of area overhead. For each simulated annealing root, there will be high level functional blocks that are not completely filled, so the filling rate will be smaller for smaller root sizes.

We use an anneal root size of 300, this leads to filling rates of 98% and up to 10x faster anneal runtimes when compared to the case where no anneal roots are used.



(a) Total anneal runtime (b) Fill rate of the functional blocks

Figure 4.7: Total anneal runtime and fill rate of the functional blocks in function of the anneal root size for *LU32PEEng*

4.5.5 Problems with PARTSA

A problem occurs when complex architectures with sparse crossbars are used. Each time two blocks are swapped, a detailed routing would be required to check if a legal solution is obtained because not all connections can be realized in the local interconnection network. This results in large runtimes because the detailed routing check would have to be executed in the kernel of the simulated annealing algorithm that is swapping blocks. Many swaps have to be performed in a simulated annealing algorithm to obtain a result with a low wiring cost and it takes too much time to perform a detailed routing every time.

4.6 MULTIPART

To solve the problems with sparse crossbars, we propose to combine the advantages of partitioning based and seed based packing in one algorithm that we call MULTIPART. It consists of two main parts (Figure 4.8): a partitioning step to partition the circuit hierarchically to a set of subcircuits and a seed based packing step that packs all these subcircuits concurrently. There are a number of advantages to this approach:

- The natural hierarchy of the circuit is retained in the partitioning step, so a large reduction in total wirelength is achieved.

- No partitioning is required on the deep hierarchical levels, which saves runtime.

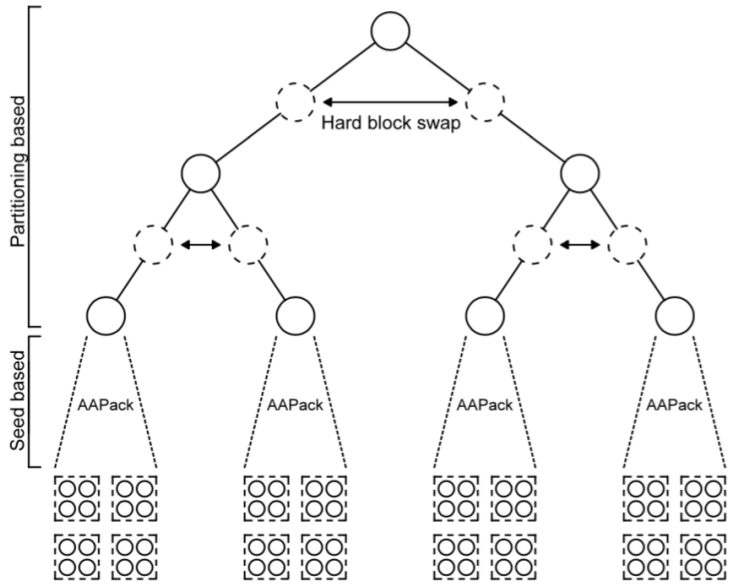


Figure 4.8: Overview of the MULTIPART algorithm. The tool consists of two steps, a partitioning step to split the circuit into N independent subcircuits and a seed based packing step to pack these independent subcircuits concurrently.

- Heterogeneous designs can be packed on architectures with sparse crossbars, because detailed routing is taking care of by the seed-based packer backend.

The subcircuits that result from the partitioning step are packed with a seed based packer. To obtain a maximum gain in total runtime, all these subcircuits are packed independently on a virtual target with its own resources. Afterwards all resulting functional blocks are combined into a single netlist and placed onto the target device.

The seed based packer introduces a large runtime overhead to initialize all modes of the modern and complex architectures. To allow fast packing on complex architectures, we automatically generate a design specific architecture in which only the required modes are described. Otherwise all modes are initialized even if these modes are never used by the design. This method effectively reduces the runtime overhead of the seed based packer.

4.6.1 Optimal Number of Subcircuits

To determine the optimal number of subcircuits for multithreaded seed based packing, there are two conflicting motivations. Firstly the num-

ber of subcircuits should be minimized because partitioning results in an increase in runtime. There should also be enough subcircuits in order to fully exploit the available number of threads available on the target CPU. This way all threads are used to pack the subcircuits concurrently during the seed-based packing stage. Secondly packing a circuit with a partitioning based packer leads to better results for the total wirelength. The number of subcircuits should be maximized in order to partition the circuit as much as possible, thereby reducing the total wirelength.

A maximum cluster size (N_{max}) is defined for all subcircuits. If the size of a subcircuit is smaller than this value, then it is further clustered with the seed based packer. Because all circuits have a different size N, choosing an N_{max} value that gives good results for all circuits is difficult. Therefore an N_{sub} is introduced, which is the minimum number of subcircuits resulting from the hierarchical partitioning. By making use of this factor, N_{max} is calculated as follows: $N_{max} = N/N_{sub}$. We found that a N_{sub} of 30 leads to a good result for both small and large designs.

4.6.2 Passing Timing Information via Constraint Files

For some circuits it is impossible to avoid that critical edges are cut, as explained in Section 4.4.2. This problem is more pronounced for MULTIPART, because a cut critical path results in smaller paths in both subcircuits that are treated independently by the seed-based packing phase. In Figure 4.9 the critical path P2 is cut during partitioning, leading to two subcircuits where the cut critical edge is replaced with an input/output pin. If these subcircuits are further processed, the path P2 will not be considered as critical by the seed-based packer because now it is shorter than in the original circuit. This will lead to an increase in critical path delay.

To solve this problem, additional timing information is passed on to the seed-based packer for each subcircuit. We use the Synopsys Design Constraints (SDC) format in our implementation [103]. In this format, it is possible to assign a delay to any input and output pin of the circuit. In MULTIPART these delays are added to all input/output pins that are the result of cutting a critical edge. This way, the seed based packer knows the total delay of all critical paths in the circuit and is able to minimize the maximum delay.

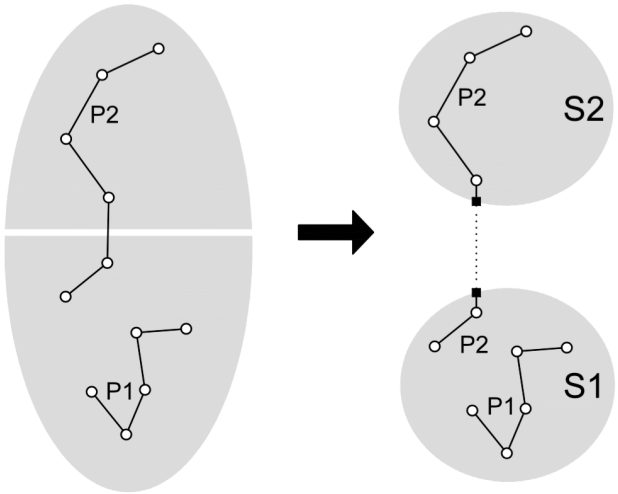


Figure 4.9: Partitioning of a circuit with cut and uncut critical path

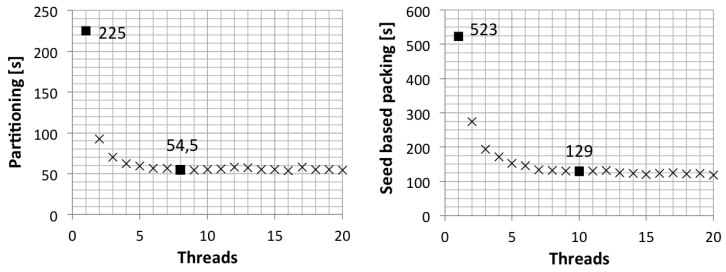
4.7 Experiments

Post-routing total wirelength and critical path delay are obtained by placing and routing the packed circuits with VPR 7.07. For PARTSA an architecture with complete crossbars is used in a 40 nm technology (*k6_N10_40nm*). The results of MULTIPART are also analysed on this architecture to compare both packers. A second more realistic architecture with sparse crossbars in a 22nm technology (*k6_N10_gate_boost_0.2V_22nm*) is used to further analyse the MULTIPART packer. Finally we also use a commercial architecture, Altera's Stratix IV to evaluate MUTLIPART for the heterogenous Titan benchmark suite.

The performance of the routed designs produced by VPR can vary. To account for this variance all experiments are run eight times, each time with a different seed for placement. The results shown below are averaged out over these eight experiments.

4.7.1 Optimal Number of Threads

First, we experimentally determine the optimal number of available threads for the multithreaded algorithms. In all our experiments a workstation with an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with four cores is used. For the PARTSA packer, the partitioning runtime is shown in function of the number of threads in Figure 4.10(a). When the number of threads is increased from one till eight, then we maximally



(a) PartSA: partitioning based packing (*k6_N10_40nm*)

(b) MultiPart: seed based packing (*k6_N10_gate_boost_0.2V_22nm*)

Figure 4.10: Runtime versus available threads for *LU64PEEng*

obtain a runtime speed up of 4x. This result is expected because the target CPU has four available cores. For the MULTIPART packer, seed based packing runtime is shown in function of the number of threads in Figure 4.10(b). Here the same result is obtained when ten threads are used. Again multithreading results in 4x faster runtimes. In case the absolute values for the runtimes are compared between Figure 4.10(a) and Figure 4.10(b), then we should take into account that a different target architecture is used. The *k6_N10_gate_boost_0.2V_22nm* architecture has depopulated local interconnect crossbars, which makes it harder to pack, because each time a block is packed, the packer has to check if the local routing is feasible.

4.7.2 An Architecture with Complete Crossbars

In Figure 4.11 the reduction in total wirelength is shown for the 10 largest VTR benchmark circuits when compared to AAPack. PARTSA leads to the largest reduction in total wirelength for all circuits with a geomean equal to 26%. For MULTIPART smaller gains are obtained, especially for circuits where it is difficult to control the critical path delay. For these circuits, it is hard to find partitions without cutting a critical path. Each time a critical path is cut, all weights on this path are increased, leading to worse cuts on the next hierarchical level because other, less optimal, edges are cut. The geomean of the reduction for this packer is equal to 12% (Table 4.3).

The large reduction in total wirelength has the benefit that the minimum required channel width to route all connections is smaller. Therefore it is possible to route these circuits on FPGAs with a smaller channel width when these packers are used. This is important because less

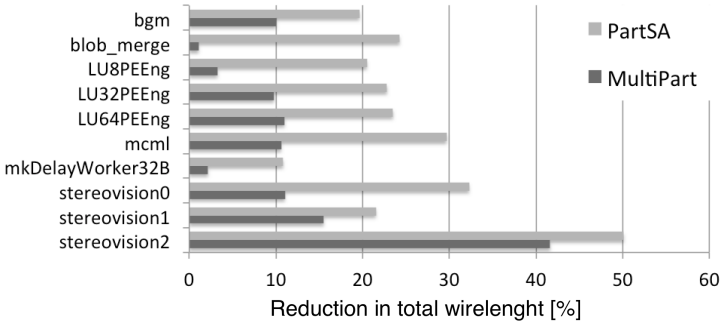


Figure 4.11: Reduction in total wirelength for PARTSA and MULTIPART when compared to AAPack with the *k6_N10_40nm* architecture

routing resources are required, and smaller and cheaper FPGAs can be used to implement the same design. Due to the computational complexity we omitted *LU64PEEng*. The gain in minimum channel width of the remaining circuits is shown in Figure 4.12 and has a geomean equal to 28% when PARTSA is used and equal to 15% for MultiPart.

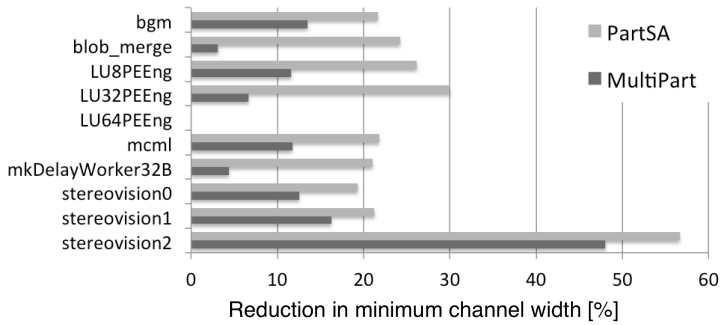


Figure 4.12: Reduction in minimum channel width for PARTSA and MULTIPART when compared to AAPack with the *k6_N10_40nm* architecture

The total runtime is shown relative to the area in Figure 4.13. On average a gain of 1.8x is obtained for PARTSA while MULTIPART has a gain of 2.7x (Table 4.3). The results for the largest circuits are most important because these require largest packing runtimes. With AAPack 162 seconds are required to pack *LU64PEEng*. This is reduced to 69 seconds for the PARTSA packer and to 40 seconds for the MULTIPART packer, which is 4x faster than AAPack for a CPU with four cores.

We notice that larger runtime gains are mainly obtained for the larger circuits. In Table 4.1 the gain for PARTSA and MULTIPART is shown for the LU circuits. These are three scalable linear system solvers

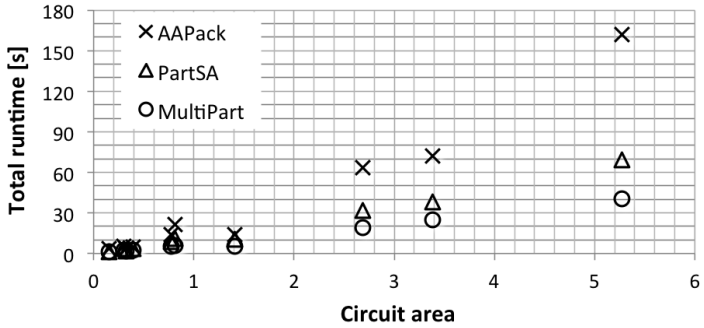


Figure 4.13: Runtime of AAPack, PARTSA and MULTIPART with the *k6_N10_40nm* architecture

with a different size. The larger the size of the circuit, the larger the obtained gain in total runtime.

Table 4.1: Gain in total runtime for LU circuits

Name	Area	PARTSA	MultiPart
LU8PEEng	770K	1.7x	2.6x
LU32PEEng	2.7M	2x	3.3x
LU64PEEng	5.3M	2.3x	4x

The ratio of the critical path delay is shown relative to AAPack in Table 4.2. For both packers, the geomean of the ratio is smaller than one, which means that they do not lead to an increase in critical path delay. Although for some circuits an increase is obtained, especially for the PARTSA packer. In PARTSA it is difficult to control the cut critical paths through all hierarchical levels, especially when the circuit has many critical edges. Therefore, MULTIPART leads to better results, here less hierarchy levels are partitioned and the seed based packing phase also uses information of the cut critical edges during seed based packing by making use of the SDC files.

4.7.3 An Architecture with Sparse Crossbars

MULTIPART is able to pack circuits on an architecture with sparse crossbars (*k6_N10_gate_boost_0.2V_22nm*). With this architecture, it is harder to pack all blocks because detailed routing fails many times. This leads to longer packing runtimes and an increase in total wirelength because less connections are implemented inside the functional blocks. When results from AAPack are compared on both architectures, an increase of 38% in total wirelength is observed. With MULTIPART better clustering

Table 4.2: Ratio of the critical path delay relative to AAPack with the *k6_N10_40nm* architecture as a target device

Design name	PARTSA	MultiPart
bgm	0.92	0.97
blob_merge	1.07	1.00
LU8PEEng	0.96	0.92
LU32PEEng	0.99	1.01
LU64PEEng	0.99	1.01
mcml	1.11	1.01
mkDelayWorker32B	1.07	1.02
stereovision0	1.04	1.02
stereovision1	0.92	0.91
stereovision2	0.82	0.88
geomean	0.985	0.974
stdev	0.088	0.052

results are obtained, resulting in a smaller increase in total wirelength. A gain in total wirelength of 20% is achieved when compared to AAPack on this architecture. We thus observe that better results are obtained with MULTIPART if a more complex architecture is used. For the gain in critical path delay and the gain in total runtime similar results as with the other architecture are obtained. Runtime speed-ups up to 4x are achieved for the largest circuits with a geomean of 2.9x and the gain in critical path is equal to 3.7% (Table 4.3).

Table 4.3: Summary of the properties for PARTSA and MULTIPART on the three target architectures when compared with AAPack on a CPU core with four cores

	Total wirelength	Critical path delay	Runtime speedup
k6_N10_40nm - VTR benchmark			
PARTSA	-26%	-1.5%	1.8x
MULTIPART	-12%	-2.6%	2.7x
k6_N10_gate_boost_0.2V_22nm - VTR benchmark			
MULTIPART	-20%	-3.7%	2.9x
Stratix IV - Titan benchmark			
MULTIPART	-28%	-8%	3.6x

4.7.4 A Commercial Architecture

MULTIPART is able to handle heterogeneity, so we tested it on a com-

Table 4.4: Post-route results for MULTIPART in comparison with AAPack, for the Titan benchmark designs and Stratix IV as a target device

	Size	Runtime			TWL	CPD
Design name	x1000	Packing	Placement	Routing		
neuron	91	0.31	0.95	0.79	0.66	1.01
sparcT1_core	91	0.34	0.96	0.31	0.72	0.98
stereo_vision	92	0.33	0.95	0.82	0.73	0.95
cholesky_mc	108	0.32	0.94	0.88	0.81	1.00
des90	110	0.27	0.92	0.34	0.72	0.80
segmentation	168	0.20	1.07	0.11	0.85	0.96
bitonic_mesh	192	0.26	0.86	0.05	0.66	0.81
stap_qrd	237	0.22	0.96	0.64	0.60	0.89
cholesky_bdti	256	0.29	0.91	0.88	0.74	0.93
Geomean ratio		0.28	0.95	0.39	0.72	0.92
Std. Dev.		0.05	0.06	0.34	0.08	0.08

TWL = Total Wirelength
CPD = Critical Path Delay

mercial architecture, the Stratix IV, and used the larger designs in the Titan suite to benchmark the packer in comparison with AAPack. The results are listed in Table 4.4. The resulting gains are better for the larger designs with a 28% reduction in total wirelength and an 8% decrease in critical path delay while speeding packing with a factor of 3.6x.

An additional advantage of the better packing quality is the reduction in routing runtime and this becomes apparent for the larger designs in the Titan suite. For the selection of benchmark designs in Table 4.4, the routing runtime is reduced with a factor of 2.56x, albeit not consistent with a high standard deviation of 0.34.

4.8 Conclusion and Future Work

A partitioning based packing method leads to a better quality in terms of total wirelength but is slower when a single core approach is used. In contrast to seed based packing, partitioning a circuit offers the oppurtunity to multithread packing. Once a circuit is split into two parts, then both parts can be further processed concurrently by two separate threads. Furthermore, a large reduction in minimum channel width is obtained, which reduces the required number of metal layers and generally allows designers to move to a smaller and cheaper target FPGA if the routing congestion is a bottleneck.

Although obtaining good results, a fully partitioning based methodology is not suitable for architectures with sparse crossbars. This is solved by combining the advantages of partitioning based and

seed based packing. The end result is 3.6 x shorter packing runtimes and 2.56 shorter routing runtimes on average, while increasing the quality of the routed design with 8% shorter critical path delays and 28% less wiring. The improved packing performance substantially reduces the gap between the academic and commercial results from the previous chapter.

In Future work we want to use the intermediate results produced by the partitioning based packing algorithms during placement. The partitioning based packers that we described in this chapter build a partitioning tree in the first phase of their algorithm. The partitioning tree can be used to guide the placement process by producing an initial placement based on the partitioning tree for example.

5 Steepest Gradient Descent Based Placement

In this chapter, we introduce a new placement algorithm and compare it to conventional approaches. In the introduction we start with describing why we should develop fast placement approaches and subsequently we give an overview of the conventional techniques. We discuss the problems of conventional techniques and the oppurtunities we exploited in our new placement tool.

5.1 Introduction

Placing a design on an FPGA encompasses assigning each block in the mapped input circuit onto a physical block on the FPGA while ensuring that no two blocks occupy the same physical location. Placement is a runtime intensive task and easily takes one third of the total runtime of the FPGA CAD tool flow, as shown in Figure 3.3 and described in Chapters 1 and 3. Since the FPGA CAD runtime is a bottleneck in the FPGA design cycle, we want to reduce the placement runtime as much as possible. Unfortunately the placement problem is NP hard.

Several heuristic approaches have been proposed to solve the placement problem. The oldest is the simulated annealing approach to FPGA placement. It produces results with a very high quality, but at the cost of a high runtime. Circuit sizes have grown over the years, and placing a large circuit with the VPR simulated annealing based placer

can now take up to multiple hours. A lot of research has gone into speeding up simulated annealing. One approach is to parallelize the simulated annealing algorithm. This approach has yielded speedups of $2.2\times$ on four cores with no quality loss [86] and $10\times$ using GPGPU with some quality losses [33]. The latter result however was achieved on an unrealistic homogeneous devices with smaller logic clusters. Highly heterogeneous devices with larger clusters were introduced in the early 2000's.

Another class of placement algorithms that shows promising results, is analytical placement. Analytical placers were first invented in the 1980s in the field of ASIC design [71] and continue to be widely used in that field. For FPGAs, Xilinx first started using analytical techniques in 2007 [54]. Academic research on analytical placers is still lagging behind proprietary tools in terms of QoR. Multiple analytical placers have been proposed in academic literature that are competitive with the VPR placer, most notably [53, 152, 83, 151]. None of these placers have publicly available source code and a lot of details are omitted in the publications, which makes it hard to reproduce the results. The results are also reported for unrealistic homogeneous FPGA architectures with smaller logic clusters. We had to develop our own timing-driven analytical place, which is described in Section 5.4. Our implementation is largely based on the wirelength-driven placer HeAP [53], which is in turn based on an analytical placer for ASICs called SimPL [69].

In this chapter we describe a new way to calculate FPGA placements and compare it to analytical placement and simulated annealing. Analytical placement places a design by minimizing a cost function. Unfortunately it is not possible to put all the constraints in one analytical solvable cost function. So the problem is divided in multiple iterations. In each iteration the cost function is adapted to the result of the previous iterations and minimized again. The minimization encompasses deriving the cost function and finding the minimum by solving a linear system. Experiments show that it is not necessary to have the high accuracy of the analytical solution. In LIQUID we move all the blocks in the placement in several small steps, instead of minimizing the cost function. The steps are taken in the direction that reduces the cost function the most. This is called the steepest gradient descent (SGD). After each step is taken the cost function is updated. This SGD optimization is much faster than solving a linear system.

Another big advantage of using SGD optimization is that there is more freedom in choosing the cost functions. More complex cost functions are possible as long as they are derivable. SGD optimization is popular in machine learning. It is successfully being used in training

deep convolutional networks [66], which are complex non-linear functions. In analytical placement the form of the cost function is restricted. Minimization of the cost function should lead to system of equations that are solvable in a reasonable time frame. This forces the cost function to be a sum of squares. After derivation, a sum of squares can be represented by a matrix, which can be inverted to find the minimum.

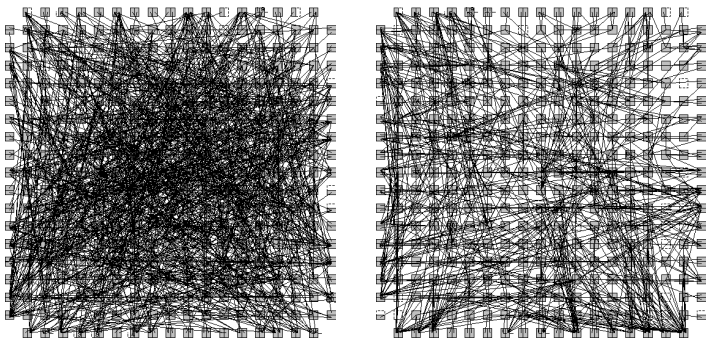
Calculating the SGD move vectors is computationally less intensive than solving the linear system, which makes LIQUID much faster than classic analytical placement. Additionally the movement vectors can be calculated independently, which makes it uniquely suitable for parallelisation. The total workload is very fine grained and liquid, hence the name of the algorithm. It can be divided in arbitrary partitions over multiple CPU or GPU cores. We describe a single threaded implementation as a proof-of-concept. Experimental results are already showing superior performance at lower runtime budgets. LIQUID performs better in the runtime-quality tradeoff at lower runtimes in comparison with simulated annealing and analytical placement.

The work in this chapter was done in collaboration with Seppe Lenders. Seppe Lenders was a master thesis student in our research group in 2015-2016. The work is reported in a conference publication [49].

5.2 FPGA Placement

An FPGA placement algorithm takes two inputs: the mapped input circuit and a description of the target FPGA architecture. A placement algorithm searches a legal placement for the functional blocks of the input circuit so that circuit wiring and timing are optimised. In a legal placement every functional block is assigned to one of the physical blocks that is capable of implementing the functional block and no physical block can be assigned more than one functional block. This type of optimization problem is called an assignment problem.

The main optimisation goal used by placement tools is to minimize the total wire length required to route the wires in the given placement. Placers that are only based on this goal are called wire-length-driven placers. More complex tools such as routability-driven [124] and timing-driven placers [97] trade some of the wire-length for a more balanced wiring density across the FPGA or a higher maximum clock frequency of the circuit, respectively. The total wirelength leads to more convex optimization functions and is generally easier to optimize, it has an impact on power consumption, routability and critical path delay. However optimizing for maximum clock frequency directly is nec-



(a) Initial Random Placement. Cost: 74.5526.

(b) Final Placement. Cost: 28.5384.

Figure 5.1: Visualization of the placement for the MCNC benchmark circuit e64.

essary to obtain reasonable timing characteristics. Our new placement tool, LIQUID can be run in two modi, wirelength-driven and timing-driven.

Finding a placement with a decent quality is important. Poor quality placements generally cannot be routed or lead to low operation frequencies and high power consumption. In Figure 5.1 the initial random placement and an optimized placement are visualized for the *e64* benchmark design from the MCNC suite. We choose this small benchmark design because it is still feasible to visualize its placement. For this design a random placement leads to a critical path delay of 73ns and a minimal channel width of 22 if we want to implement it on the *4lut_sanitized* architecture. In case the placement is optimized with the simulated annealing placement tool in VPR 4.30, the post-routing critical path delay decreases to 40ns and the minimum channel width drops to 9. This quality difference increases enormously for larger circuits and larger target devices. So it is important to find a good placement to reduce the wiring on an FPGA which will reduce the silicon area and the number of metal layers needed to fabricate an FPGA. For FPGA designers the wiring of the design can be a bottleneck that forces them to buy a larger and therefor more expensive chip.

Additionally, the placement problem is computationally hard, so there are no known algorithms that can find an optimal solution in a reasonable time. Therefore, many heuristics have been developed for the placement problem. Most of these algorithms belong to one of three

types of placers: partition-based placers [94], analytical placers [22] (Section 5.4) and simulated annealing placers [16] (Section 5.3). The two main commercial FPGA vendors use different techniques. Analytical placement is the main technique used in Xilinx Vivado's placement tool [85] and Altera's Quartus placer is mainly based on simulated annealing [46]. In the past Xilinx also used simulated annealing in its now deprecated Integrated Synthesis Environment (ISE) Design Suite.

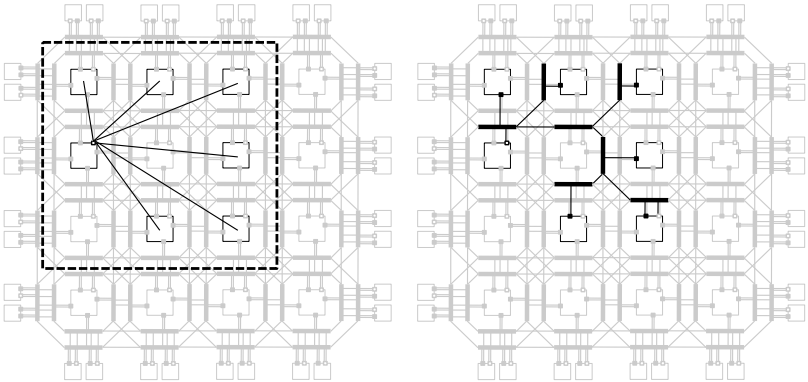


Figure 5.2: The bounding box (left) and a routing solution (right) for a net placement. The Half Perimeter Wire Length (HPWL) for this net placement is 6.

5.2.1 Wire-length Estimation

The only way to exactly calculate the total wire-length of a given placement is to route the wires in a placement and sum the wire-length over all nets. Since routing is in itself a computationally hard problem, solving it repeatedly in the inner loop of the placer and in this way exactly calculating the total wire-length, leads to unacceptably long execution times for the placer. Therefore, the cost is not exactly calculated but estimated. A common way of estimating is calculating the sum of the estimated wire lengths of each net, where the Wire Length of a net is estimated as the Half-Perimeter of its bounding box (HPWL) weighted by a factor $q(n)$ which depends on the number of terminals of the net (taken from [29], see Table 5.1). It is equal to 1 for nets with up to three terminals because the HPWL is equal to the wirelength of the minimum Steiner tree routing solution for these nets. It slowly grows to 2.79 for nets with 50 terminals. An illustration of the bounding box is depicted

Table 5.1: The net weight factors to account for the number of net terminals while estimating post-routing wirelength.

n=#sinks	**q(n)**
$1 \leq n \leq 3$	1.000
4	1.0828
5	1.1536
6	1.2206
7	1.2823
8	1.3385
9	1.3991
10	1.4493
$10 < n \leq 15$	$(n-10) \cdot (1.6899 - 1.4493)/5 + 1.4493$
$15 < n \leq 20$	$(n-15) \cdot (1.8924 - 1.6899)/5 + 1.6899$
$20 < n \leq 25$	$(n-20) \cdot (2.0743 - 1.8924)/5 + 1.8924$
$25 < n \leq 30$	$(n-25) \cdot (2.2334 - 2.0743)/5 + 2.0743$
$30 < n \leq 35$	$(n-30) \cdot (2.3895 - 2.2334)/5 + 2.2334$
$35 < n \leq 40$	$(n-35) \cdot (2.5356 - 2.3895)/5 + 2.3895$
$40 < n \leq 45$	$(n-40) \cdot (2.6625 - 2.5356)/5 + 2.5356$
$45 < n \leq 50$	$(n-45) \cdot (2.7933 - 2.6625)/5 + 2.6625$
$n > 50$	$(n-50) \cdot 0.02616 + 2.7933$

in Figure 5.2. The cost of a placement is then calculated as follows:

$$C_{wl} = \sum_{n \in nets} q(n) \cdot HPWL(n) \tag{5.1}$$

As an example the wirelength estimation is applied to the simple example in Figure 5.2. With an $HPWL = 6$ and $q(6) = 1.2206$, we calculate the estimated wirelength to be 7.3236 which is very close to the effective total wirelength which is 7.

5.2.2 Timing Cost

All placement tools compared in this thesis keep track of the critical paths in a placement by building a timing graph. The timing graph is a directed acyclic graph. The vertices represent block pins such as CLB, LUT and FF pins, ... The edges represent timing arcs and each edge is assigned a delay depending on the physical delay and position of the

blocks on the device. The timing cost is calculated by multiplying the delay of all the timing edges with their respective criticality:

$$C_{timing} = \sum_{e \in edges} Crit(e) \cdot T_{delay}(e) \tag{5.2}$$

To speed up the calculation of T_{delay} a delay lookup table is built before placement by routing connections with a different Manhattan distance and saving the routing delay. The criticality of an edge is calculated during a timing graph update as follows:

$$Crit(e) = \left(\frac{Slack(e)}{WNS} \right)^{\epsilon} \tag{5.3}$$

with WNS the worst negative slack in the timing graph and ϵ the criticality exponent. The slack is calculated during static timing analysis, as explained in section 4.4.1. The routing delay between blocks is optimistically estimated as the shortest path between the physical block locations. Before placement all the possible shortest paths between the blocks are calculated and kept in a delay lookup table. During static timing analysis the delay lookup table is then used to calculate the slacks.

The total placement cost is calculated as a weighted sum of C_{timing} and C_{wl} with the timing tradeoff factor α;

$$C_{total} = \alpha \cdot C_{timing,total} + (1 - \alpha) \cdot C_{wl,total} \tag{5.4}$$

$$C_{\Delta} = \alpha \cdot \frac{C_{timing,\Delta}}{C_{timing,total}} + (1 - \alpha) \cdot \frac{C_{wl,\Delta}}{C_{wl,total}} \tag{5.5}$$

The timing and wirelength delta costs are normalized by the total timing or total wirelength cost of the placement respectively.

5.3 Simulated Annealing

We start with explaining the simulated annealing heuristic approach. We already introduced simulated annealing in the previous chapter, but here we repeat the most important basics.

Simulated annealing is an older although still relevant approach that is still used for FPGA placement by Altera. The algorithm has been fine tuned over many years and is implemented in the VPR suite. We also implemented our own version and use it as baseline in our comparative analysis.

Listing 5.1: Pseudo code for a simulated annealing algorithm.

```
1  function simulatedAnnealing():
2    s = randomSolution()
3    T = T_0
4    R = R_0
5    while not stopCondition:
6      repeat n times:
7        c_previous = cost(s)
8        s_proposed = randomAlteration(s, R)
9        c_proposed = cost(s_proposed)
10       Δc = c_proposed − c_previous
11       if Δc < 0 or e^(−Δc/T) > random([0, 1]):
12         s = s_proposed
13     T = nextTemperature(T)
14     R = nextR(R)
15   return s
```

5.3.1 The Basic Algorithm

The pseudo code of the algorithm is shown in Listing 5.1. The placement starts by randomly, but legally, placing the logic blocks in the input circuit on physical blocks of the FPGA architecture. Simulated annealing is an iterative algorithm in which a solution is repeatedly randomly altered and evaluated.

A random block in the input netlist and a random physical block location that is not the location of the random block are chosen. If there is a block assigned to the chosen physical block location, then the blocks are considered to be swapped. In the other case the chosen block is considered to be moved to the vacant physical block location. If the move/swap causes a decrease in placement cost, it is always accepted. To avoid getting stuck in local minima, sometimes a move/swap causing a higher cost is accepted as well. The probability of such a solution to get accepted depends on how much worse the solution is and the value of the temperature (T) at the current iteration. The move/swap is accepted with a probability of $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost due to the move. T, the temperature, controls the probability by which hill-climbing moves/swaps are accepted. Initially, T is very high so that most moves are accepted, which allows a thorough exploration of the solution space. Gradually T is decreased so that the probability by which hill-climbing moves are accepted decreases and the placement converges towards a close to optimal solution. The algorithm stops when the solution does not improve anymore.

Annealing Schedule The ensemble of the following parameters is called the annealing schedule: the initial temperature, the rate at which the temperature is decreased, the number of moves that are attempted at each temperature, the way in which potential moves are selected and the exit criterion. A good annealing schedule is crucial for finding a good solution in a reasonable amount of time. We used an annealing schedule (Equation (5.6)) with exponential cooling using a variable parameter γ. The goal of the variable parameter $\gamma(\beta)$ is to make the algorithm spend more time in the stage where the algorithm makes the most improvement, namely when the fraction of accepted alterations β is between 15% and 96%.

$$T_{new} = \gamma(\beta) \cdot T_{old} \tag{5.6}$$

$$\gamma(\beta) = \begin{cases} 0.5, & \text{if } 96\% < \beta \\ 0.9, & \text{if } 80\% < \beta \leq 96\% \\ 0.95, & \text{if } 15\% < \beta \leq 80\% \\ 0.8, & \text{if } \beta \leq 15\% \end{cases} \tag{5.7}$$

The same cooling schedule is used in VPR's simulated annealing placment tool. The schedule is described in more detail in [16]

During placement the algorithm has to be able to estimate the quality of the placement at all times, therefore the optimization goals are estimated based only on the location of the functional blocks on the device.

Random Alterations The random alterations that are used are a swap of two blocks and a move of a block to a vacant site. Only alterations are considered that move a block or swap blocks over a distance smaller than R in the x or y direction. The variable R is used to keep the fraction of accepted solutions, β, close to $\beta_{target} = 44\%$ in order to speed up convergence. It is updated once per temperature iteration according to the following equations.

$$R'_{new} = (1 - \beta_{target} + \beta) \cdot R_{old} \tag{5.8}$$

$$R_{new} = \max(R_{min}, \min(R'_{new}, R_{max})) \tag{5.9}$$

5.3.2 Fast and Low Effort Simulated Annealing

In the comparison in section 5.7 a low effort fast simulated annealing placement approach is used as a baseline. To achieve a low effort placement we tweaked the two following parameters of the temperature schedule:

- The kernel of an SA based algorithm is finding a move and calculating the cost difference. The kernel is performed N_{moves} times during each temperature step.

$$N_{moves} = b \cdot (N_{blocks})^R \tag{5.10}$$

In the experiments the effort level b is varied from 0.1 to 5.0. The value of R is set to a more scalable 1.0 instead of 1.333 as suggested by Sidiropoulos et al. in [118].

- The starting temperature is typically determined by trying to swap a number of blocks and depending on the swap cost the initial temperature is calculated as follows

$$T_{init} = M \cdot \sigma(\Delta C_{accepted}) \tag{5.11}$$

with $\Delta C_{accepted}$, an array with the cost differences of the accepted swaps and M, the temperature multiplier. In VPR $M = 20$ as first proposed in [59], which is typically too high. In our implementation we set $M = 1$ without significant loss of quality of results.

5.4 Analytical Placement

Another common placement technique is analytical placement. Our analytical placement implementation is largely based on HeAP [53]. Unfortunately there is no open source analytical placer available and so we built an analytical placer from scratch and made it publicly available [136]. The analytical placement implementation was also used as the foundation for LIQUID. It also serves a baseline in our comparative analysis in section 5.7.

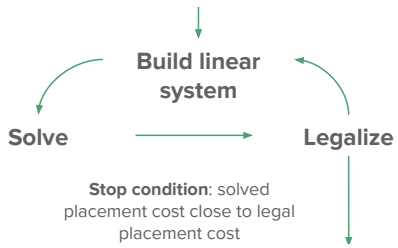


Figure 5.3: Analytical Placement Cycle

Listing 5.2: Pseudo code for our analytical placement implementation.

```
1  function analyticalPlacement():
2    s_legal = randomSolution()
3    c_legal = calculateCost(s_legal)
4    c_sol_lin_sys = 0.000001
5    while C_legal / C_solution > β:
6        sys = buildLinearSystem(s_legal)
7        s_lin_sys = solve(sys)
8        c_sol_lin_sys = calculateCost(s_lin_sys)
9        s_legal = legalize(s_lin_sys)
10       c_legal = calculateCost(s_legal)
11   return s_legal
```

5.4.1 High level overview

In Analytical Placement (AP) the placement problem is represented as a linear system. The linear system is built by taking the partial derivatives of the cost function with respect to the position of the movable blocks. The linear system is solved and results in the most optimal x- and y-locations for all movable blocks. However the solution contains rational numbers and on an FPGA only integer block coordinates coincide with legal placement positions. Furthermore, several blocks might be placed on the same location. Consequently the solution of the linear solver cannot be used as a final result. The solution should first be legalized. The legalization process will be discussed in section 5.6. After legalization the placement quality is typically degraded. Kim et al. [69] proposed a cyclic solve - legalize approach to solve the problem of overlapping blocks and degraded quality of the legalized solution. In Figure 5.3 the cycle is visualized and the pseudocode of the cyclic approach is listed in Listing 5.2. The placer always keeps track of two solutions: a non-legal solution obtained by solving the linear system and a legal solution obtained by legalizing the solution of the linear system. The algorithm performs multiple iterations. In each iteration one non-legal solved and one legalized solution are obtained. The amount of overlap between blocks is gradually reduced over consecutive iterations by adding so-called anchor connections to the system. These pseudo connections connect the moveable block to the last legalized position. By gradually increasing the weights of the pseudo connections the placement is gradually spread and converges closer to the legal placement:

$$w_\rho(i) = \rho_0 \cdot \rho^i \tag{5.12}$$

with ρ_0 the initial anchor weight, ρ the anchor weight multiplier and i the iteration number. In [53] w_ρ is increased linearly, but we have

found that exponential increase produces comparable results in slightly less iterations.

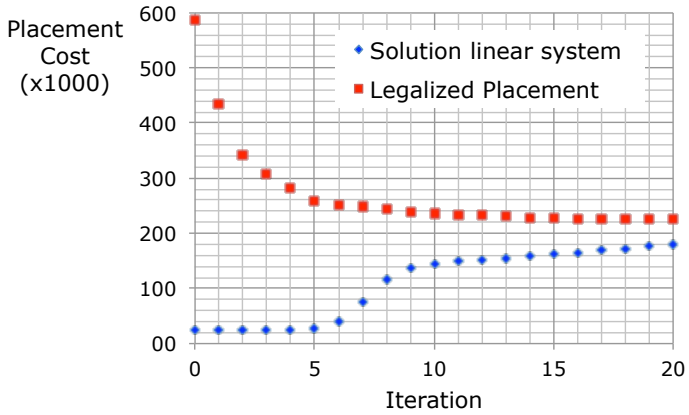


Figure 5.4: The evolution of the placement cost of the bitcoin_miner benchmark design for the 20 first iterations of the analytical placement process ($\rho = 1.2$)

Over the course of several iterations better legalized solutions are produced. The placement cost of the solved placement and the legalized solutions will gradually converge towards each other, as can be seen in Fig. 5.4. When the placement cost of the solved solution reaches a predefined fraction β of the legalized solution the algorithm stops.

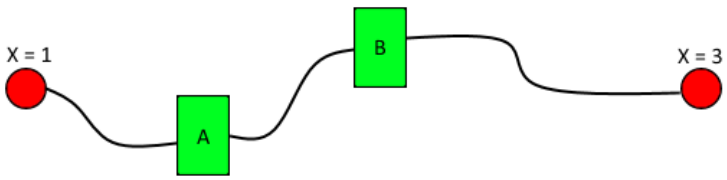


Figure 5.5: A simple netlist

5.4.2 Building the linear system

Let us consider the simple netlist depicted in Figure 5.5. The two red circles indicate fixed blocks, e.g. fixed IO block positions. We will only consider the placement problem in the x-direction here, as the solution in the y-direction is fairly easy to obtain: all blocks should be placed at the same y-location. The green rectangles indicate movable blocks, such as CLBs. The placement problem can now be formulated as find-

ing the most optimal locations of the movable blocks A and B such that the total necessary wire-length is minimized.

Let us first consider the sum of the squares of the wire-lengths in the x-direction.

$$C_x = (x_A - 1)^2 + (x_A - x_B)^2 + (x_B - 3)^2 \tag{5.13}$$

In order to find the values of x_A and x_B that minimize C_x;

$$\frac{\partial C_x}{\partial x_A} = (x_A - 1) + (x_A - x_B) = 0 \tag{5.14}$$

$$\frac{\partial C_x}{\partial x_B} = -(x_A - x_B) + (x_B - 3) = 0 \tag{5.15}$$

It is clear that we end up with a linear system of equations. This linear system of equations can be represented by a matrix and a vector:

$$\boldsymbol{A} \cdot \mathbf{x} = \mathbf{b} \; with \; \boldsymbol{A} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_A \\ x_B \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \tag{5.16}$$

Note that when the problem in the y-direction would not have been trivial it could be solved in exactly the same way as the problem in the x-direction. The problems in the x- and y-direction are always completely independent from each other and can thus be solved at the same time.

Now we found the placement that minimizes the square wire-length, but the optimization goal is the wire-length. The square wire-length is used because it makes $\frac{\partial \Phi_x}{\partial x} = 0$ meaningful equations and it can be represented by a symmetric and positive definite matrix which can be solved more efficiently with the conjugate gradient method than regular linear systems. To target the wirelength instead of the square wirelength constant weight factors are added for each term:

$$C_x = w_1 \cdot (x_A - 1)^2 + w_2 \cdot (x_A - x_B)^2 + w_3 \cdot (x_B - 3)^2 \tag{5.17}$$

More generally;

$$C_x = \sum_{i,j} w_{i,j} \cdot (x_i - x_j)^2 \tag{5.18}$$

In timing-driven analytical placement for example these weight factors are also used to incorporate timing information in the linear system formulation.

To trick the system in minimizing the wire-length instead of the square wire-length the weight factors are chosen as follows:

$$\begin{cases} w_1 = \frac{1}{|x'_A - 1|} \\ w_2 = \frac{1}{|x'_A - x'_B|} \\ w_3 = \frac{1}{|x'_B - 3|} \end{cases} \tag{5.19}$$

More generally;

$$w_{i,j} = \frac{1}{|x'_i - x'_j|} \tag{5.20}$$

with x'_A and x'_B the position of the blocks in the current placement. In case the current positions $x'_A = 0.5$ and $x'_B = 1.0$, then the minimization of (5.17) results in $\mathbf{x} = [\frac{4}{3}, \frac{5}{3}]$ and contains rational numbers. So this result should be legalized.

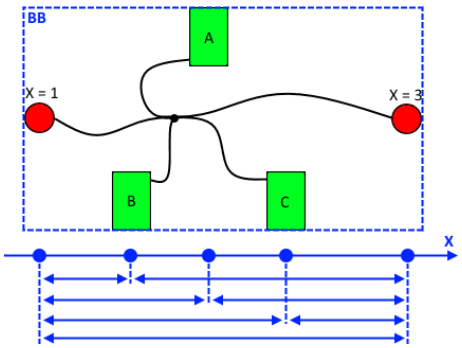


Figure 5.6: Bound-to-bound Net model

5.4.3 Bound-to-bound Net Model

In the toy example of Fig. 5.5 only nets with two terminals are considered. In Fig. 5.6 a net with 5 terminals is depicted. To build a linear system the higher fanout nets are split up in 1-to-1 connections. It is important that the number of connections is kept low and the added connections represent the effective cost as good as possible. The bound-to-bound net model first introduced in [122] represents the HPWL exactly with $2 \cdot (n-2) + 1$ connections per dimension.[1] The weights of (5.19) are adapted to take into account the number of pins and the number of times the bound-to-bound model accounts for the HPWL:

$$w_{AB} = \frac{1}{n-1} \cdot \frac{1}{|x'_A - x'_B|} \cdot q(n) \tag{5.21}$$

[1]The number of connections is $2 \cdot (n-2) + 1$ only in case each block inside the bounding box is a moveable block.

with n the number of terminals and a weighting factor q(n) taken from [25], see Table 5.1.

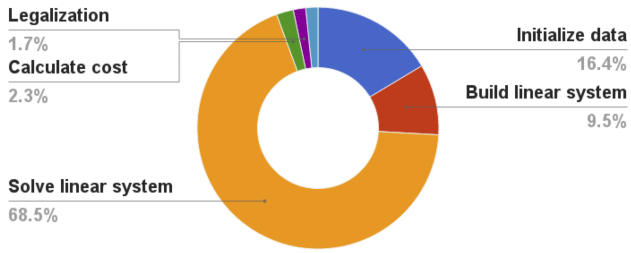


Figure 5.7: The runtime breakdown for analytical placement ($\rho = 1.1, \beta = 0.8$) in wire-length driven mode to reach a placement cost of 221.6K for the *bitcoin_miner* design.

5.4.4 Runtime Breakdown

The donut chart in Fig. 5.7 shows the runtime breakdown for the analytical placement of the bitcoin_miner design. The bitcoin_miner design is the largest Titan23 benchmark design that we were able to place on our 32GB workstation. By far the largest runtime consumer is solving the linear system with 68.5% followed by initializing the data with 16.4% and building the linear system with 9.5%. In the development of LIQUID we focused on reducing the runtime of solving the linear system, because it accounts for the majority of the total runtime.

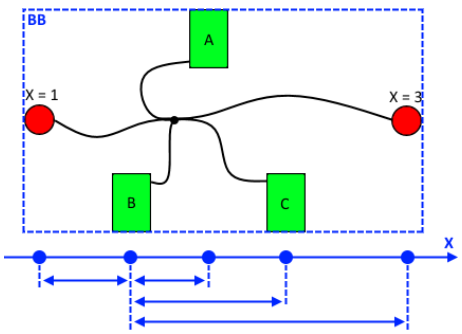


Figure 5.8: The source-sink connections of a net that are added to the system if the connection's criticality exceeds the criticality threshold. The source pin of the net is located in block B.

5.4.5 Timing-Driven Analytical Placement

In our timing-driven analytical placement we add a third type of connections to the linear system: source-sink connections. As explained in 5.2.2 every connection corresponds with an edge in the timing graph and has a criticality (5.3). A connection from block i to block j is added to the linear system with the weight

$$w_{i,j} = \begin{cases} 0, & crit < \theta_c \\ \alpha \cdot \frac{Crit^{\epsilon c}}{|x'_i - x'_j|}, & \text{otherwise} \end{cases} \tag{5.22}$$

There are three parameters that determine the weight of the source-sink connections:

- The criticality threshold θ_c: Source-sink connections with a criticality lower than θ_c are not added to the linear system. Increasing θ_c speeds up calculations, but only to a certain extent. The runtime and quality penalty of adding too much connections is small so the default value is set to 0.8.

- The timing trade-off factor α determines the importance of the source-sink connections compared to the bound-to-bound connections. Setting the timing tradeoff high does not hurt the quality of the placement in terms of wirelength much but improves the critical path delay so the default value is set to 30.

- The criticality exponent ϵ. A higher criticality exponent puts more emphasis on minimizing the wirelength of the most critical connections, at the cost of an increased total wirelength. The default value is set to 1.

5.5 Liquid

5.5.1 The Basic Algorithm

LIQUID is based on the same cyclic approach used in AP. Each iteration consists of an optimization phase and a legalization phase. The biggest difference between LIQUID and AP is the linear solving step. AP builds a linear system and calculates the exact solution to this system. LIQUID approximates this solution in an iterative way. This iterative procedure resembles a classical gradient descent solving procedure. In every iteration, all the blocks are moved over a small distance. The directions

and sizes of these steps are chosen so that the total cost function decreases. The steps are calculated for each block individually: there is no global view of the optimization problem at hand.

5.5.2 Modeling the Problem

The problem is modeled in a slightly different way than is the case for AP. Every block is connected with extended springs to fixed locations and other movable blocks. Every one of these springs exerts a pulling force on the block. The magnitude F_s of this force increases with the wire length. Every iteration these forces are combined into a single force with a direction and a magnitude F_c. In that iteration the block will move in that direction over a distance F_c.

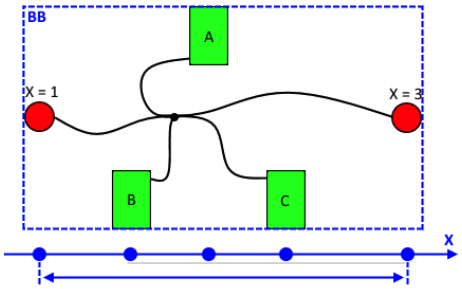


Figure 5.9: The Extremes Net Model.

The Extremes Net Model

The x and y problem are treated separately, as was the case for AP. We only describe the x problem; the y problem is completely analogous. The cost function for wirelength-driven placement is the sum of the half-perimeter wirelengths of all the nets in the circuit. Minimizing the cost function in the x-dimension is accomplished by minimizing the widths of all nets simultaneously. The width of a net is determined by its leftmost and rightmost block: the extreme blocks on the edge of the bounding box. The positions of the internal blocks do not matter.

For every net in the circuit we only add one spring between the two extreme blocks. This is the ”extremes net model” as shown in Figure 5.9. As a result of this single spring the two extreme blocks will be pulled towards each other without affecting the internal blocks. They are not important for the cost of the net in question, but they could be

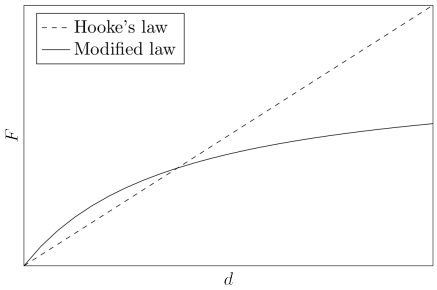


Figure 5.10: Hooke's law and the modified law that has a maximum force.

relevant to other nets they are attached to and we want to give these blocks all the freedom in this aspect.

At first sight this rudimentary net model doesn't seem to work very well: it does not keep all the blocks in the net together. If an internal block of the net is also an extreme block in some other net, as is often the case, it can be pulled away far from the other blocks of the net. However every linear solving step consists of multiple iterations (up to fifty for high-quality solutions). Every iteration different blocks end up at extreme positions. Over the course of multiple iterations the movement is smoothed and the placement converges to a good solution. Moreover we have found that this simple net model introduces some randomness to the blocks's movements that helps to escape local optima. Experiments with the star and bound-to-bound net models yielded inferior results.

Spring Force

The magnitude of a spring's pulling force F_s depends on the distance d_{b_1,b_2} between the two blocks. If the springs were to obey Hooke's law, the force would be proportional to this distance:

$$F_s = k \cdot d_{b_1,b_2} \tag{5.23}$$

where k is the spring stiffness. All springs have the same stiffness: this is a placer-wide parameter that influences the step size of the blocks. We have found that the solution quality increases if we limit the maximal force a spring can exert. Otherwise long distance connections would be represented by a very high force and they would overshadow the smaller distance connections. We use the following formula:

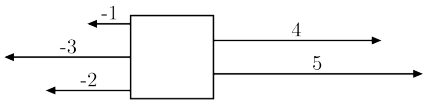


Figure 5.11: A block on which five forces pull

$$F_s = k \cdot \frac{d_{b_1,b_2} \times d_{max}}{d_{b_1,b_2} + {}^{d_{max}}/_2} \tag{5.24}$$

where d_{max} is a placer-wide parameter. Using this formula an infinitely long spring will only exert the force that a spring of length d_{max} would exert according to Hooke's law. Figure 5.10 plots Hooke's law and our modified law. $d_{max} = 30$ is the value that produces the best results.

Combining the forces

Only the nets attached to the block are under consideration. This is an important insight that allows us to consider each block separately. Additionally only the blocks on the edge of the bounding box have a direct impact on the HPWL cost.

As most blocks are part of a large number of nets, many are an extreme block in more than one net. This results in multiple forces pulling on a single block, e.g. as shown in Figure 5.11. The most straightforward way to combine these forces is to simply sum them. In our small example this would result in a force that points to the right and has a magnitude of 3. However moving the block in the direction of this force would not decrease the total cost function: this block is the rightmost block in three nets, and it is the leftmost block in two other nets. Moving the block to the right over a distance Δx increases the width of three nets with Δx each, and decreases the width of two other nets with Δx each. The net cost function increase is equal to Δx.

To overcome this problem LIQUID always moves blocks in the direction with the maximum cardinality, the highest number of forces. This is the direction for which the total wirelength cost function reduces the most, so the steepest gradient. The step size, $\Delta_{x,step}$, is taken equal to the average force magnitude in that direction. We do not sum the magnitudes of the forces because we want the step size to be independent of the number of nets a block is in. In the example of Figure 5.11 this calculation leads to a step of size 2 to the left.

Timing-Driven Springs

The timing-driven version of LIQUID adds a second type of springs: between the net source and net sinks (see Figure 5.8). The magnitude of these forces is calculated in the same way as the wirelength-driven forces, i.e. according to Equation 5.24. To balance the importance of wirelength objectives against timing objectives we introduce a new property for every force: the tradeoff weight.

Every spring in the system is given a weight w_t. The weight of wirelength-driven springs is fixed to 1. Only springs are added for connections with a criticality larger than θ_c. The weight of timing-driven springs is closely related to the weight of timing-driven connections in timing-driven AP (see Equation 5.22):

$$w_t = \begin{cases} 0, & crit < \theta_c \\ tradeoff \times crit^{\epsilon_c}, & \text{otherwise} \end{cases} \tag{5.25}$$

The placer-wide parameters θ_c, $tradeoff$ and ϵ_c are explained in Section 5.4.5.

To determine the direction that a block will move to LIQUID does not simply count the number of springs on each side anymore: it now sums the tradeoff weights of the springs attached to the block. The block will move in the direction that has the highest sum. Once the direction is known, the tradeoff weights do not influence the magnitude of the step: the step size still equals the average of the force magnitudes, as explained in Section 5.5.2

Including Legalized Positions

LIQUID follows the same cycle of linear solving and legalization as AP. In the first linear solving step LIQUID places all movable blocks very close to each other near the center of the FPGA, much in the same way that AP does. In all subsequent linear solving steps LIQUID incorporates the legal positions found in the most recent legalization step.

AP includes legal positions by means of pseudo-connections. These are forces that pull a movable block to it's legal position. These forces are entirely equivalent to the other forces in the linear system. This approach does not work very well in LIQUID, because it is not possible to gradually increase the importance of a connection as the placer progresses. We tried to adopt the "weight" concept introduced inSection 5.4.1 for pseudo-springs. In wirelength-driven mode it only makes sense to increase this weight in steps of 1, because of the steepest gradient descent procedure. However the number of springs attached

to an average block is relatively small, so the effect of increasing the weight by 1 is large. So we propose the following solution based on linear interpolation.

Linear Interpolation We call x_i the position of a block at the end of iteration i, and x_l the legal position for that block found in the last legal solving step. If we ignore the legal position of a block, the block movement is only determined by the wirelength- and timing-driven springs. We calculate a good new position for the block as explained in the previous paragraphs. We call this position x_{opt}. We combine these two different positions by interpolating linearly between them:

$$x_i = (1 - w_\rho)x_{opt} + w_\rho x_l \tag{5.26}$$

w_ρ is a placer-wide variable that indicates the importance of a block's legal position while solving linearly. It changes as the placement progresses. In the first linear solving step w_ρ is equal to 0: the legal position is not taken into consideration. Every linear solving step a fixed value $w_{\rho,step}$ is added to w_ρ: the influence of the legal position increases. The placer stops when w_ρ reaches a predefined value $w_{\rho,stop}$, typically chosen below and close to 1.

We have now introduced two new parameters that can be changed to make a trade-off between the placer speed and quality of the final placement: $w_{\rho,step}$ and $w_{\rho,stop}$. We want to highlight an important consequence of this method: the number of solving steps is determined only by the pre-defined parameters, and can be calculated prior to starting the placement. In AP algorithms execution is stopped when the placement quality reaches some satisfactory level. For this the placement quality has to be recalculated every iteration. In LIQUID we never need to know the current placement quality, this means we can avoid a lot of time-consuming calculations.

5.5.3 Momentum Update

We introduced an momentum update, because it smoothes out the jittery steepest gradient descent vectors. As an analogy we consider a particle with speed $\mathbf{v} = 0$ at some random location in a hilly landscape. Gravity is a conservative force so the particle has a potential energy, $U = mgh$ and the force can be expressed as $\mathbf{F} = -\nabla\mathbf{U}$. In LIQUID we try to simulate the particle rolling down to the bottom and apply it to the linear system representing the FPGA placement problem. In several steps we update the speed and the location of all the blocks. The blocks

are given a certain mass. During simulation, the blocks built up momentum and can escape local minima. The new location and speed of the block, $\mathbf{x}_i$ and $\mathbf{v}_i$ are updated following Nesterov's momentum update. Nesterov's Accelerated Gradient Descent is an optimal method for smooth convex optimization [107] and is widely used for training deep convolutional networks [66]. The speed of each block at the beginning of the optimization phase is $\mathbf{v}_0 = 0$. The start location of the blocks, $\mathbf{x}_0$, is determined by the previous legalization procedure (or random in the first iteration). First the x-location is updated, $\mathbf{x}'$, with the momentum the block still has from the previous speed $\mathbf{v}_{i-1}$:

$$\mathbf{x}' = \mathbf{x}_{i-1} + \mu \cdot \mathbf{v}_{i-1} \tag{5.27}$$

Subsequently the velocity is integrated:

$$\mathbf{v}_i = \mu \cdot \mathbf{v}_{i-1} - \gamma \cdot \nabla \mathbf{C}(\mathbf{x}') \tag{5.28}$$

with $\nabla \mathbf{C}(\mathbf{x}')$ the gradient of the cost function for position $\mathbf{x}'$, calculated as explained in Section 5.5.2, and μ the coefficient of friction. Friction dampens the velocity and reduces the kinetic energy of the system, otherwise the block would never come at a stop. γ is the gradient sensitivity rate and can be compared to the product of the mass of the particle and the gravitational constant. μ and γ are parameters of the algorithm that mainly affect the quality of the placement. With the new speed the new position is found by integrating:

$$\mathbf{x}_i = \mathbf{x}' + \mathbf{v}_i \cdot \Delta_{x,step} \tag{5.29}$$

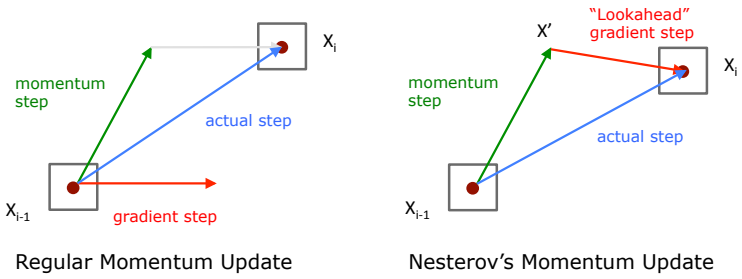


Figure 5.12: The Nesterov momentum update versus regular momentum update

The Nesterov's momentum update converges slightly better than the regular momentum update and it is theoretically proven that it is an optimal first order method for convex optimizations [107]. The difference between Nesterov's momentum update and the regular momentum update is clarified in Fig. 5.12. The main difference is that the

Listing 5.3: Pseudo code for the inner optimizaton function.

```
1  function optimize(sys, n_inner):
2    for i in range(n_inner):
3        for block b in sys:
4            b.x = b.x + μ · b.v
5            v_force = w_ρ · (b.ll − b.x) + (1 − w_ρ) · ∇C(b.x)
6            b.v = μ · b.v + γ · v_force
7    return s_legal
```

momentum update is performed first before calculating the steepest gradient descent vector.

Up until now we neglected the anchor positions in our description. In order to converge towards a good legalized placement, (5.28) is updated to include a weighted sum of the last legal position of the block $\mathbf{x}_l$ and $\nabla\mathbf{C}$ based on the anchor weight w_ρ as explained in Section 5.5.2;

$$\mathbf{v}_i = \mu \cdot \mathbf{v}_{i-1} - \gamma \cdot (w_\rho \cdot (\mathbf{x}_l - \mathbf{x}') + (1 - w_\rho) \cdot \nabla\mathbf{C}(\mathbf{x}')) \tag{5.30}$$

The new block positions and speeds can be calculated for each block individually based on the previous positions of other blocks. There is no global view of the optimization problem at hand. Once all the new positions and speeds are calculated, the old positions and speeds are discarded. Next, a new iteration is started in which new positions and speeds are calculated. The momentum update is summarized in Listing 5.3.

5.5.4 Optimizations

Effort Level

Every optimization phase consists of a number of steepest gradient descent block movement iterations. We call the number of iterations in a solving step the effort level. It is an important parameter to control the placement quality: a higher effort level leads to a higher QoR.

LIQUID does not stop when the optimised placement cost reaches a fraction of the legalized placement cost as is the case for our analytical placer. Instead we empirically devised a fixed schedule in which all parameters are inferred from the number of outer loop iterations N_{outer}, which can be used as a single effort level tuning parameter. An outer loop iteration contains an optimization phase and a legalizing phase.

We have found that as the algorithm progresses the number of inner iterations which perform momentum updates can be reduced without

compromising placement quality. The number of iterations in a solving step is interpolated linearly between e_{first} in the first solving step and e_{last} in the last step. Normally one would choose these so that $e_{last} \ll e_{first}$.

The first solving step must start from a random placement. We considered it likely that the first solving step might require more iterations to reach a good solution, so we tried to increase the number of iterations for just the first step. Experiments have shown that this did not improve the runtime-quality trade-off. Linearly decreasing the weights worked very well.

Let i be the outer loop counter then the anchor weights are calculated as follows;

$$w_\rho(i) = i \cdot \frac{w_{\rho,target}}{N_{outer} - 1} \tag{5.31}$$

with $w_{\rho,target}$ the final anchor weight with a default value of 0.85. The number of inner loop iterations (number of momentum updates) N_{inner} starts at N_{outer} and is linearly decreased to end at 1 in the final outer iteration.

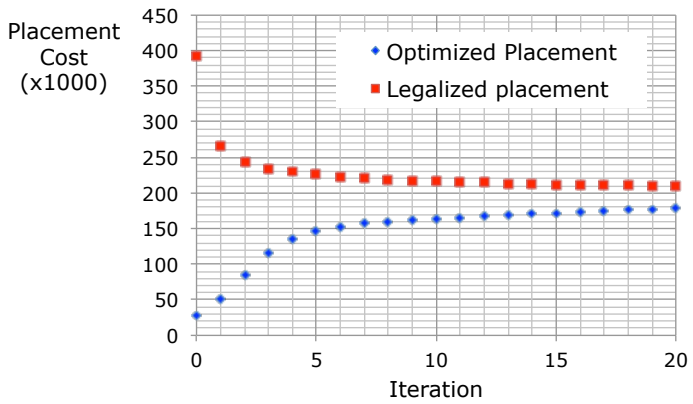


Figure 5.13: The evolution of the placement cost of the bitcoin_miner benchmark design when placing with LIQUID for the 20 first iterations of a run with $\rho = 1.2$

In Figure 5.13, the evolution of the placement cost of the legalized and optimized placement of the bitcoin_miner design is shown for the first 20 iterations for $N_{outer} = 37$. The evolution of the optimized placement cost for LIQUID looks slightly different when comparing with analytical placement in Figure 5.4. The main cause for this difference is the different way in which the anchor weights are updated. In our

analytical placement the weights are updated exponentially, see Equation (5.12) and in LIQUID they are updated linearly, see Equation (5.31)

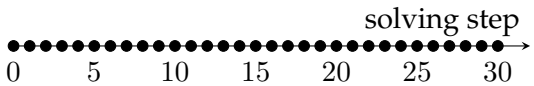
Recalculate Criticalities

The timing graph keeps the criticality for each connection. These criticalities depend on the positions of the blocks, and should be updated as the placement algorithm progresses. Initially we recalculated the criticalities prior to each optimization step, as shown in Figure 5.14a. This recalculation is very time consuming. We have found that the QoR doesn't deteriorate when the criticalities are only recalculated after some of the legal solving steps, for example 40% as shown in Figure 5.14b. How often the criticalities are recalculated can be controlled using a parameter we call F_c. For values lower than 0.4 the QoR usually decreases.

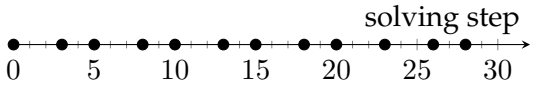
We expected that recalculations would be more important near the beginning of the algorithm. As the algorithm progresses a single step perturbs the placement less and less, so we assumed changes in the timing graph would also reduce. This means recalculations would optimally be distributed as in Figure 5.14c. Experiments have shown that this assumption is not true: the QoR for uniform recalculation distribution is comparable to non-uniform distributions. In further experiments we always used a uniform distribution.

5.5.5 Runtime Breakdown Comparison

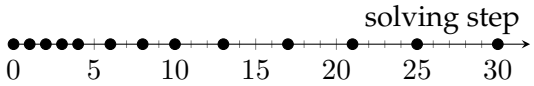
The runtime breakdown of the analytical placer and LIQUID for the wirelength-driven mode are charted in Fig. 5.15 for placing the *bitcoin_miner* benchmark design with a same placement quality with the total wirelength at 221.6K. Different qualities can be reached by changing the effort level, which is the anchor multiplier for AP and N_{outer}, the number of outer iterations for LIQUID. A total wirelength of 221.6K is chosen because beyond this point the analytical placer requires much more time to improve the quality of the design. According to the breakdown we have succeeded in our intent to reduce the time to solve or optimize the system which is indicated in green in the bar chart. The gradient descent based approach in Liquid is with 0.1s much faster than solving the linear system with 79.1s. This leads to an overall 3.9x runtime speedup.



(a) $F_c = 1$



(b) $F_c = 0.4$, $P_c = 1$



(c) $F_c = 0.4$, $P_c = 2$

Figure 5.14: Recalculating the criticalities every step as in (a) is not necessary. Fewer recalculations can be spread (b) uniformly or (c) with an emphasis on the first solving steps.

5.6 Legalization

We describe legalization in a seperate section because it is both used in LIQUID and our analytical placement implementation. The fundamentals of our legalizer are the same as those of the legalizer described in HeAP [53]. However the details provided in that publication are scarce and no source code is publicly available.

Our legalization method legalizes the blocks per type in three steps. In the first step each block is mapped to the closest legal position. If there are no overlapping blocks than legalization is finished, but typically there are blocks occupying the same legal position. These blocks are spreaded gradually in the next steps. In the second step the legalizer builds a set of regions on the FPGA. The regions are rectangular and should not overlap. Every region should also contain enough positions for the blocks it contains.

The legalizer starts at the center of the FPGA and then it spirals outwards until al positions have been investigated. Positions containing ≤ 1 blocks or positions that are already assigned to a region are neglected. Starting from an overused position a region is grown in search for more empty block positions. The region is expanded in one direction at a time in a round robin fashion to keep the region rectangular

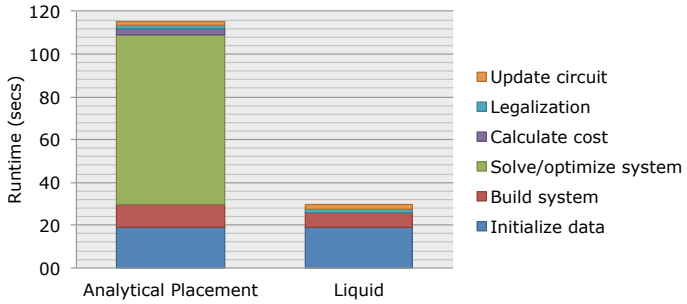


Figure 5.15: The runtime breakdown for analytical placement ($\rho = 1.1, \beta = 0.8$) and LIQUID ($N_{outer} = 17$), both in wire-length driven mode to reach a placement cost of 220K for the *bitcoin_miner* design.

and the initial block in the center of the region. The region is expanded until there are enough legal positions for the blocks it contains. In case a new expansion contains blocks they are absorbed in the same region. In case an expansion contains a block that is already assigned to a region, the two regions are merged. In Fig. 5.16 is illustrated how a region is grown on a small FPGA fragment.

Once we have a set of disjunct under-utilized regions, the blocks inside the regions are legalized separately. The blocks and the region are bi-partitioned recursively and equally until each block in the region has its own position.

5.7 Experiments

The main purpose of the experiments is investigating the runtime-quality tradeoff for LIQUID and AP for large heterogeneous designs. To investigate the runtime-quality tradeoff the effort levels of the placements tools are swept.

5.7.1 Methodology

The Titan23 designs are used for benchmarking. The designs are packed by VPR 7.07 with default settings. VPR failed packing the *LU_Networks* benchmark. For *gaussianblur* and *directrf* our workstation ran out of memory [106]. What remains are the 20 benchmark designs listed in Table 5.2. The target device for the Titan23 designs is Altera's Stratix-IV FPGA.

LIQUID and the conventional analytical placer are implemented in the FPGA placement framework, which is available online [136]. The

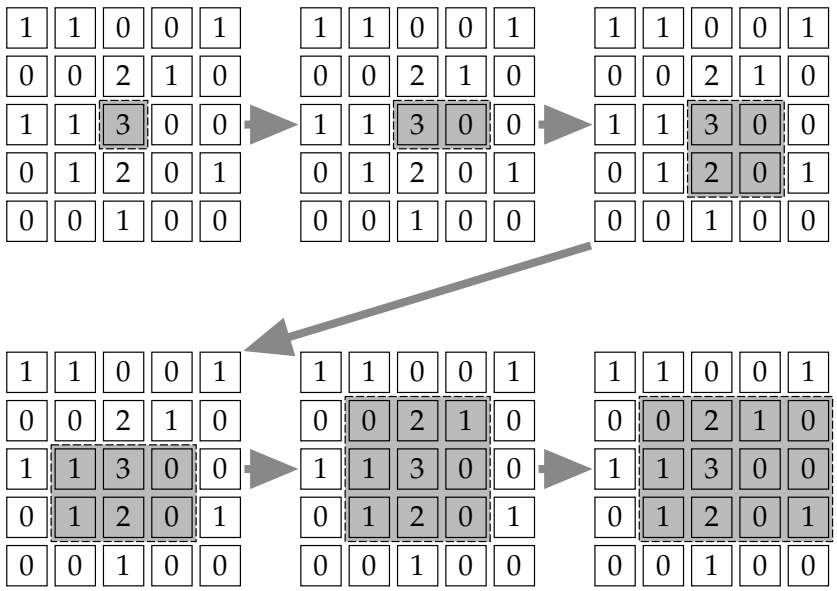


Figure 5.16: Growing a region until it is not overutilized anymore. The number in the rectangles denote the number of blocks assigned to that position

framework is written in Java and the experiments are performed with OpenJDK 64-Bit Server VM v24.95 on a workstation with an Intel Core i7-3770@3.40GHz and with 32GB 1600Mhz DDR3 work memory running Scientific Linux 6.5.

5.7.2 Runtime versus Quality

To make a fair comparison in the following sections we first give an overview of the capabilities of both LIQUID and analytical placement. For both algorithms we looked at the parameter that influenced the runtime-quality tradeoff the most. For Liquid this is N_{outer}, the number of outer iterations. N_{outer} is varied from 3 to 96. For analytical placement the most sensitive parameter is ρ, the anchor weight multiplier. ρ is varied from 1.005 to 2.4. The placement results in terms of the critical path delay are charted in Fig. 5.17 and in terms of wirelength in Fig. 5.18 in respect to the runtime of the placement tool. All metrics reported are the geometric mean of the values for the individual benchmark designs. The results are post-placement estimates, because routing all the pareto points is computational too intensive. Using estimates also allows us to also use the largest Titan23 benchmark designs

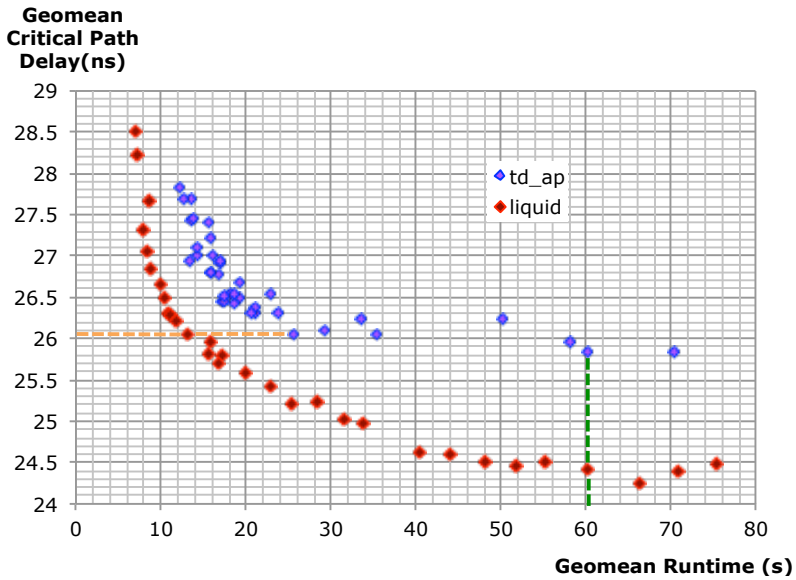


Figure 5.17: Pareto front for LIQUID and analytical placement: critical path delay versus runtime

which VPR is unable to route. In Section 5.7.6 the post route quality for a few important points is presented and the ratios of post route results are almost identical to the ratios of the placement estimates. So we advocate that the placement estimates are very good and can be used to make conclusions.

We can clearly observe that the placement results of LIQUID are superior to the placement results of analytical placement. All the data points on the Pareto front of Liquid have either a better runtime or a better critical path delay and wirelength cost with their respective counterparts on the Pareto front of the analytical placer. While developing LIQUID we hypothesized that we can rely on the legalization phase to disturb the solution enough to keep the placement from crystallising and converging to a local minimum. It seems that this hypothesis is correct, because we are able to obtain better quality placement results.

Another observation is that the data points for analytical placement show more variance. The behaviour of the analytical placement tool seems more erratic. The main cause is that the initial placement determines the weights of (5.19) in the first iteration, which in turn influences the result of the linear system. The analytical placer is set on different courses depending the initial placement. For LIQUID we see a more smooth progress, because simulating an accelerated gradient descent process gradually changes the placement's structure, which leads to a

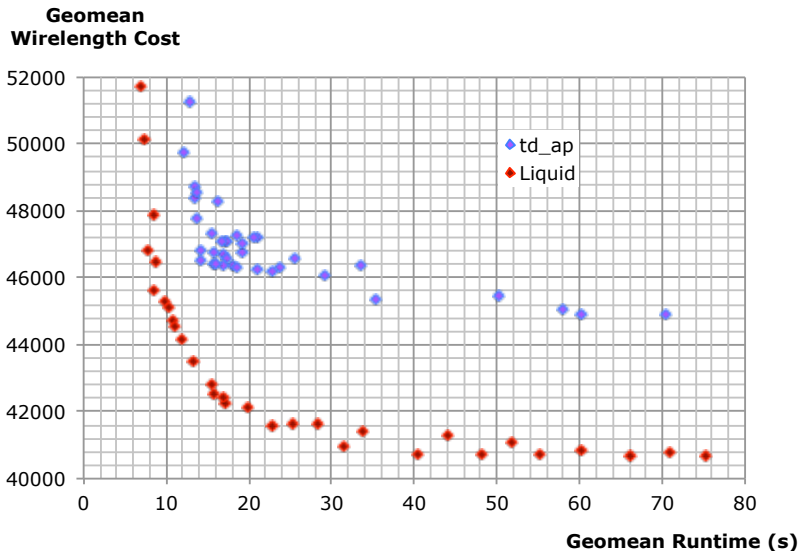


Figure 5.18: Pareto front for LIQUID and analytical placement: wire-length versus runtime

more consistent placement improvement.

5.7.3 Runtime Speedup

To investigate the runtime speedup we measure how fast LIQUID is when trying to achieve the same quality as analytical placement. The datapoints chosen for the comparison are indicated by the orange striped line in Fig. 5.17. We choose the datapoint ($\rho = 1.11$) in the knee of the analytical placement's curve, so AP can comfortably achieve this quality without slowing down too much. We call this modus *AP-Fast*. To reach the same quality LIQUID needs 20 outer iterations, this setting is called *Liquid-Fast*. The results are reported per benchmark design in the first two major columns of Table 5.2. The results in the first major column *Liquid-Fast* are reported in real numbers. The other datapoints in Table 5.2 are reported relative to *Liquid-Fast*. *Liquid-Fast* and *AP-Fast* obtain on average the same quality, but LIQUID is able to produce the results 1.93x faster. Additionally the total wire-length cost of LIQUID's placement is 7% lower. However, we should note that there is a lot of variability if we look across the different designs for all metrics. The speedup is clearly visible with almost all runtimes of AP-fast being slower, but the wire-length and critical path delay results are more erratic, which can be explained by the fact that we are investigating the

Table 5.2: Runtime (RT), wire-length (WL) and critical path delay (CPD) for analytical placement (AP) and Liquid. The results are reported relative to **Liquid-Fast**

	Liquid - Fast			**AP - Fast**			**Liquid - HQ**			**AP - HQ**			**SA - HQ**		
Setting	$N_{outer} = 20$			$\rho = 1.11$			$N_{outer} = 80$			$\rho = 1.018$			$R = 1.0, b = 6$		
Design**Metric**	**RT[s]**	**WL[K]**	**CPD[ns]**	**RT**	**WL**	**CPD**	**RT**	**WL**	**CPD**	**RT**	**WL**	**CPD**	**RT**	**WL**	**CPD**
bitcoin_miner	57.86	215.97	16.90	1.29	1.12	1.03	4.30	0.95	0.95	3.59	1.13	1.01	49.32	1.23	0.65
bitonic_mesh	19.86	66.79	17.10	1.53	1.08	1.07	3.69	1.01	1.05	3.35	1.11	1.06	36.47	1.11	0.92
cholesky_bdti	11.33	32.93	11.46	1.61	1.32	0.95	5.26	0.91	1.03	2.48	1.33	0.99	58.64	1.15	0.81
cholesky_mc	4.97	12.35	9.67	1.44	1.05	0.94	4.96	1.02	0.90	1.77	1.06	0.93	65.66	1.05	0.87
dart	8.99	31.02	17.88	2.84	1.09	1.14	5.56	0.98	0.94	6.26	1.07	1.11	49.22	1.40	0.88
denoise	24.59	55.91	1472.31	8.22	0.77	0.74	5.66	0.90	0.84	41.49	0.78	0.75	62.57	0.92	0.77
des90	7.59	29.41	15.10	3.94	1.00	0.99	4.85	1.00	1.00	11.23	0.98	1.02	50.10	1.08	0.94
gsm_switch	17.15	103.32	15.52	2.95	1.12	1.15	5.17	0.95	0.93	4.97	1.08	1.12	65.54	1.09	0.84
LU230	34.15	281.23	39.93	1.20	1.27	0.71	3.79	0.90	0.69	6.55	0.80	0.70	49.53	0.78	0.73
mes_noc	37.15	85.92	20.53	1.98	1.11	0.95	3.88	0.96	0.93	1.90	1.14	0.96	40.38	1.11	0.76
minres	14.08	52.57	10.56	1.10	0.91	0.98	3.79	0.92	0.99	4.31	0.85	0.93	34.81	0.88	0.90
neuron	3.01	13.12	11.09	0.39	1.24	1.14	5.40	0.95	0.97	0.38	1.24	1.14	49.08	0.81	0.97
openCV	15.96	47.85	12.65	1.57	1.36	1.17	3.66	0.93	0.93	5.94	1.08	1.11	37.41	1.14	0.99
segmentation	15.46	25.96	1297.12	5.94	0.82	0.85	4.84	1.01	0.87	31.59	0.83	0.84	48.27	0.93	0.83
SLAM_spheric	9.21	24.11	110.28	6.80	0.83	0.93	5.02	0.87	0.89	37.74	0.83	0.93	53.44	0.90	0.89
sparcT1_chip2	42.04	123.76	31.51	1.84	1.19	1.17	3.49	0.90	1.02	2.00	1.25	1.18	43.80	0.97	0.98
sparcT1_core	3.56	12.43	10.24	3.08	1.17	1.20	6.15	0.97	0.98	10.19	1.12	1.18	60.48	1.02	0.93
sparcT2_core	17.49	50.45	13.19	2.93	1.13	1.12	3.95	0.96	1.01	3.76	1.13	1.10	46.55	0.93	0.90
stap_qrd	14.41	45.84	11.82	1.02	0.83	0.90	3.56	0.87	0.92	4.33	0.82	0.91	58.59	0.95	0.76
stereo_vision	3.07	10.62	11.90	0.38	1.31	1.05	5.66	0.87	0.97	0.38	1.31	1.05	46.66	0.89	0.94
Geomean	**13.22**	**43.48**	**26.04**	**1.93**	**1.07**	**1.00**	**4.56**	**0.94**	**0.94**	**4.55**	**1.03**	**0.99**	**49.49**	**1.01**	**0.87**
Std. Dev.				2.13	0.18	0.14	0.85	0.05	0.08	12.38	0.18	0.13	9.30	0.15	0.09

HQ stands for high quality

low effort modus of both placers.

5.7.4 The Best Achievable Quality

To investigate the best achievable quality we gave both placement tools more time. We choose a geomean of 60s, because beyond this point the quality did not improve much for both placers. The datapoints under investigation are indicated by the green striped line in Fig. 5.17. For AP this comes down to $\rho = 1.018$, we call this *AP-HQ* and for LIQUID this equates to setting the number of outer iterations to 80. We call this modus *Liquid-HQ*. The individual results of the benchmark designs are reported in the third and fourth major column of Table 5.2. All results are reported relative to *Liquid-Fast*. For *AP-HQ* we don't observe much quality improvement compared to *AP-Fast*, 1% improvement for the critical path delay and 4% for the wirelength cost. For *Liquid-HQ* on the other hand a clear quality improvement compared to *Liquid-Fast* can be observed with a 6% improvement for both the critical path delay and the wirelength cost and clearly a lower variance in the results for the different benchmark designs. Overall this leads to a gap between *AP-HQ* and *Liquid-HQ* of 5% for the critical path delay and 9% for the wirelength cost.

5.7.5 Comparison with Simulated Annealing

In Table 5.2 we added the results for a high effort simulated annealing run, *SA-HQ*, with the effort set to $b = 6$ (Equation 5.10). LIQUID in low effort modus achieves the same quality in terms of wirelength as *SA-HQ* in only a fraction of the time, 49x faster. However the critical path delay of the result is 13% worse. If we take a look at the high quality modus of LIQUID which is 11x faster than *SA-HQ*, then the critical path delay gap reduces to 7% and the wirelength further improves with a 7% decrease in comparison with *SA-HQ*. We can conclude that the current version of LIQUID optimizes for wirelength too aggressively and should focus more on critical path delay. To achieve this new ways to incorporate timing information in LIQUID should be investigated, such as non linear timing cost functions. We also have to point out that our analytical placement tool is also not able to reach the critical path delay results of *SA-HQ*. Improvements to our legalizer could help here to improve the timing results for both LIQUID and AP.

Table 5.3: Post-route Quality Results. The geomean ratios for the total wirelength and critical path delay are reported relative to the post-route results of *Liquid - Fast* ($N_{outer} = 20$)

	Effort	Total Wirelength	Critical path delay
AP - Fast	$\rho = 1.11$	1.08	1.02
Liquid - HQ	$N_{outer} = 80$	0.92	0.91
AP- HQ	$\rho = 1.018$	1.05	1.00
SA - HQ	$b = 6$	1.03	0.87

5.7.6 Post-route Quality

In Table 5.3 the post-route results are listed for a subset of the benchmark designs that were able to route in the reasonable timespan of 5 hours. The designs are routed with the default settings of the router in VPR 7.07. The post-route geomean ratios are very similar to the post-placement quality ratios in Table 5.2. This indicates that the placement cost estimation is good and makes the conclusions based on placement quality valid. The wirelength decrease achieved by LIQUID is slightly more pronounced with 8% compared with AP both in fast modus and 13% both in high quality modus. For the critical path delay the gap between LIQUID and AP increases. The post-route solutions placed with *Liquid-HQ* have a 9% better critical path delay than the solutions placed by *AP- HQ*. The critical path delay gap of LIQUID compared with *SA-HQ* reduces slightly to 10% with *Liquid-fast* and 4% with *Liquid-HQ*.

5.8 Future Work

There are several interesting future research paths around improving LIQUID:

- Investigating the use of nonlinear functions for the timing cost. In analytical placement the system has to be solved, so you are tied to a linear cost function, but for a steepest gradient descent method there is much more freedom. The only condition for the function is that it is derivable.
- In our current implementation we gave each block the same mass in our accelerated gradient descent simulation, but we could differ the mass for each block type.
- LIQUID calculates the move vectors of the blocks only based on the location of the attached blocks in the previous iteration. It could be improved by also taking into account the momentum of the attached blocks in the previous iteration.

- Our legalization method could be improved by adding the advanced architecture aware techniques described in [27].
- All movement vectors in LIQUID can be calculated independently, which makes it uniquely suitable for parallelisation, so a straightforward path is adapting LIQUID for GPU acceleration.

5.9 Conclusion

The main conclusion of this chapter is that it is not necessary to solve the linear system built in conventional analytical placement techniques. It is sufficient to optimize the placement in between legalization phases by following the steepest gradient descent with a momentum simulation. We implemented this method in our new placer called LIQUID and achieved a speedup of 1.9 compared to our own conventional analytical placement implementation for the same quality. Another important result is that given more time LIQUID is able to find higher quality placements in comparison to our analytical placement implementation with a 9% decrease in critical path delay and 9% decrease in wire-length cost. We want to help the community by releasing our source code for both LIQUID and our analytical placer on GitHub [136].

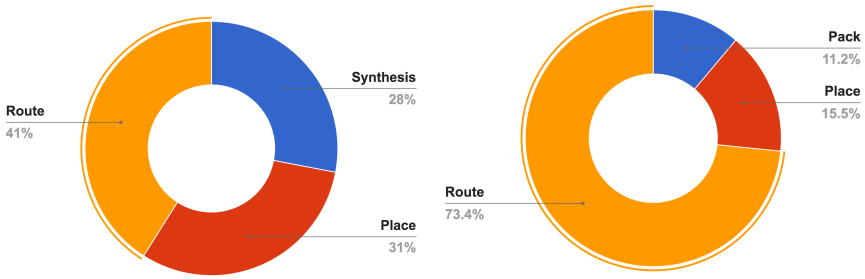
6 A Connection-based Routing Mechanism

In this chapter we introduce a new routing algorithm that is faster than the conventional routing strategies. In the introduction we discuss the significance of speeding up the routing process. We describe the conventional routing algorithm and how we developed a faster routing algorithm.

6.1 Introduction

In most cases the most time consuming step of the FPGA design flow is the routing step. In Chapter 3 we reported the runtimes for compiling the VTR benchmark designs with Xilinx' Vivado tool flow targeting an UltraScale device. Vivado spent 28% in synthesis and technology mapping, 31% in pack and placement and 41% in routing. Clearly the largest runtime consumer is the routing process. However, the VTR benchmark suite contains relatively small benchmarks. In case we consider larger benchmarks, the routing runtime grows more relative to the other tool runtimes. In [106] Murray et al. report that VPR needs on average 26 min to pack, 36 min to place and a massive 171 min to route the Titan benchmark suite. The runtimes for the different tools in the compilation flow are depicted in Figure 6.1. The routing runtime is clearly the most problematic and will only worsen if FPGA devices and designs further continue to increase in size. Therefore, one of the main

goals of the work in this chapter is to improve the runtime scalability of the state-of-the-art routing algorithms.



(a) Vivado targeting the VTR benchmark suite

(b) VTR targeting the Titan benchmark suite

Figure 6.1: The runtime breakdown of the compilation runtime. The routing process is clearly the largest runtime consumer.

Traditionally PATHFINDER-based algorithms are used to route designs [100]. PATHFINDER routes a design in several iterations. In each iteration all the nets in the circuit are ripped up and rerouted. After each iteration the cost of overused routing resources is increased. PATHFINDER stops if a legal routing solution is found. A Pathfinder-based router is implemented in VPR [89]. The runtime breakdown in Fig. 6.2 of the VPR router shows that each routing iteration consumes about the same amount of runtime. A straightforward speedup method is to trade some quality for runtime by only ripping up and rerouting congested nets [51]. This leads to a 1.9x speedup when routing the VTR benchmark suite. A runtime inspection of the routing process with the speedup method reveals a new clear culprit. Rerouting high fanout nets takes up the majority of the runtime, see Figure 6.3. Large fanout nets typically require a lot of routing resources and they span a large area of the die and therefore they have more chance to be congested. In practice they almost always have to be rerouted. To further reduce the runtime, we propose a new routing mechanism in which only parts of the routing tree are rerouted. We implemented this mechanism in the Connection Router (CROUTE) . To achieve a speedup CROUTE reroutes a connection only if it's congested or if it's critical and the delay diverges too much from the minimum delay.

We described the connection router mechanism first in [133] and in this chapter a timing-driven version is described and we focus on its ability to reduce the routing runtime. The connection router mechanism is orthogonal to a lot of related work on speeding up routing, such as parallelisation. If we further analyse the runtime of the CROUTE, see

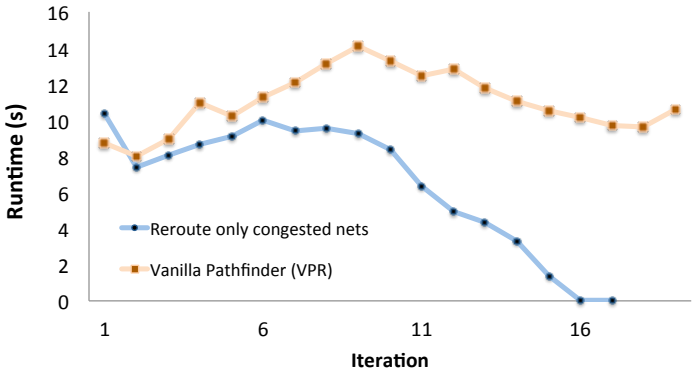


Figure 6.2: Runtime per iteration when routing *stereovision1* for the PATHFINDER-based router in VPR and for an adapted version that only reroutes congested nets.

Fig. 6.3, the breakdown shows that a large amount of the runtime is spent in the first routing iteration. Each connection needs to be routed at least once. Typically in the first iteration the router tries to reduce the path delay and is therefor congestion-oblivious. So nets can be routed independent of each other. No synchronization is required during the first iteration, only at the end of the first iteration. To further reduce the runtime parallelization techniques can be used that are described in related work [51, 52]. The negotiated congestion mechanism in itself is harder to parallelize because the router tends to have convergence problems and solutions tend to suffer quality loss if the occupation of the routing resources is not synchronized in time.

6.2 The Routing Resource Graph

The FPGA's routing infrastructure consists of a network of wires organized in channels. At the intersections of the channels, multiplexers interconnect the wires from different channels. Multiplexers also hook up the functional blocks to the routing network. They connect some of the wire tracks in the channel to the inputs and outputs of the functional block. The capacity of the routing infrastructure can be regulated by the number of tracks per channel. The routing infrastructure can be represented by a Routing Resource Graph (RRG), which is a directed graph, $C = (N, E)$, used as a model for the routing architecture of an FPGA. This graph can be constructed for any routing architecture. The

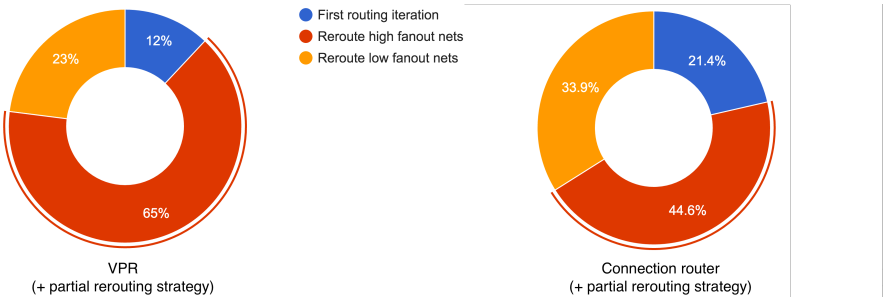


Figure 6.3: The average runtime breakdown for the VPR router and the connection router with partial rerouting strategy for a subset of the Titan23 benchmark designs.

nodes N represent wires. A directed edge (t, h) represents the possibility of routing a signal from wire t (the tail of the edge) to wire h (the head of the edge), by setting a switch. For example, when a unidirectional switch is closed, the logic value of resource i is forced on resource o, this is modeled as a directed edge (i, o) in the RRG.

6.3 The Routing Problem

After placement, each of the functional blocks in the input circuit is assigned to a physical block on the FPGA. The router has to determine which of the switches in the routing architecture need to be closed in order to connect the physical blocks in accordance with the way their associated functional blocks are connected in the mapped circuit. The interconnections between functional blocks are conventionally defined as a list of nets. Each net is an interconnection between one source and one or more sinks. The netlist can also be easily defined as a list of source-sink connections.

When the routing architecture of the FPGA is represented as a routing-resource graph, the routing problem reduces to finding a subgraph of the routing resource graph, called a routing tree, for each net in the input circuit. The routing trees should be disjoint in order to avoid short circuits. Each routing tree should contain at least the source node and sink nodes of its associated net and enough wire nodes so that source and sink nodes are connected.

This problem definition can also be expressed in terms of connections, in which a connection is an ordered pair containing a source and

a sink. Each net can be seen as a set of connections and thus all interconnection between logic blocks can be defined as a set of connections. The routing problem then reduces to finding a simple path in the routing-resource graph for each connection in the circuit. Each path starts at the source node and ends at the sink node of its associated connection. These paths should only share nodes if the corresponding connections have the same source. Allowing other connections to share nodes would lead to short circuits.

Listing 6.1: Pseudo code for PATHFINDER

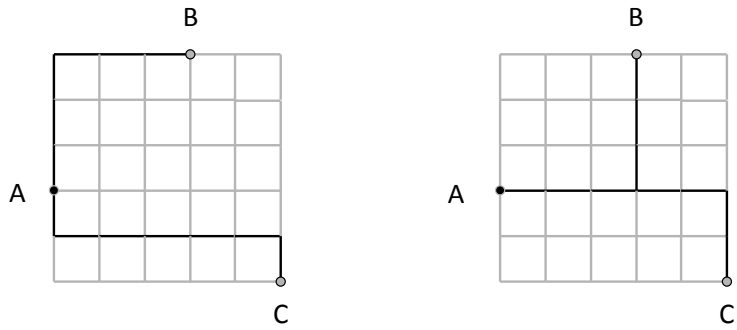
```
1  while (IllegalSharedResourcesExist()):
2      for each Net n do:
3          ripUpRouting(n)
4          route(n)
5          n.resources().updatePresentCongestionCost()
6      allResources().updateHistoryCost()
7      updatePresentCongestionMultiplier()
8      allResources().updatePresentCongestionCost()
```

6.3.1 PATHFINDER: A Negotiated Congestion Mechanism

The main structure of PATHFINDER is the negotiated congestion loop [100]. The pseudo-code is shown in Listing 6.1. In every routing iteration, the algorithm rips up and reroutes all the nets in the circuit. These iterations are repeated until no resources are shared illegally. This is achieved by gradually increasing the cost of illegally shared resources. During the first iteration, nets can illegally share resources at little extra cost. In the subsequent iterations, the cost of a routing resource does not only depend on the current sharing cost but also on the sharing history of the resource. Resources that were illegally shared in past routing iterations become more expensive. In this way a congestion map is built, which enables the router to avoid routing through heavily congested areas, if possible.

Listing 6.2: Pseudo code of the maze router implemented in VPR routers as a net router heuristic.

```
1  function route(Net n):
2      routingTree = {source}
3      for each Sink s of n:
4          path = dijkstra(routingTree, s)
5          routingTree = routingTree∪path
```



(a) A possible solution from Lee's maze router (#wires=13)

(b) One of the optimal Steiner tree solutions (#wires=10)

Figure 6.4: An example of the Lee's maze router producing a suboptimal solution for a net with three terminals

To route one net, the embedded net router tries to find a minimum cost routing tree in the RRG. This problem is called the minimum steiner tree problem and is NP-complete [65]. To find a solution in a reasonable time a heuristic is needed. In Listing 6.2 the pseudocode is shown for a variant of Lee's maze router [78] as it is implemented in VPR. For each sink of the net, the heuristic extends the already found routing tree with the shortest path from the routing tree to the sink under consideration with Dijkstra's shortest path algorithm [42]. This net router heuristic is fast but does not always come up with an optimal solution, certainly in case of nets with a high fanout. It forces the router to extend the routing tree alongside the paths that were previously added to the routing tree. As an example we show a suboptimal solution produced by the maze router alongside an optimal Steiner tree solution in Figure 6.4. If the connection A-B is routed first and Dijkstra finds the path depicted in Figure 6.4 (a), then the resulting routing tree will not have any shared wires and will be suboptimal. To overcome this problem the Connection router has a negotiated sharing mechanism which is explained in Section 6.5.

Once the nets are routed some of the wires are used by multiple nets. To solve this congestion, a negotiated congestion mechanism is used, which is the main contribution of [100]. To achieve a negotiated congestion mechanism the cost of the node is modulated with congestion penalties and timing factors. The following equation describes how the cost of a node is calculated. In what follows the equation is broken down and each factor and penalty explained.

$$c(n) = c_{prev} + (1 - f_{crit}) \cdot b(n) \cdot h(n) \cdot p(n) + f_{crit} \cdot T_{del} + \alpha \cdot c_{exp}, \qquad (6.1)$$

The cost of the node is a weighted sum of the congestion cost and the timing cost. The weights are determined by the criticality factor. The criticality factor denotes how critical a connection is.

$$f_{crit} = min\left(1 - \frac{slack}{T_{req,max}}, f_{crit,max}\right), \tag{6.2}$$

The criticality factor is zero in the first routing iteration and is updated after each iteration following a timing analysis and according to Equation (6.2). The criticality factor is capped at $f_{crit,max}$ to prevent deadlock in case a congested wire is occupied by several critical connections. Typically $f_{crit,max}$ is around 0.99. $T_{req,max}$ is the maximum of all required times and it is used to normalize the slack. Together with the slack it is calculated during the traversals of the timing graph during the timing analysis. The slack is calculated from the net delay, the arrival time acquired during a forward traversal and the required time acquired during a backward traversal.

$$slack = T_{req} - T_{arr} - T_{del} \tag{6.3}$$

The congestion cost in Equation (6.1) is the product of $b(n)$, the base cost of the node, $p(n)$, the present congestion penalty, and $h(n)$, the historical congestion penalty. If a net is rerouted the present congestion penalty is updated as follows,

$$p(n) = \begin{cases} 1 & \text{if } cap(n) > occ(n) \\ 1 + p_f \cdot & \\ (occ(n) - cap(n) + 1) & \text{otherwise} \end{cases}, \tag{6.4}$$

where $cap(n)$ represents the capacity of the node and $occ(n)$ is the occupancy of the node. The occupancy of a node is the number of nets that are presently using the node. The factor p_f is used to increase the illegal sharing cost as the algorithm progresses. After every routing iteration, i, the historical congestion penalty is updated as follows,

$$h^i(n) = \begin{cases} 1 & \text{if } i = 1 \\ h^{(i-1)}(n) & \text{if } cap(n) \geq occ(n) \\ h^{(i-1)}(n) + & \\ h_f(occ(n) - cap(n)) & \text{otherwise} \end{cases}. \tag{6.5}$$

Again, the factor h_f is used to control the impact of the historical congestion penalty on the total resource cost. The way the congestion factors p_f and h_f change as the algorithm progresses is called the routing schedule. In the connection router the same routing schedule is used as in the VPR router.

The location of the sink node, found in the placement of the circuit, is used to direct the search and thereby reducing the number of nodes visited during the search. The nodes that lead to a least-cost path are

Listing 6.3: Pseudo code of the Connection Router

```
1  while (IllegalSharedResourcesExist()):
2      for each Connection c do:
3          ripUpRouting(c)
4          route(c)
5          c.resources().updatePresentCongestionCost()
6      allResources().updateHistoryCost()
7      updatePresentCongestionMultiplier()
8      allResources().updatePresentCongestionCost()
```

expanded first. Equation (6.1) is actually the path cost seen in the node under consideration. This results in a narrow wavefront that expands in the direction of the target pin to be connected. α is the direction factor, which determines how aggressively the router explores towards the target sink, c_{prev} the cost of the previous wire nodes on the path from the source to this wire node and c_{exp} the expected cost of the path from this wire node to the sink node. The expected cost heuristic is calculated based on the Manhattan distance, Δ_m, from current wire node to the sink node as follows,

$$\begin{aligned} c_{exp} = f_{crit} \cdot \left(\Delta_m \cdot t_{lin} + \Delta_m^2 \cdot t_{quad} + R_{up} \cdot \Delta_m \cdot C_{load}\right) + \\ (1 - f_{crit}) \cdot (\Delta_m \cdot b + b_{ipin} + b_{sink}) \end{aligned} \tag{6.6}$$

with t_{lin} and t_{quad} the linear respectively quadratic timing weights. R_{up} is the sum of all wire resistances of the wires upstream and C_{load} is the capacity per wire segment. b is the base cost of a wire segment, b_{ipin} and b_{sink} are the base cost of an input pin and a sink node respectively.

6.4 CROUTE: The Connection Router

In the connection router each connection in the circuit is ripped up and rerouted seperately. This seems straightforward to implement, but it requires changes in the node cost and the main congestion loop.

6.4.1 Ripping up and Rerouting Connections

To partially break up and rebuild the routing tree of a net, CROUTE rips up and reroutes connections in the main congestion loop instead of nets. The pseudocode is listed in Listing 6.3. There is no netrouter heuristic necessary in CROUTE. The connection router breaks up a connection, recalculates the occupation of the nodes in the old routing path

and updates the present congestion penalty according to Equation (6.4) if the occupation of a node decreases. Next, Dijkstra's algorithm is used to find a path. The occupation of each node in the path is recalculated. If the occupation of a node increases, then the present congestion penalty of that node is updated according to Equation (6.4).

The advantage of routing on a connection based routing mechanism is that the negotiated congestion mechanism is effectively applied on a finer scale compared to net based routing. First the routing trees of the nets are only partially broken down and rebuilt to solve congestion. The nodes in the path of one connection will be reconsidered when Dijkstra's algorithm is searching a path for another connection of the same net. This differs with the mechanism applied by the maze net heuristic, see Listing 6.2. Once a node is added to a routing tree of a net, the maze net router is forced to make use of the node, if it is along the way to the sink of another connection of the same net. Second, the present congestion penalties are updated along with adding the nodes to the routing path of a connection. In PATHFINDER, the present congestion penalties are only updated after the whole routing tree is found.

To make ripping up and rerouting connections possible, the node cost in Equation (6.1) has to be modified to take into account the nodes that will be shared between connections driven by the same source.

6.4.2 The Change in Node Cost

The cost of a node for a connection should be cheaper in case it is already being used by other connections of the same net, but it can't be zero either, because that would force the router to explore these nodes. The node cost model should optimise the total wirelength of a circuit. To derive the node cost for a certain connection, let's look at the definition of the total wirelength: The total wirelength is the sum of the base costs of the different nodes used in the routing solution. This total cost can be partitioned according to the nets in the circuit. The total wirelength is the sum of the cost of each net. In a legal routing solution, the nets are disjoint so no nodes can be shared between nets. So the cost of a net can be calculated as the sum of the cost of each node used in the net, see Equation (6.7).

$$\begin{aligned}\text{Total Wirelength} &= \sum_{Node\ n\ \in\ routing\ solution} b(n) \\ &= \sum_{Net\ k\ \in\ nets} \left(\sum_{n\ \in\ k} b(n) \right) \\ &= \sum_{Con\ c\ \in\ cons} \left(\sum_{n\ \in\ c} \frac{b(n)}{share(n)} \right)\end{aligned} \tag{6.7}$$

In the same way, it is also possible to partition the cost according to the source-sink connections in the circuit. However, in a legal solution the connections do not have to be disjoint. Connections can legally share routing nodes if they are driven by the same source. So if the total wirelength is partitioned by connections, the cost of a connection is the sum of the base costs of the nodes that realise the connections, but if a node is shared between a number of connections, $share(n)$, then the cost of a node has to be shared by all the connections using it.

In one iteration all the connections are ripped up and rerouted. Dijkstra's algorithm searches the lowest cost path between the sink and the source of the connection. The cost of a node is now calculated as follows

$$c(n) = c_{prev} + (1 - f_{crit}) \cdot \frac{b(n) \cdot h(n) \cdot p(n)}{1 + share(n)} + f_{crit} \cdot T_{del} + \alpha \cdot c_{exp}, \tag{6.8}$$

The most important difference with Equation (6.1) is that the cost of the node is now divided by $share(n)$, the number of connections that legally share the node with the connection that Dijkstra's algorithm is currently searching a path for. Note that the definition of $share(n)$ slightly differs from the definition used in equation 6.7.

The congestion penalties are calculated in the same way as in Equation (6.4) and (6.5), but the occupancy is now expressed in terms of connections, $occ(n)$, is now the number of different sources that drive the connections that use the node n.

To calculate share(n), the routing algorithm keeps a map for each net. The map contains the used nodes and for each node the number of connections that use the node. In the rip up method, the values for share(n) are loaded in into the node data structure and loaded out after a path is found. This implementation limits the extra data that has to be kept per node in the routing resource graph, and is therefore scalable with respect to the size of the FPGA.

$$\begin{aligned}c_{exp} = f_{crit} \cdot \left(\Delta_m \cdot t_{lin} + \Delta_m^2 \cdot t_{quad} + R_{up} \cdot \Delta_m \cdot C_{load}\right) + \\ (1 - f_{crit}) \cdot \left(\frac{\Delta_m \cdot b}{1 + share(n)} + b_{ipin} + b_{sink} \right)\end{aligned} \tag{6.9}$$

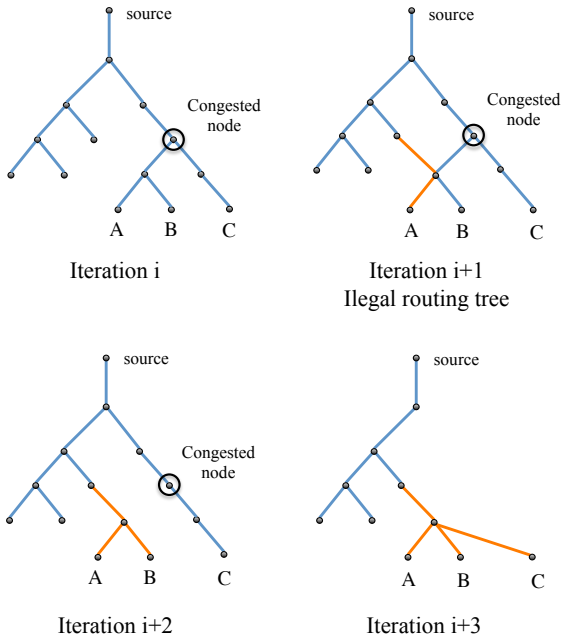
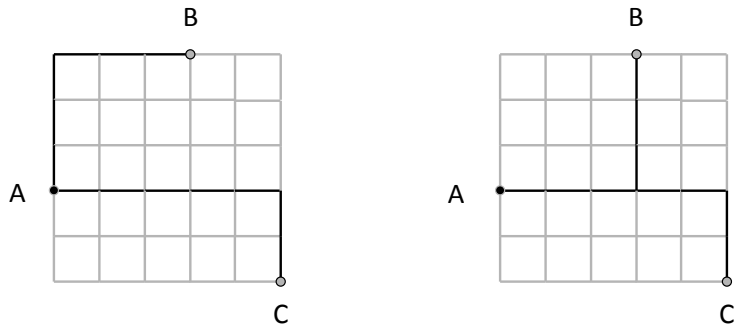


Figure 6.5: Intermediary illegal routing tree

The expected cost heuristic in the connection router needs to change in order to keep it admissible, as shown in Equation (6.9). The base cost of the segments has to be divided by the number of connections that use the node under consideration because it is possible that all the wire segments up to the input pin of a connection are shared with other connections that have the same source.

After routing each connection, the timing graph and the delay of each connection in the net is updated. In case a node is congested, the router will try to circumvent the congestion, but it is possible that the routing graph will be temporary illegal in-between iterations. An example is given in Figure 6.5. Connections with sink A, B, and C use a congested node in iteration i and the congestion mechanism is gradually solving the connection. In iteration i+1 the routing graph is not a tree. To take into account illegal routing graphs, the calculation of the net delays is modified. It always assumes the worst case scenario. In the first iterations, this will happen occasionally, but in almost all cases the number of occurrences drops to zero as the congestion dissolves. A minimum delay tree is extracted from the illegal graph in the rare case there are illegal routing graphs left after all the congestion is solved.



(a) After iteration 1 (b) After iteration 2

Figure 6.6: An example of the negotiated share mechanism which improves the routing tree over multiple iterations by encouraging routing resource reuse.

6.5 Negotiated Sharing Mechanism

In Figure 6.4 a suboptimal routing tree is depicted. The suboptimal routing tree is the result obtained by the maze router in VPR. The main cause why VPR generates a suboptimal routing tree in this case, is that there are a large number of equivalent shortest paths. For example connection A-B has 20 equivalent paths with a minimal cost of 6 wires. The router arbitrarily chooses one of these shortest paths. In the case of Figure 6.4 (a) this path does not allow any resource sharing for connection A-C. There are however other possibilities, that would allow more routing resource sharing and result in a routing tree closer to a minimum steiner tree, as depicted in Figure 6.4 (b)

This problem worsens if the manhattan distance between the source and the sink of a connection increases, because the number of equivalent shortest paths increases exponentially. To overcome this problem, two mechanisms are built in the connection router.

6.5.1 The Negotiated Sharing Mechanism Inherent to CROUTE

By ripping up a connection at a time the rest of the routing tree remains and influences the cost of the nodes via the $share(n)$ division in the calculation of the node cost. This effectively encourages sharing routing resources over multiple routing iterations. As an example we consider the situation depicted in Figure 6.6. After one iteration we have no resource sharing but if connection A-B is ripped up and rerouted the

shortest path will be alongside the routing path of connection A-C, because the cost of these wires will only be half the base cost.

6.5.2 Trunk Bias

The negotiated sharing mechanism only works if one of the other connections is routed on a part of one of the shortest paths. When the router is routing the first connection of a net with Dijkstra, it is clueless about the location of the rest of the net's sinks. In order to help the router with initially choosing a good path from the equivalent shortest paths we add a bias towards the geometric center of the net. The bias must have a smaller influence than the wire cost, because it is only meant to be a tie breaker. We add the following term to the node cost:

$$c_{bias} = \frac{b(n)\cdot}{2 \cdot fanout} \cdot \frac{\Delta_{m,c}}{HPWL} \tag{6.10}$$

The minimum wire cost is $b(n)/fanout$ in case a wire is shared by all of the connections. The bias cost will be maximally half of the minimum wire cost. The bias cost depends on the manhattan distance to the geometric center which is normalized against the Half Perimeter Wire Length (HWPL). As we close in on the geometric center the effect reduces.

During the negotiated congestion mechanism the cost of the nodes can only increase, so the effect of the bias cost becomes smaller towards the later iterations of the routing algorithm.

6.6 Partial Rerouting Strategies

To speed up the routing process the router could be programmed to only focus on the congested nets. Right before ripping up the net, the nodes of the old routing tree are checked and if no nodes are congested, the net is skipped. This method has an average speedup of 1.9x for the VTR benchmark suite, but the quality of the result decreases with a 25% increase in critical path delay. The cause of the critical path detoriation is that the routing solution converges too quickly. The router focuses too much on routability with not enough emphasis on optimizing critical path delay. In the first iteration the router routes the connections in the circuit congestion-oblivious. Each connection is routed with a minimum delay. To put more stress on timing, the most critical nets are rerouted if the delay of one of its connections is higher than the minimum delay obtained in the first iteration independent of their congestion status. Only nets with one of their pin criticalities above a tim-

ing criticality threshold, $f_{crit,cutoff}$ are rerouted. We experimented with different values for the criticality threshold and for a value of 0.9 the resulting critical path delays are on average the same as the PATHFINDER algorithm built in VPR.

In order to further reduce the runtime, the runtime profiling results in Fig. 6.3 show that the majority of time is spent by rerouting high-fanout nets. High-fanout nets have more chance to be rerouted because they typically require a lot of routing resources, span a larger area of the die and are more critical than their low-fanout counterparts. Partially rerouting the high-fanout nets would obviously reduce runtime and the connection-based routing mechanism allows us to only reroute the congested and the critical connections. Instead of rerouting an entire critical net, only the connections with a criticality above 0.9 are rerouted. Since the connection delays are updated after routing a connection, we can check if a critical connection is longer than its minimum after routing. Unfortunately the congestion status of the connection can be changed after routing the connection, so the router rechecks the old routing trace for congested nodes before it decides if the connection will be rerouted.

6.7 Experiments and Results

6.7.1 Methodology

To measure the performance of the connection router with partial rerouting we compare it to the PATHFINDER router implemented in VPR [88]. CROUTE is implemented in the VTR framework, which is mainly written in C. The Titan suite is used to benchmark the routers [106]. The target architecture is the Stratix-IV based architecture. The architecture has identical functional blocks as in the commercial Stratix-IV, but a simplified routing architecture with 87% wires that span four logic blocks and 13% wires that span 16 logic blocks. The wires are fully populated, branching of in each connection block and switch block they come across. The switch blocks have a Wilton topology. The Titan benchmark suite contains 23 small designs and 19 larger designs that are more challenging to place and route. To prevent overfitting the smaller designs were used to fine-tune the routers and search for optimal parameter values, such as f_{crit}. The larger designs are used for the final evaluation. The placements were generated with the VPR timing-driven placer with default settings. For each benchmark design the following properties of the routing implementation were measured:

- The **Runtime** is the time the router needs to find a solution for an FPGA architecture with a given channel width. It does not include the time to parse the netlist and generate the routing resource graph. The routing algorithms were executed on a workstation with an Intel quad-core i7-3770@3.40GHz and with 32GB 1600Mhz DDR3 memory.
- The **Maximum Clock Frequency** is determined by the slowest path delay, also called the critical path delay. In case there are more clock domains, the geomean of all maximum clock frequencies is used for comparison.
- The **Total Wire-length** is the number of wires needed to realise the design.

Table 6.1: Runtime, total wire-length and critical path delay for CROUTE and the baseline which is the routing algorithm implemented in VPR slightly adapted to only reroute the congested and critical nets.

	Runtime [s]			**Critical Path Delay [ns]**			**Total Wirelength [x1000]**		
Benchmark Design	baseline	Croute	ratio	baseline	Croute	ratio	baseline	Croute	ratio
cholesky_mc	289	215	0.74	9.3	9.3	1.00	1443	1265	0.88
stereo_vision	130	115	0.88	9.3	10.0	0.93	998	982	0.98
sparcT1_core	514	540	1.05	10.1	9.9	1.02	1674	1647	0.98
neuron	179	148	0.83	10.0	9.7	1.03	1275	1252	0.98
segmentation	4456	3475	0.78	894.2	897.3	1.00	3222	3152	0.98
stap_qrd	609	437	0.72	9.1	9.9	0.92	4112	3346	0.81
SLAM_spheric	3682	3539	0.96	84.8	84.8	1.00	3166	3159	1.00
mes_noc	4183	2273	0.54	16.8	16.6	1.01	10955	9678	0.88
des90	894	664	0.74	14.6	13.7	1.07	3522	3333	0.95
bitonic_mesh	3464	2286	0.66	16.0	16.9	0.95	7641	7440	0.97
cholesky_bdti	502	371	0.74	10.5	11.4	0.92	3575	3161	0.88
Geomean			0.77			0.99			0.92
Std. Dev.			0.13			0.05			0.06

6.7.2 Results

We slightly adapted the router in VPR to reroute only congested and critical nets, because then we can clearly show the advantage of connection based routing apart from the effect of partial rerouting. Partial rerouting strategies can be both applied to the net and connection based router. The modified VPR router will be used as a baseline to compare to CROUTE and we will refer to this router as the baseline router. The routers target the Stratix IV architecture with a fixed channel width of

300. In what follows the runtime to find a routing solution and quality of the solutions are compared. The results are summarized in Table 6.1. We only listed the benchmark designs that we were able to route in the reasonable timespan of 5 hours.

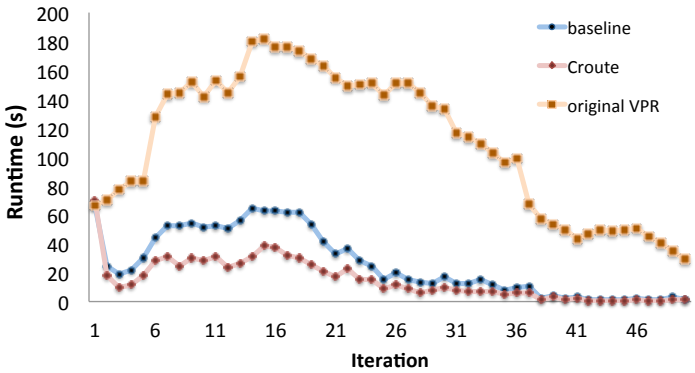


Figure 6.7: Iteration runtime for the LU32PEEng benchmark design

The connection router is able to find routing solutions on average 23% faster than the baseline router. Almost all the designs are routed faster with CROUTE. The main cause for the drop in runtime is the decrease in runtime for one routing iteration. The routing iteration runtime dramatically drops in the first few iterations for both routers, because the easier to solve congestion is solved and those parts are not rerouted anymore. For CROUTE this effect is even more prononounced. However, CROUTE typically needs slightly more iterations. The net effect is a reduction in runtime. To further elaborate the decrease in routing iteration runtime, we investigate the routing iteration runtime for the benchmark design *LU32PEEng* implemented on the *k6_frac_N10_mem32K_40nm* architecture in Figure 6.7. The channel width is set to 168, which is the minimal channel width for the default router in VPR, so there is a fair amount of congestion the routers need to solve to find a solution. We clearly see the initial drop in runtime both for the baseline and CROUTE, but for CROUTE the drop is more pronounced. The runtime for one iteration for CROUTE typically stays lower than an iteration for the baseline router. To highlight the difference between a router with and without a partial rerouting strategy, we also included the results for the original VPR router. The runtime clearly is much higher without partial rerouting strategy and this is more extreme compared to CROUTE, mainly because the speedup from partial rerouting is larger when applied on connections, because it is a

finer scale.

The quality of CROUTE solutions in terms of critical path delay is similar as the baseline routing solutions with an average 1% reduction in critical path delay, but with a standard deviation of 5%. The total wire-length is consistently better. The total wire-length for each benchmark design is lower with an average decrease of 8%, albeit with some variation with a standard deviation of 6%. This improvement is attributed to CROUTE's negotiated sharing mechanism which is described in section 6.5. The mechanism improves the routing tree of a net over multiple iterations.

6.8 Conclusion and Future Work

In this chapter we proposed a new routing algorithm called the connection router, CROUTE. It is a router which applies a more congestion oriented rerouting instead of simply rerouting the whole circuit. CROUTE rips up and reroutes parts of the routing tree of a net, by ripping up and rerouting connections instead of nets in the main congestion loop CROUTE is more runtime efficient than the PATHFINDER-based router implemented in VPR. Given an FPGA with a certain channel width, CROUTE is able to find routing solutions 23% faster, the quality of the routing solutions is slightly better in terms of total wirelength with an average decrease of 6%. The solutions have a similar critical path delay.

The techniques applied in CROUTE are largely orthogonal to many advanced routing runtime reduction techniques described in the literature. They can be applied in tandem with CROUTE, for example selective routing resource graph expansion [104] and parallel routing [51].

Nevertheless routing remains an important bottleneck in the design cycle.

7

Place and Route tools for the Dynamic Reconfiguration of the Routing Network

In this chapter we describe the placement and routing tools that we developed for the special FPGA technique that exploits the reconfigurability of the routing switches in the FPGA. The goal is to improve the circuit properties by dynamically reconfiguring the routing switches. We first start with giving an overview of dynamic partial reconfiguration and clarifying the contributions we made. Subsequently we describe the modifications made to the conventional placement and routing approaches to enable the dynamic reconfiguration of the FPGA's routing network.

7.1 Overview of Dynamic Partial Reconfiguration

Traditionally, a configuration of a Field Programmable Gate Array (FPGA) is loaded at the end of the design phase, and it remains the same throughout the runtime of the application. In order to change the configuration, the computation is stopped, the new configuration is written to the configuration memory and then the computation can start again. The high write speed of SRAM memory cells in SRAM-based FPGAs opened up the possibility to reconfigure the FPGA between different stages of computation. In 1996 Xilinx released one of

the first dynamically and partially reconfigurable FPGAs, the XC6000 [140]. Dynamic Partial Reconfiguration (DPR) is a technique that allows reconfiguring only a portion of the FPGA at runtime, while the rest of the FPGA remains active. In order to change the functionality implemented by certain FPGA resources, the configuration memory bits that control those resources can be updated by sending a new partial configuration to the FPGA. In 2010 Altera, Xilinx's main competitor, also introduced an FPGA that supported DPR, the Stratix V (28 nm) [11].

DPR can be used to build self-reconfiguring systems that write a configuration to the FPGA's configuration memory, specialized for the application specifics at hand. Specialized configurations use fewer resources, because resources are reused, and therefore less FPGA resources are occupied. Thus, this technique allows larger circuits to be implemented on smaller and cheaper FPGAs. Additional advantages are that specialized configurations have a reduced logic depth and require less power than their generic counterparts. However, the downside is that the gain in efficiency can be diminished by the specialization overhead, because when the problem at hand changes, a new specialized configuration is needed. The specialization overhead is the extra FPGA resources and the extra time needed to obtain a configuration update and write the FPGA's configuration memory using DPR. Furthermore, the complexity of implementing a design using DPR and the lack of automatic tools hinder the widespread use of the DPR technique.

7.1.1 Introduction to Dynamic Circuit Specialization

In [21] these problems are addressed by using the Dynamic Circuit Specialization (DCS) technique. The proposed two-stage tool flow for DCS starts from an HDL (Hardware Description Languange) description in which the slowly varying signals, called parameters, are annotated. The parameters represent the application specifics at the current moment. Infrequently varying input signals are good choices to consider as parameter candidates. We don't want the system to calculate configuration updates frequently.

A computational intensive offline stage calculates a parameterized configuration. This is an FPGA configuration in which the bits are Boolean functions of the parameters. In the second, online, stage the parameterized configuration can be quickly evaluated to a configuration update. The configuration update can be applied using DPR and after the update the FPGA is running the new specialized configuration. This rapid evaluation leads to a low specialization overhead. Since the technique uses an annotated HDL file as input, it enables designers to

benefit from DPR without much extra design effort. The only requirement to use this tool flow is an FPGA that is capable of dynamic partial reconfiguration.

The tool flow introduced in [21], is called the TLUT tool flow and is available online [55]. It was the first tool flow developed for DCS. The offline stage is quite similar to a conventional FPGA tool flow. Only the mapping stage differs greatly. However, the TLUT tool flow is limited because it is only capable of dynamically specialising the values of the LookUp Tables (LUTs) present in FPGAs. The TCON tool flow, described in section 7.3, extends the TLUT tool flow with the possibility to dynamically specialize the interconnection network of an FPGA. This specialization has a lot of potential, because reusing routing resources could drastically reduce the area usage, delay and power consumption. For the test cases in this chapter, a Clos switch network and a Virtual Coarse Grain Array, the TCON tool flow generates implementations that need 50% up to 92% less LUTs and 36% up to 81% less wiring than conventional implementations. The logic depth decreases with 63% up to 80% in comparison with their corresponding conventional implementations.

7.1.2 Contributions

To implement the TCON tool flow, major changes were needed, not only in the mapping, but also in the pack, place and route steps. In [57], Heyse and Bruneel proposed an altered technology mapping algorithm. We introduced a pack and placement algorithm, called TPACK and TPLACE. TPACK and TPLACE are described in [130]. We also introduced the routing algorithms for the TCON tool flow. In [131], [132] and [130] routing algorithms, enhancements and extensions are described, leading to a honed routing algorithm, called TROUTE.

This work was done in collaboration with Karel Bruneel and Brahim Al Farisi. At the time, Karel Bruneel was a post-doctoral researcher and Brahim Al Farisi was a Phd. student in our research group.

7.2 Background

7.2.1 Configuration Swapping

In conventional dynamic reconfiguration tool flows runtime reconfiguration is used to have multiple processes timeshare a set of FPGA resources. The functionality of a part of an FPGA is swapped with another pre-computed configuration that is loaded from a storage

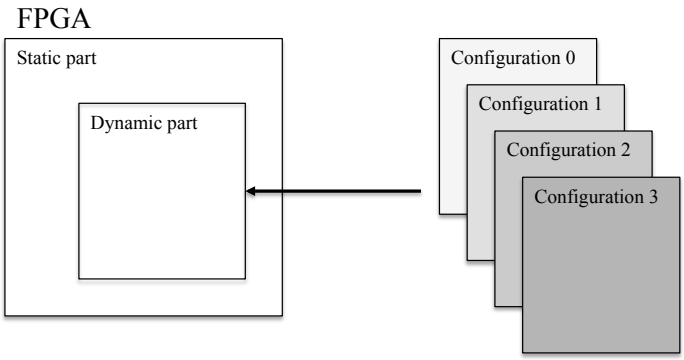


Figure 7.1: Conventional Dynamic Reconfiguration: Configuration swapping, different modules are loaded in

medium. This method is called configuration swapping or modular reconfiguration. The concept is illustrated in Figure 7.1. Tool support for configuration swapping can simply be realized by slightly adapting a conventional tool flow. FPGA manufacturers have adapted their tool flows for this type of reconfiguration. Xilinx has adapted its modular design flow [143] and a similar tool flow is available for Altera [12]. The commercial configuration swapping tool flows are complex and require a skilled designer. The designer has to prepare one or more parts of the FPGA for dynamic reconfiguration and he will need to describe all the functionalities that might be implemented using an HDL. Preparing a dynamically reconfigurable area encompasses:

- selecting the FPGA resources that are part of that area;
- defining a fixed interface for this area.

In the commercial tool flows the resources are selected by defining a rectangular area on the FPGA. The area must be large enough so that the worst case functionality in terms of resource usage fits into it. The fixed interface is important because whichever functionality is loaded in the area, it needs to plug into the static part present on the FPGA in exactly the same way so that the FPGA's configuration does not get corrupted.

Next, the tool flow needs to generate the partial configurations for each of the functionalities. All the functionalities that will be implemented by the same area need to have the same interface and this interface must be mapped on the interface of the associated area. The conventional tool flow is then run for each of the functionalities, with the

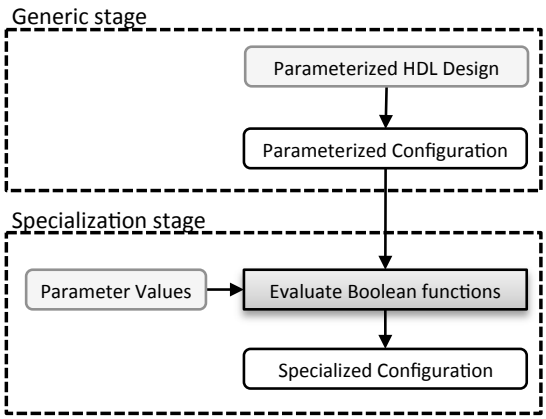


Figure 7.2: The two-staged tool flow

exception that the placement and routing tools which are constrained to the defined area and that the signals of the functionality's interface are forced to hook up to the area's interface.

Configuration swapping is only applicable to applications that have distinct subtasks that are not to be executed simultaneously. The method is also restricted in the number of different tasks that can be time multiplexed. Every task must be defined in a different HDL file, ran by the conventional tool flow and needs to be stored on a storage medium.

7.2.2 Dynamic Circuit Specialization

As mentioned in the introduction the specialization overhead can negate the benefits of a self-reconfiguring system. To keep the specialization overhead low, new specialized configurations need to be up and running as quickly as possible, but this is not trivial. For example, the conventional tool flow can be used to create specialized configurations at runtime. This would in many cases take minutes to hours, because it encompasses computationally hard problems, such as placement and routing. Another solution could be computing all specialized configurations in advance and storing them in a database, but this solution quickly becomes infeasible because of the immense number of possible configurations. E.g. 2^{32} or ca. 4 billion configurations are needed for a 32-bit multiplier with an infrequently varying coefficient.

To solve this issue the authors in [21] introduced a two-staged method, called Dynamic Circuit Specialization, to generate specialized configurations. A simple diagram can be seen in Figure 7.2. The first

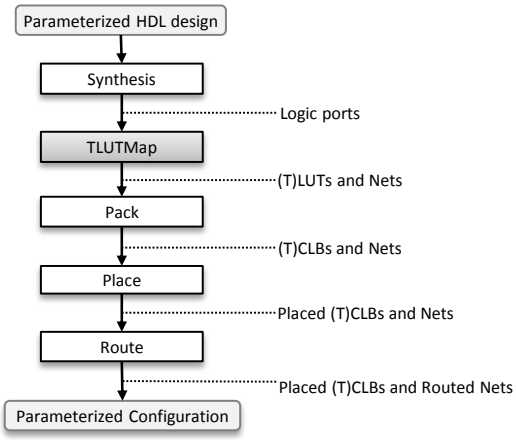


Figure 7.3: Generic stage of the TLUT Tool flow

stage, called the generic stage, starts from a parameterized HDL design. This is a regular HDL design in which some of the input signals are annotated as parameters. From this a *parameterized configuration* (PConf) is generated. A PConf is a multi-valued Boolean function that expresses a specialized FPGA configuration bitstream as a function of the parameters. At runtime this Boolean function can be evaluated using a specific set of parameter values to obtain a regular FPGA configuration (during the specialization stage). So it is possible to derive specialized configurations with different properties and/or functionality from a single PConf during runtime.

The advantage of generating specialized configurations from a PConf, instead of directly running the conventional FPGA tool flow, is the much lower overhead per specialized configuration. Generating a PConf costs about the same amount of time as generating a regular configuration, but the evaluation of Boolean functions is several orders of magnitude faster. An additional advantage is that only one configuration needs to be stored while the different specialized configurations can be generated in milliseconds [21]. For example, a PConf for an adaptive FIR filter, generated by the TLUT tool flow, described in section 7.2.3 can be evaluated to a specialized FIR configuration (8-bit input, 8-bit coefficients and 32 taps) in only 1.3 ms on a PowerPC 405 (PPC405) clocked at 300 MHz (available in the Xilinx Virtex-II pro FPGAs), while the conventional method needs several minutes to produce a specialized configuration on a standard desktop PC.

7.2.3 TLUT Tool Flow

The tool flow, that generates the PConf in the generic stage, consists of adapted versions of the tools found in the conventional tool flow. It takes a parameterized HDL design as input and outputs a PConf. A first tool flow for DCS, called the TLUT tool flow, is described in [21]. In figure 7.3 an overview of the different steps of this tool flow is given. In the first step of the tool flow, the parameterized HDL design is synthesized in a conventional way. Next, the design is mapped with a modified technology mapper, called TLUTMAP. A conventional mapper maps the design to a LUT circuit, while TLUTMAP maps the design to a Tunable LUT circuit. A Tunable LUT (TLUT) is a parameterized abstraction of a LUT. The truth table bits are expressed as functions of parameter inputs. Since parts of the design functionality depending on the parameters are incorporated in the parameterized truth table bits of the TLUTs, the size of the TLUT circuit is much smaller than the regular LUT circuit for the same design [21]. The next step is to pack, place and route the TLUT circuit. These steps can be performed by the conventional pack, place and routing tools, because the packing, placement and routing is independent of the content of the LUTs.

A PConf, produced by the TLUT method, only expresses the truth tables of the LUTs as a function of the parameters. All the routing between the LUTs is fixed. This leads to high quality specialized configurations. However, it has been shown in [130] that also expressing the routing configuration as a function of the parameters (TCON tool flow) leads to specialized configurations with an even better quality. For example, a 256×256 Clos switch, implemented as described in section 7.7, requires 1792 LUTs when the TLUT method is used, while only 768 LUTs are needed when the TCON method is used.

7.3 The TCON tool flow

The TCON tool flow produces PConf's following the same principle concept as the TLUT tool flow. It also starts from an RT level HDL description and generates a PConf. However, to allow dynamic specialization of the FPGA's interconnection network, changes are also required in the technology mapping, packing, placement and routing steps of the tool flow. In Figure 7.4 an overview of the different steps of the generic stage of the TCON tool flow are depicted. The changes needed in each step are described in the following sections. A simple 2×2 crossbar is used as an example.

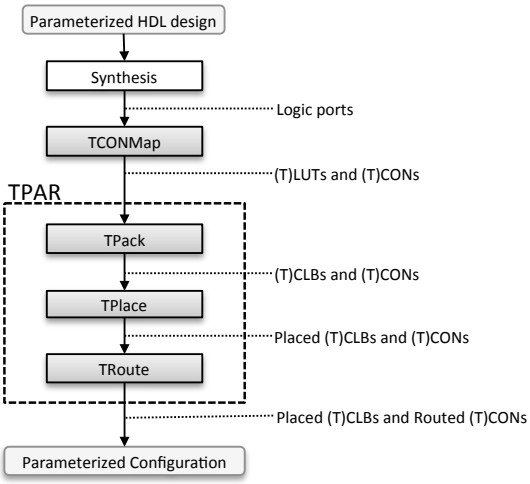


Figure 7.4: Generic stage of the TCON Tool flow

7.3.1 Synthesis

The synthesis step converts a HDL description in which some inputs are annotated as parameters into a parameterized Boolean network. In Listing 7.1 you can find the VHDL description of the 2×2 crossbar switch example. No significant changes need to be made to this step because parameters can be synthesized just like regular inputs. Except that parameter inputs in the HDL description do have to be annotated as parameter inputs in the resulting Boolean network as well. Designers only have to add a comment before and after the parameters in the description, as can be seen on line 3 and 5 of the VHDL code of the crossbar switch in Listing 7.1.

7.3.2 Technology Mapping

During technology mapping, the parameterized Boolean network generated by the synthesis step is not mapped onto the resource primitives available in the target FPGA architecture, but on abstract primitives that represent parameterized versions of these resource primitives:

- a **Tuneable LookUp Table (TLUT)**: a LookUp Table with the truth table expressed in terms of parameters, $\tau(p) : \mathcal{B}^k \rightarrow \mathcal{B}^{2^n}$, with k the number of parameters and n the number of inputs LUTs have on the target FPGA. A TLUT will be implemented by a regular LUT and the dynamic reconfiguration of its truth table.

Listing 7.1: The VHDL code for the 2×2 crossbar switch example.

```
1  entity crossbar is
2  port(
3    -- param
4    sel : in  std_logic_vector(1 downto 0);
5    -- param
6    i  : in  std_logic_vector(1 downto 0);
7    o : out std_logic_vector(1 downto 0);
8  );
9  end crossbar;
10
11 architecture behavior of crossbar is
12 begin
13   o(0) <= i(to_integer(sel(0)));
14   o(1) <= i(to_integer(sel(1)));
15 end behavior;
```

- a **Tuneable Connection (TCON)**: a connection with a connection condition expressed in terms of parameters. A TCON will be implemented by a set of wires and switches, and the dynamic reconfiguration of some of the switches in the set. A schematic of a TCON can be found in Figure 7.6.

Analogously to a regular connection, a TCON has a source and a sink. Additionally, every TCON has a connection condition $\zeta(p) : \mathcal{B}^k \rightarrow \mathcal{B}$, with k the number of parameters. This is a Boolean function of the parameters that returns true when the design requires the connection to be active. The connection conditions of the TCONs reflect the fact that not all connections are needed at the same time. They allow us to distinguish which connections are mutually exclusive in time. These connections will be allowed to share FPGA routing resources. TCONs are abstract concepts generated in the technology mapping step. These are refined to concrete FPGA resources during packing, placement and routing. After placement the endpoints of the TCONs are fixed and the router further refines the TCON by reserving the switches and wires to realize the connection.

The abstract concepts, TCON and TLUT, also cover static connections and static LUTs. A static connection is a TCON with a connection condition equal to 1 and a static LUT is a TLUT with a truth table independent of the parameter inputs.

In [57] TCONMap is presented. It is a technology mapping algorithm, able to exploit both the reconfigurable properties of the LUTs and the interconnect network of the FPGA. For more details on the TCONMap algorithm, we refer to the Phd. thesis of Karel Heyse [56].

Listing 7.2: Netlist for the 2×2 crossbar mapped by a conventional technology mapper

```
1  .input i0
2  pinlist: n0
3  .input i1
4  pinlist: n1
5  .input sel0
6  pinlist: n2
7  .input sel1
8  pinlist: n3
9
10 .output o0
11 pinlist: n4
12 .output o1
13 pinlist: n5
14
15 .clb clb_a
16 pinlist: n0 n1 n2 open n4 open
17 subblock: n4 0 1 2 3 4 open
18 .clb clb_b
19 pinlist: n0 n1 n3 open n5 open
20 subblock: n5 0 1 2 3 4 open
```

TCONMap produces a tunable circuit. Tuneable circuits contain TLUTs and TCONs. A schematic of the 2×2 crossbar switch and the TCON netlist, generated after TCONMap can be seen in Figure 7.7. TCONMap succeeds at mapping the functionality of the 2×2 crossbar on four TCONs and four Input/Output Blocks (IOBs). As a reference, conventional technology mapping needs six nets, two CLBs and six IOBs, see the netlist generated by the conventional technology mapping in Listing 7.2.

7.3.3 TPACK and TPLACE

In the packing step, LUTs and FFs are packed into CLBs. Subsequently the placer chooses a physical CLB on the FPGA for every CLB instance in the circuit. To optimise routability and interconnect delay, wirelength-driven placers use wire length estimates of the interconnections between CLBs. Conventional placers have to take into account that a connection can share wires and switches starting at source end of the connection. However TROUTE is not only able to let TCONs share resources at the source end, but also at the sink end, see Figure 7.8. To better estimate routing resource usage of a placement, new wire lenght estimation methods for TCONs are introduced for TPLACE, in

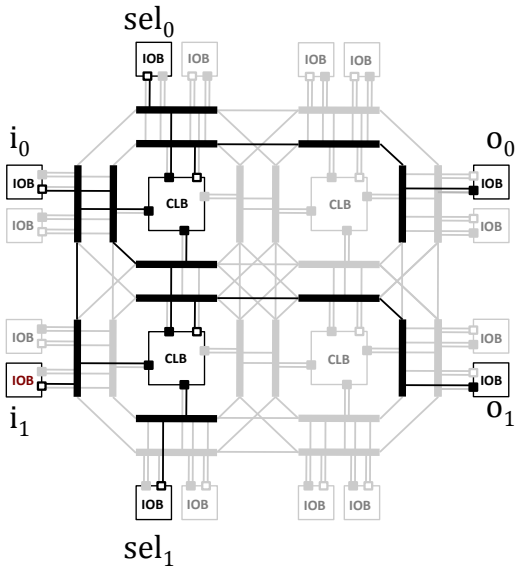


Figure 7.5: The FPGA circuit for the 2×2 crossbar switch example, compiled by the conventional tool flow. The target FPGA has 2×2 CLBs. The thick lines in the schematic represent the wires of the FPGA, the thin lines the edges. The wires and edges that are used, are accentuated in black. Output pins are open boxes and input pins are filled boxes.

section 7.5.

7.3.4 TROUTE

In the routing step routing resources are assigned to the TCONs. Important changes have to be made to the conventional routing algorithms, because the routing problem changes significantly. In the conventional case the router needs to route a set of nets. Each net is an interconnection between a source and one or more sinks. Nets may not share resources, otherwise this would lead to shorts. In the TCON tool flow TROUTE has to route a set of TCONs. TCONs have more than one resource sharing possibility. TCONs may share resources if they have the same source (because they carry the same signal) or if they are not active at the same time. Two TCONs, t_1 and t_2, with their connection conditions ζ_1 and ζ_2 respectively, are not active at the same time, if ζ_1 and ζ_2 are not simultaneously true for every parameter value:

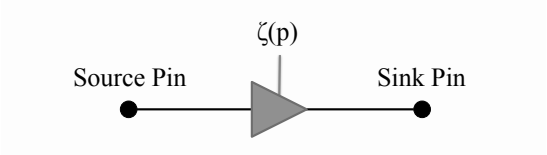


Figure 7.6: Schematic of a Tuneable Connection (TCON)

Listing 7.3: The TCON netlist for the 2×2 crossbar switch functionality

```
1  .parameter sel_0
2  .parameter sel_1
3
4  .input i0
5  pinlist : i0p
6  .input i1
7  pinlist : i1p
8  .input o0
9  pinlist : o0p
10
11 .output o1
12 pinlist : o1p
13
14 .tcon i0p o0p ¬sel_0
15 .tcon i0p o1p ¬sel_1
16 .tcon i1p o0p sel_0
17 .tcon i1p o1p sel_1
```

t_1 en t_2 not active simultaneously

$$\iff \forall p \in P : \neg(\zeta_1(p) \wedge \zeta_2(p)) \quad (7.1)$$

This is an important property that can be used to minimize routing resource usage. In section 7.6 the connection router, first presented in [132] and [131] is discussed, it is capable of routing TCONs while stimulating both resource sharing possibilities.

In Figure 7.8, two possible implementations are given for the 2×2 crossbar switch example on a FPGA with an array of 2 by 2 CLBs and a very basic routing architecture. The first routing solution exploits the first resource sharing possibility, TCONs driven by the same source may share routing resources. The second routing solution exploits the second sharing possibility, TCONs not active at the same time may share resources. Both routing solutions contain 8 wires and 14 switches and the same number of switches need to be reconfigured when the value of the control signals of the crossbar change. In this example

sel_0	sel_1	o_0	o_1
0	0	i_0	i_0
0	1	i_0	i_1
1	0	i_1	i_0
1	1	i_1	i_1

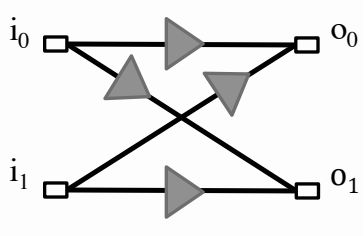


Figure 7.7: A schematic and truth table of a 2×2 crossbar switch functionality represented by 4 TCONs

there is no real advantage by choosing one of the two sharing mechanisms. However, practically, one sharing possibility may outperform the other. Choosing the most economic solutions reduces the required wiring resources.

7.3.5 Limitations

In academic tool flows, such as VTR, multipliers, block RAMs and other hard blocks are recognised at the synthesis level. These hard blocks can not be parameterized, in contrast to LUTs. However, if two hard blocks of the same type are not used at the same time, the TCON tool flow could be used to reuse one hard block and thereby reducing the area used. At this stage, the TCON tool flow does not support heterogeneous FPGAs. For applications that benefit from hard blocks, such as DSP applications, this could lead to considerable area gains.

In the following sections we describe in more detail the packing, placement and routing algorithms that have to be changed to enable compiling to configurations with the parameterized routing bits.

7.4 TPACK

TPACK packs the LUTs and FFs of the circuit into Configurable Logic Blocks (CLBs) (also called Adaptive Logic Modules in Altera devices). TPACK is similar to conventional packing, more details on conventional packing can be found in [17]. Some alterations are made to be able to deal with tuneable circuits. In this section, a CLB architecture with a two level hierarchy is assumed. The first level is a basic logic element (BLE), it contains a LUT and a FF. The second level consists of Configurable Logic Blocks (CLBs) containing several BLEs and fast internal routing resources to interconnect them.

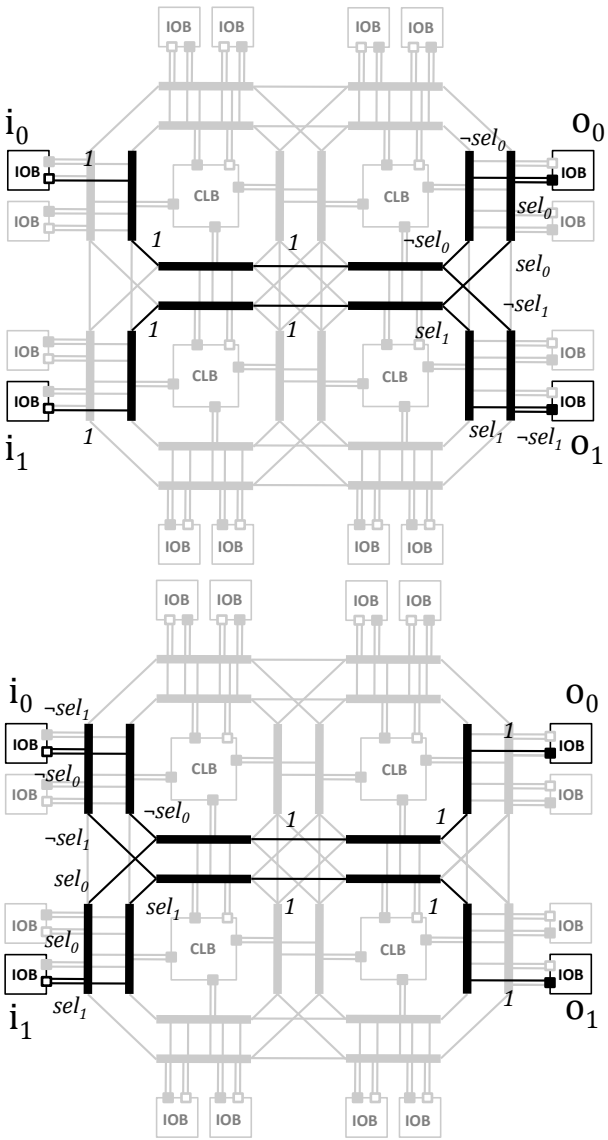


Figure 7.8: Two routing solutions for the 2×2 crossbar switch example, produced by the TCON tool flow. For every switch used in the circuit, the connection condition is given. The connection condition is a boolean function of the parameters sel_0 and sel_1.

Accordingly conventional packing is composed of two stages. In the first stage the LUTs and FFs are packed into LUT-FF pairs via a pattern matching algorithm. No alterations are needed in the first stage to pack TLUTs and FFs interconnected via TCONs.

In the second stage the LUT-FF pairs are packed in CLBs. The optimization goals are two-fold. The first optimization goal is to fill each CLB to its capacity. The second optimization goal is to minimize the number of occupied inputs to each CLB. This is done in order to reduce the number of connections to be routed between CLBs and thereby exploiting the fast internal routing resources of the CLBs and enhance the overall routability of the circuit.

In a similar manner as in the conventional packing algorithm, TPACK greedily packs the BLEs with highest attraction in CLBs sequentially. The attraction of a BLE and a cluster of BLEs is the number of inputs and outputs they have in common. A hill-climbing phase is invoked only if the greedy phase is unable to fill a CLB completely. However, the problem of packing BLEs connected via TCONs slightly differs from the problem of packing BLEs connected via nets. TCONs can share input pins of the CLBs, nets cannot. TCONs driven by the same source or TCONs not active at the same time can share a CLB input pin. The only modification applied to the conventional packing, is the way the number of occupied inputs of a CLB is calculated. Given the set of TCONs that connect the inputs of the BLEs to the routing interconnect network, the number of occupied inputs is the maximum number of sources connected to the CLB at the same time.

7.5 TPLACE

This section describes modifications made to the different aspects of a conventional FPGA placement algorithm to deal with tuneable circuits. For an introduction on placement algorithms we refer the reader to Chapter 5. We use the wirelength-driven simulated annealing placement algorithm as a starting point to build TPLACE. TPLACE has two inputs: a tuneable circuit and a description of the target FPGA architecture. TPLACE searches a legal placement for the functional blocks of the input circuit so that the total wire length is minimized. In the future more complex optimisation goals, such as timing-driven [97] placers, can be implemented.

The placement problem is computationally hard, so there are no known algorithms that can find an optimal solution in a reasonable time. Therefore, many heuristics have been developed for the placement problem. In this chapter we used a simulated annealing algo-

rithm to place tuneable circuits, but the same principle can be applied to more advanced placement techniques like an analytical placement technique or a gradient descent based placement technique as described in Chapter 5.

Wirelength-driven simulated annealing placers try to minimize the total wire length needed to route all wires in a placement. They use the total wire length as their cost function. The only way to exactly calculate the total wire length for a given placement is to route the connections in the circuit and to count the used wires. Since routing is in itself a computationally hard problem, solving it repeatedly for every move tried in the inner loop of the SA algorithm, leads to very long execution times. Therefore, the cost is not exactly calculated but estimated.

7.5.1 Wire Length Estimation for Nets in Static Circuits

In this section we partially repeat what we described in Section 5.2.1 to clearly accentuate the difference with the wire length estimation in tuneable circuits, which is explained in the next section.

In case of a conventional static circuit, a legal routing solution contains a disjoint set of routing resources for each net. A wirelength-driven placer estimates the total wire length as the sum of the estimated wire lengths of each net. The wire length of a net is estimated as the half-perimeter of its bounding box weighted by a factor which depends on the number of terminals of the net.

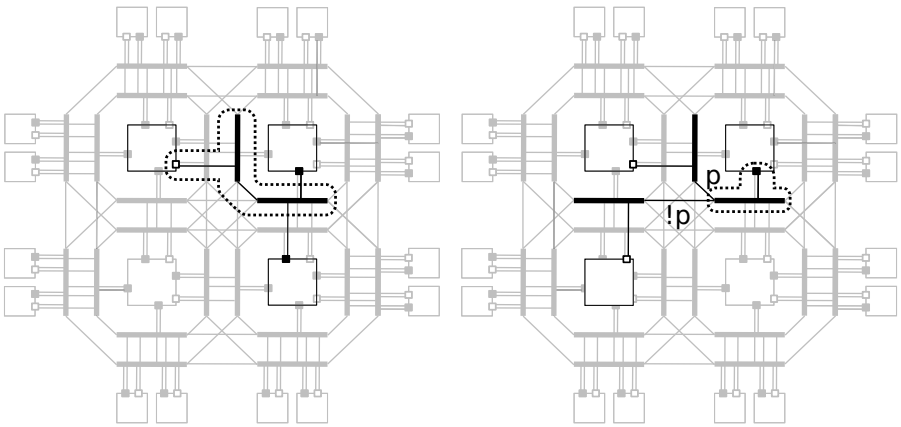
$$C_{wl} = \sum_{n \in nets} q(\#terminals(n)) \cdot HPWL(n) \tag{7.2}$$

The factor $q(.)$ is taken from [29]. It is equal to 1 for nets with up to three terminals and slowly grows to 2.79 for nets with 50 terminals.

To evaluate the estimation, the circuits in the Toronto 20 benchmark suite [15], the 20 largest circuits of the MCNC benchmark suite, were placed with the wirelength-driven placer and routed with the breadth-first router in VPR 4.30 with default settings. The channel width had 20% more tracks than the minimum channel width, to allow a relaxed routing, as recommended in [17]. To evaluate the conventional estimation we calculate the correlation between the estimated and the actual routing cost of the placed benchmark circuits. The resulting correlation coëfficient is 0.9705. Further on this correlation coëfficient will be compared with the correlation coëfficient of the newly proposed estimation method for tuneable circuits to evaluate the quality of the proposed estimation.

7.5.2 Wire Length Estimation for Tuneable Circuits

This section describes how the routing resource usage of a tuneable circuit can be estimated. It is important not to make the calculation of the estimation more complex than the estimation of the nets in static circuits, because the estimation is needed in the kernel of the simulated annealing algorithm. Hence the time needed for the estimation should be reduced to the very minimum.



(a) Sharing of routing resources between connections with the same source.

(b) Sharing of routing resources between connections (TCONs) with the same sink.

Figure 7.9: Sharing of routing resources between connections with the same source or sink. Shared resources are annotated with a dotted shape.

A new wirelength estimation method is necessary because in tuneable circuits, a routing solution does not contain a disjoint routing set for each of the TCONs. There are two sharing mechanisms. In Figure 7.9 the two sharing mechanisms are demonstrated on a small example routing solution. TCONs can legally share resources with other TCONs if they carry the same signal or if they are not active at the same time. The first resource sharing mechanism, TCONs carrying the same signal, is easily distinguished, because the TCONs are driven by the same source. The second resource sharing mechanism, TCONs that are not active at the same time, is harder to recognise. The connection conditions of the TCONs have to be compared. After comparison, each TCON t has an associated set of TCONs. Each of the TCONs in that set may share resources with t, but some of the TCONs in the set can be far away from t and are not interesting to consider as sharing resource with. For example, two connections that are not active at the same time

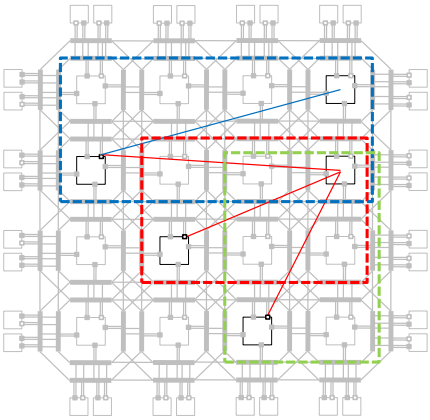
with the terminals of the connections situated on the other side of the FPGA. These two connections will most likely not share resources, even if it is allowed. The most interesting TCONs in the set are the TCONs that have the same sink as t. They can be distinguished easily and they are forced to overlap because they have to reach the same sink anyway. So to simplify the problem we consider only overlap between TCONs with the same source or TCONs with the same sink.

The estimation is not as straightforward as in case of static circuits. In case we use the same estimation method that is used for static circuits, then the second sharing mechanism is neglected and the wirelenght is systematically overestimated. Let's consider the toy example in Figure 7.10a. The conventional estimation method considers all connections starting from the source as a collection and tracks the bounding boxes of these collections during placement. The estimation for the toy example is 10 wires, but the post-route solution contains only 9 wires. The number of bounding boxes that are tracked during placement equals the number of sources in the circuit.

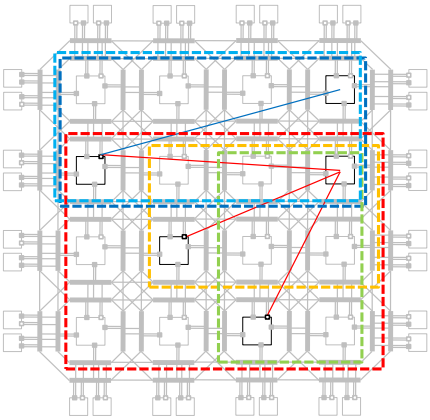
To achieve a better estimation, all connections starting from a sink could also be considered a collection. Each connection would then be part of a source collection and a sink collection. The estimation method would than track the bounding boxes of the source and the sink collections. The total wirelength estimate in this case is the half of the sum of the bounding box estimates. As can be seen in Figure 7.10b, the estimated wirelength is 9.5, which is slightly better than the conventional estimation, but the number of bounding boxes that needs to be tracked during placement is the number of sources and sinks in the circuit, which is typically at least twice the amount of bounding boxes. The toy circuit has three sources and two sinks, so there are five bounding boxes which need to be tracked.

To reduce the number of bounding boxes that needs to be tracked and improve the accuracy even more, we propose to partition the TCONs in the tuneable circuit according to the dominant resource sharing mechanism. After the partitioning process, each TCON is only part of one collection and the number of collections is typically in the same ballpark as the conventional estimation and the accuracy is typically better. The partition-based estimation correctly predicts the number of wires for our toy example in Figure 7.10c while only needing 2 bounding boxes. This improvement is even more pronounced for tuneable circuits with larger fanout source collections and/or larger fanin collections.

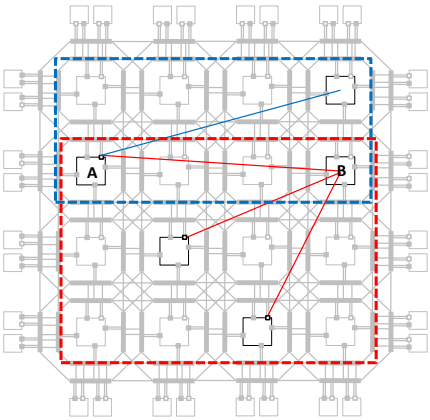
We define the dominant resource sharing mechanism to be the sharing mechanism that is used by the largest number of connections. Let's



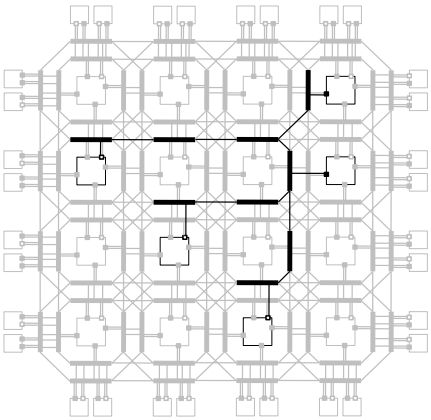
(a) Conventional
$C_{wl,est} = 10$, $\#BB = 3$



(b) Both sharing mechanisms
$C_{wl,est} = 9.5$, $\#BB = 5$



(c) Partitioning according to the dominant sharing mechanism
$C_{wl,est} = 9$, $\#BB = 2$



(d) Post-route solution
$C_{wl} = 9$

Figure 7.10: The wirelength estimation methods for tuneable circuits applied to a simple toy circuit and the post-route solution

Listing 7.4: Pseudocode of the greedy multi-iteration algorithm used to partition the TCONs

```
1  while (candidateParts.size > 0):
2    int maxCardinality = 0
3    Set maxPart = 0
4    /* Find the set with the maximum number of TCONs */
5    for each Set s in candidateParts do:
6      if (s.size > maxCardinality || (s.size == maxCardinality && BB(s) > BB(
           maxPart)))
7        maxCardinality = s.size
8        maxPart = s
9    partitioning.add(maxPart)
10   /* Remove each TCON of maxPart from its other set */
11   for each TCON t in maxPart do:
12     Set source = candidateParts.getSourceSet(t)
13     Set sink = candidateParts.getSinkSet(t)
14     if (maxPart == source) sink.remove(t)
15     else source.remove(t)
```

consider the TCON that connects block A and B in the toy circuit depicted in Figure 7.10c. The TCON is driven by a source that has a fanout of two (two connections in the circuit are driven by this source). On the other hand, the sink of the TCON is also used by two other connections. In this case the dominant resource sharing mechanism for this TCON is sharing between TCONs with the same sink.

To partition the TCONs, according to the dominant resource sharing mechanism, we propose a greedy multi-iteration algorithm. In Listing 7.4 the pseudocode of the partitioning algorithm is given. Initially, the TCONs sharing the same sink are part of the same sink set and TCONs sharing the same source are part of the same source set. Each TCON is therefore initially part of 2 sets, a source set and a sink set. All the sets are added to a *possibleParts* collection. Each iteration of the partitioning algorithm consists of two steps. It starts with greedily selecting the largest set in the *possibleParts* collection and in the second step each TCON in the selected set is removed from the other set of which they are part of. The loop iterates until all the connections are covered by the chosen sets. In the end every TCON belongs to only one set.

The total wire length of the tuneable circuit is then estimated as shown in Equation (7.3). The estimate is the sum of the estimated wire lengths of each set in the partition, where the wire length of a set is estimated as the half-perimeter of its bounding box weighted by a factor which depends on the number of terminals of the set.

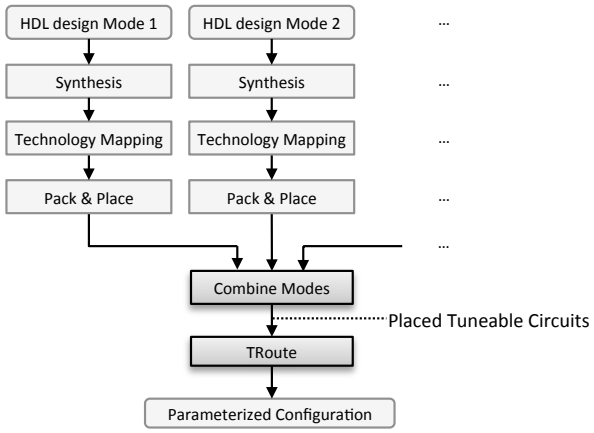


Figure 7.11: A tool flow for the compilation of multi-mode circuits, that increases the routing similarity between modes. The tuneable circuits and the corresponding placement produced by this tool flow can be used to evaluate routing cost estimations for tuneable circuits.

$$C_{wl} = \sum_{\forall s \in partition} q(\#terminals(s)) \cdot HPWL(s) \tag{7.3}$$

The same weighting factors $q(.)$ are used as in the conventional estimation of a net, see Equation (7.2). For source sets this is quite straightforward, because they are the equivalent to nets in a static circuit, but the estimation can be generalised for the sink sets. The factor $q(.)$ is independent of the fact if a set is a source or a sink set. This generalisation can be justified because the bounding box estimation first proposed in [29] has as goal to estimate the cost of a minimum Steiner tree. Routing sink or source sets is in fact the same problem, finding the shortest interconnect for a given set of terminals, which is the minimum Steiner tree problem.

7.5.3 Evaluation of the Wire Length Estimation

To evaluate the proposed partition-based estimation, tuneable circuits generated by the multi-mode tool flow in Figure 7.11 are used as benchmarks. The tool flow in Figure 7.11 was first proposed in [8]. A multi-mode circuit implements the functionality of a limited number of circuits, called mode circuits or modes. For example all the circuits implemented in the dynamic part of modular configuration, see Figure 7.1. At any given time only one mode circuit needs to be realised. Using

DPR all the modes of a multi-mode circuit can be time-multiplexed on the same FPGA area, requiring only an area that can contain the biggest mode. This way significant area savings can be realized compared to a static implementation of the multi-mode circuit that uses space-multiplexing to switch between modes. The tool flow aims at decreasing the reconfiguration time when changing modes by increasing the similarity of routing configurations of the different modes. Given a number of placed mode circuits, a tuneable circuit is generated where the parameter indicates the mode. The tuneable circuit is then routed with TROUTE, which stimulates sharing of routing resources between the connections of different modes. This results in a factor of 1.5 up to 5 reduction in reconfiguration time (depending on the number of nodes and the granularity of reconfiguration) in comparison with conventional methods. The downside is an increase of 10% up to 25% in total wirelength (depending on the number of modes).

The multi-mode tool flow was used to combine the following circuits from the MCNC benchmark suite, *e64, rd73, s400, alu4, s1238, s1488* and *s1494*. The placed tuneable circuits with 2 up to 5 modes were routed with TROUTE. The wire length usage was first estimated using the partition-based method and compared with the actual wire length after routing. In this experiment it turned out that the proposed estimation and the actual routing cost for these multi-mode circuits correlate strongly, with a correlation coëfficient of 0.9904, which is stronger than the conventional estimation for the Toronto 20 benchmark suite, which obtained a correlation coëfficient of 0.9705, see section 7.5.1. This newly proposed partition-based estimation is used to build TPLACE and enable high quality placement for tuneable circuits.

7.6 TROUTE

After placement, there is a physical block on the FPGA assigned for each of the functional blocks in the tuneable circuit. The router then needs to determine which of the switches in the routing architecture need to be closed and which need to be opened in order to connect the physical blocks in accordance to the way their associated logic blocks are connected in the tuneable circuit. In a tuneable circuit with TCONs, the interconnection pattern is dependent upon the value of the parameters. So TROUTE outputs a Boolean function of the parameters for each switch used in the FPGA's interconnect network. This Boolean function indicates the state the switch should be in, dependent on the value of the parameters. TROUTE is built based on the CROUTE algorithm described in Chapter 6. We also refer the reader to Chapter 6 for an intro-

Listing 7.5: Pseudo code of the TROUTE algorihm.

```
1  while (IllegalSharedResourcesExist()):
2      for each TCON t do:
3          t.ripUpRouting()
4          t.Dijkstra(t.source,t.sink)
5          t.resources().updateSharingCost()
6      allResources().updateHistoryCost())
```

duction on the routing problem and algorithms. This section describes the alterations made to CROUTE to enable routing tuneable circuits.

7.6.1 The TCON Routing Problem

When the routing architecture of the FPGA is represented as a routing-resource graph, the routing problem for tuneable circuits reduces to finding a simple path in the routing-resource graph for each of the TCONs in the tuneable circuit. These paths should only share nodes if the corresponding TCONs may legally share resources, otherwise this would lead to short circuits. Each path starts at the source node and ends at the sink node of its associated TCON and contains the wires needed to realise a connection between the sink and the source of the TCON. Figure 7.8 shows the legal solution of the 2×2 crossbar on a routing resource graph of a simple 2×2 island style FPGA with wires spanning only 1 CLB and bidirectional switches. The wires are represented by solid black lines, the input pins and output pins by small squares. The input pins are filled and the output pins are not. For the sake of clarity the individual edges are not drawn but the thin lines each represent two edges, one for each direction.

7.6.2 Modifications to the Negotiated Congestion Loop

The main structure of the TROUTE algorithm is the negotiated congestion loop, similar to the CROUTE algorithm from Chapter 6. The pseudo-code is shown in Listing 7.5. In every routing iteration, the algorithm rips up and reroutes all the TCONs in the tuneable circuit. The negotiated congestion mechanism determines the cost of the node by modulating the cost of a node with congestion penalties:

$$c(n) = c_{prev} + \cdot \frac{b(n) \cdot h(n) \cdot p(n)}{1 + share(n)} + \alpha \cdot c_{exp}, , \tag{7.4}$$

where $b(n)$ is the base cost, $p(n)$ the present congestion penalty, and $h(n)$ the historical congestion penalty. The cost of the node is divided

by $share(n)$, the number of TCONs that legally share the node with the TCON that Dijkstra's algorithm is currently searching a path for.

To explain the main difference with CROUTE, we repeat the equations 6.4 and 6.5 for the update to the congestion multipliers from Chapter 6. The present congestion penalty, $p(n)$, is updated whenever a TCON is rerouted. The update is done as follows

$$p(n) = \begin{cases} 1 & \text{if } cap(n) > occ(n) \\ 1 + p_f(occ(n) - cap(n) + 1) & \text{otherwise} \end{cases}, \quad (7.5)$$

The historical congestion penalty is updated after every routing iteration i, except for the first iteration. The update is done as follows

$$h^i(n) = \begin{cases} 1 & \text{if } i = 1 \\ h^{(i-1)}(n) & \text{if } cap(n) \geq occ(n) \\ h^{(i-1)}(n) & \\ +h_f(occ(n) - cap(n)) & \text{otherwise} \end{cases}. \quad (7.6)$$

$cap(n)$ represents the capacity of the node and $occ(n)$ is the occupancy of the node. The occupancy is a measure for the congestion of a node. For CROUTE this is straightforward, the number of different nets that use the node in question. For TROUTE the occupancy of a wire node is calculated by taking the minimum of the number of different sources and the number of different sinks of the TCONS using the node in question. In case of source nodes and output pin nodes the occupancy is calculated as the number of different sources of the TCONs that are using the resource. In the same manner the occupancy of sink nodes and input pin nodes is the number of different sinks of the TCONs that are presently occupying the node. So herein lies the biggest difference with CROUTE.

The factor p_f is used to increase the illegal sharing cost as the algorithm progresses and the h_f is used to control the impact of the historical congestion penalty on the total resource cost. The way the factors p_f and h_f change as the algorithm progresses is called the routing schedule. Again the same routing schedule proposed for CROUTE is used.

7.6.3 Resource sharing extension

As explained in section 7.5.2, TPLACE and TROUTE only consider routing resource sharing between TCONs with the same source or same sink. A TCON may share resources with another TCON with the same sink, because they are not active at the same time. However, there is a possibility that there are other TCONs not active at the same time, with another sink but with terminals in the vicinity of the terminals of

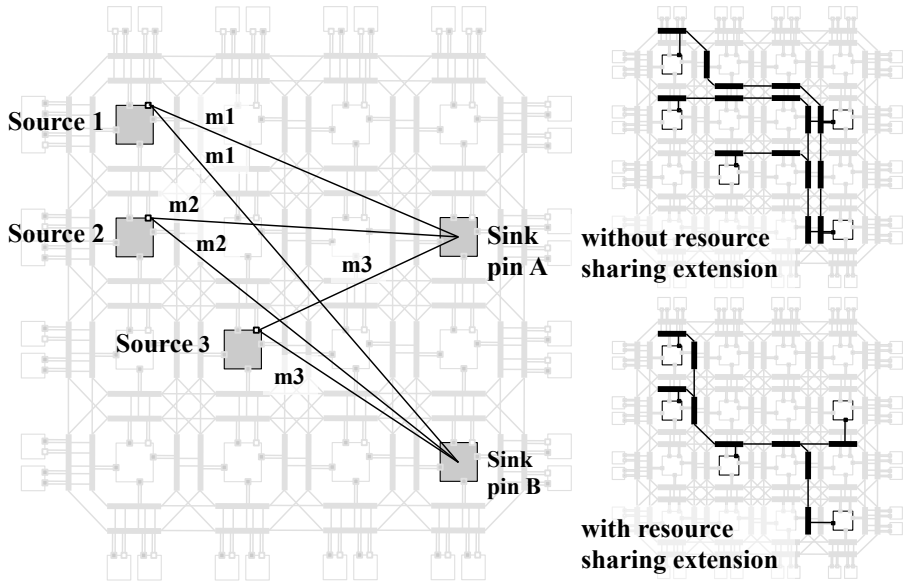


Figure 7.12: Two neighboring sink sets of a multi-mode circuit that will be merged in the resource sharing extension. The modes, in which each TCON has to be active, are annotated.

the first TCON. These TCONs are interesting to share resources with, because this can further reduce the total wire length. In this section we describe an extension to our initial resource sharing mechanism. The consequences for the total wire length and minimum channel width are described.

For each sink set, the neighboring sink sets are considered to share resources. Neighboring sink sets are sink sets that contain at least one TCON driven by the same source. In Figure 7.12 two sink sets of a multi-mode circuit are depicted. The set of sink pin A and the set of sink pin B are neighboring sink sets, because both sink sets have a TCON from each of the sources emphasized in the figure.

If every TCON in the neighboring sink set may share resources with all the TCONs in the current sink set, the sink sets can be merged. The set of sink pin A and the set of sink pin B can be merged, because the TCONs are driven by the same source or are active in a different mode. On the right side of figure the wires necessary to realise these sink sets are given. Without the resource extension 15 wires are needed and with it only 9 wires. We used a multi-mode circuit because it is intuitive and easy, but this extension is also applicable to other kinds of tuneable circuits.

Table 7.1: The resource sharing extension tested on the hybrid TCON-TLUT implementation of the Clos switch networks.

Size		without	with
16	Min. Channel Width	8	6
	Total Wire Length ($\mathbf{CW}_{test} = 10$)	514	415
64	Min. Channel Width	18	16
	Total Wire Length ($\mathbf{CW}_{test} = 21$)	3921	4055
265	Min. Channel Width	30	28
	Total Wire Length ($\mathbf{CW}_{test} = 36$)	30213	29572

This extension leads to an average decrease in the total wire length of 7.23% (standard deviation 3.39%) and a decrease in minimum channel width of 6.41% (standard deviation 4.28%) for the multi-mode circuits introduced in section 7.5.3. Another application that benefits from this extension is the TCON implementation of the Clos switch networks, described in section 7.7.4, with a modest but significant decrease of 2 tracks for the minimum channel width, the solutions need about the same amount of wires, see Table 7.1. To measure the wire length an FPGA is used with a channel width ($\mathbf{CW}_{test}$) with 20% more tracks than the minimum channel width needed to implement the Clos switch without resource sharing extension.

7.7 Applications and Experiments

This section describes the experiments done with the TCON tool flow. We demonstrate the usefulness of this flow for two main applications: a Clos switch network and VCGRAs. This section is divided in 4 subsections. The first subsection describes the properties of the FPGA architecture that is used to implement the applications. In the second subsection an overview is given of the different properties of the implementations that were measured and the circumstances under which they were measured. The third and the fourth subsection each describe the Clos switch network and the VCGRAs respectively. The conventional implementation is compared with the TCON implementation.

7.7.1 FPGA Architecture

The FPGA architecture used in this chapter is based on the `4LutSanitized`[1] architecture. It is a basic architecture, with only three

[1] A description of this architecture is provided with the VPR tool suite 4.30 in `4lut_sanitized.arch`.

types of functional blocks (inputs, outputs and CLBs) and two types of physical blocks (IOBs and CLBs). The inputs and the outputs can be placed on the IOBs while the functional CLBs can be placed on the physical CLBs. The CLBs contain only one 4-input LUT and one flip-flop. The logic blocks are connected to the routing network via fully-connected connection blocks. The wire segments in the interconnection network only span one logic block. Two modifications were made to the routing architecture to better resemble the commercial FPGAs. Wilton switch blocks are used instead of disjoint switch blocks and unidirectional wires are used instead of bidirectional wires [79]. The architecture is specified by three parameters: the number of logic element columns ($cols$), rows ($rows$) and the number of wires in a routing channel (W).

7.7.2 Methodology

For each application three FPGA implementations were compared, **Conv**, **TLUT** and **TCON**. They were generated with the conventional, the TLUT and the TCON tool flow, respectively. The Conv and TLUT implementations were placed with the wire length driven simulated annealing placer and router in VPR 4.30 with default settings. For each implementation, the following properties of the implementation were measured;

- **Area**: The number of LUTs in the circuit (**#LUTs**). To place and route the implementation a square FPGA was used with 20% more LUTs than present in the circuit. The dimensions of the FPGA will be given in the **'Dim'** column.
- **Logic Depth** (**LD**): The maximum number of LUTs a signal needs to travel through to get from the input to the output.
- **Minimum Channel Width** (**CW**$_{min}$): the minimum number of tracks the router needs to find a routing solution. The routers have 30 routing iterations to find a solution.
- **Total Wire Length** (**WL**): The number of wires needed to realise the design. To measure the wire length, the FPGA had a channel width with 20% more tracks than the minimum channel width of the implementation with the highest minimum channel width. This guarantees a relaxed routing [17] for every implementation and a fair comparison.
- **Runtime** (**RT**): The place and route step were executed on a workstation with a 2.13GHz Intel Core 2 processor and 2GB memory.

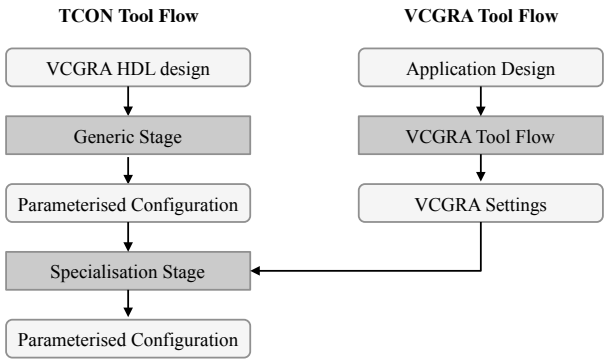


Figure 7.13: Implementing VCGRAs with the TCON Tool Flow

7.7.3 Virtual Coarse Grained Reconfigurable Arrays

FPGA tool flows need a lot of time to compile a design in a FPGA configuration. To avoid this problem, Virtual Coarse Grained Reconfigurable Arrays (VCGRA), or CGRAs implemented on FPGAs, have been proposed. Conventional implementations of VCGRAs use LookUp Tables, to implement the virtual switch blocks, registers and other components that make the VCGRA configurable. In [58] this large overhead is avoided by mapping these components directly on TCONs and TLUTs. An HDL design of the VCGRA, with the settings of the VCGRA annotated as parameters, is compiled by the TCON tool flow to a parameterised configuration. An overview of this method is given in Figure 7.13.

As an example we will describe a VCGRA for regular expression matching, and show how it can be implemented with a significantly smaller area. For a full comparison with existing approaches to regular expression matching, we refer the reader to [58].

Regular expressions are a way of describing patterns of data. They are extensively used in network intrusion detection and prevention systems to detect malicious network packets based on content signatures. These systems need to match network packets against many regular expressions at extremely high bandwidths of up to several Gbps. This warrants a hardware based regular expression matching accelerator.

The proposed architecture uses a Nondeterministic Finite Automaton (NFA) type implementation of regular expression matching. This lends itself very well to parallelisation and implementation in hardware. An NFA-style regular expression matcher basically consists of a set of states and a set of transitions between them. The matcher pro-

cesses an input string at one byte per cycle. During each cycle, every transition concurrently checks if its input state is active and its transition condition is satisfied by the input character. If that is the case it will activate its output state in the next cycle.

Each Processing Element (PE) of the proposed VCGRA contains the equivalent of one transition and one state in addition to some logic to allow for efficient implementation of constrained repetitions. The regular expression matcher in this chapter has a global decoder. The global decoder is implemented by one or more memory blocks, depending on the number of PEs in the design. For example, the 18-Kbit RAM Blocks in the Virtex-5 device family have a maximum read width of 36 bits. So one 18-Kbit RAM block can drive 36 PEs. Each 8-bit character represents an address in the memory (256 entries). Each entry contains a bit for each PE. This bit is 1 if the bit has to be activated for that character. So one PE has the following functionality:

- A PE can match one input character against an arbitrary character class.
- A PE can implement repetition ('?', '*', '+' and constrained repetition with an upper bound of at most 255) of a character class.
- A group of PEs can be configured to implement the union, Kleene-star ('*'), at-most-once repetition ('?') and at-least-once repetition ('+') of one or more subexpressions.
- A PE can also be configured as pass-through. The input signal is then routed trough the PE without applying any operation to it.

Every PE has 22 setting bits to configure it. The design of this PE is based on [43]. Since one PE only implements one state, many PEs have to be combined to match a complex regular expression. A network of connections between the PEs is provided for this purpose. The 22 setting bits of each PE and the settings that determine whether or not to establish a connection between PEs, will be the parameters of the design.

Using the TCON tool flow to set up this VCGRA for a Regular Expresssion matcher, leads to an area reduction of 50% in comparison with the conventional tool flow. The properties of the implementation can be seen in Table 7.2. The total wire length decreases with 31-35% in comparison with the conventional implementation and 12-16% in comparison with the TLUT implementations. The downside is the increase in minimum channel width with two tracks for the 8×8 and for the 14×14 grid, but the channel width increase is relatively low. The

Table 7.2: Properties of the Regular Expression Matcher for three different grid sizes

Size	Impl	#LUTs	Dim	CW_{min}	WL	RT_{place}	RT_{route}
	Conv	1141	38	6	5989	3.1s	7.3s
4×4	TLUT	653	28	6	4446	2.2s	5.0s
	TCON	577	27	6	4139	2.2s	4.6s
	Conv	4581	75	6	26428	1m36s	1m38s
8×8	TLUT	2697	57	6	19230	45s	43s
	TCON	2305	53	8	16939	48s	56s
	Conv	14061	130	6	86328	3m11s	4m45
14×14	TLUT	8403	101	6	61835	2m28s	2m37s
	TCON	7057	93	8	55049	2m19s	2m42s

number of routing resources reduces if we take into account that the dimensions of the FPGA. The dimensions of the FPGA for the TCON implementation are further reduced in comparison with the TLUT and Conv implementations. The logic depth of the expression matcher depends on the settings of the VCGRA and is not considered here, but the logic depth of one PE decreases from 8 (Conv) to 3 (TCON). The runtime to route the grids decreases with about 50% in comparison with the conventional implementation. The runtime to place the implementations decreases also with about 50% for the 4×4 and 8×8 grid, but the decrease drops to 22% for the 14×14 grid.

7.7.4 Clos Networks

TCONMap maps some functionalities to a tuneable circuit with only TCONs and no TLUTs [57]. These functionalities are crossbars, multiplexers or shifters with low speed/high latency control signals. If the control signals are selected as parameters, these functionalities will be completely implemented by the FPGA's interconnect network. Applications in which such functionalities are used, benefit from the TCON tool flow. However, in some cases, the application is built with only crossbars, shifters or multiplexers. This is not desirable, because there are not enough routing resources available. This section describes how it is possible to balance the functionality between TCONs and TLUTs. As example, a hybrid TCON-TLUT implementation for Clos Networks [34] is proposed, similar to the implementations in [93, 153]. Since our design is done at the abstract level of tunable circuits, while theirs is done at the architectural level, our method greatly reduces the design effort.

Clos networks are multi-stage interconnect networks built up by

Table 7.3: Properties of the multi-stage Clos switch network implementations for three different sizes. The size indicates the number of inputs/outputs of the Clos switch.

Size	Impl	#LUTs	Dim	LD	CW_{min}	WL	RT_{place}	RT_{route}
	Conv	202	20	5	6	2340	1.6s	1.7s
16	TLUT	48	10	3	6	492	0.5s	0.4s
	Hybrid	16	8	1	6	453	0.8s	0.6s
	Conv	1016	47	9	8	11941	18s	1m43s
64	TLUT	320	23	5	10	3034	2.4s	2.1s
	Hybrid	128	18	2	16	4289	4.9s	8s
	Conv	6760	114	12	10	98471	9m37s	25m22s
256	TLUT	1792	53	7	14	23612	52s	2m15s
	Hybrid	768	39	3	28	30579	1m32s	4m18s

crossbar switches. In our implementation we use 4×4 crossbars as building blocks, these can be efficiently implemented with 4 TLUTs or 16 TCONs. The crossbar switches in the even stages are implemented using TLUTs and the crossbar switches in the odd stages are implemented with TCONs. This results in a good balance between TLUTs and TCONs. Other balanced solutions can be easily found by simply changing the choice of TLUTs or TCONs.

The hybrid TCON-TLUT implementation is compared with the Conv and TLUT implementation for three sizes: 16×16 (3 stages), 64×64 (5 stages) and 256×256 (7 stages). Figure 7.14 shows a schematic of the TLUT and the hybrid TCON-TLUT implementation of the 16×16 Clos network.

The advantages of the hybrid TCON-TLUT implementation is the enormous reduction in area and logic depth. The hybrid TCON-TLUT implementation needs a factor of 8 to 12 less area than the Conv implementation and a factor of 2.3 to 3 less area than the TLUT implementation. The logic depth decreases with a factor of 4 to 5 in comparison with the Conv implementation and a factor of 2.3 to 3 in comparison with the TLUT implementation.

The downside is the increase in channel width up to a factor of 2.8 in comparison with Conv implementations and up to a factor of 2 in comparison with TLUT implementations. It is possible to give up some of the area gain to spread out the routing and achieve lower channel widths, but to fully exploit a routability-driven version of TPLACE needs to be developed.

The total wire length decreases with a factor 2.8 to 5 in comparison with Conv implementation but increases with a factor of 1.3 to 1.4 in comparison with the TLUT implementation for the 64×64 and the $256\times$

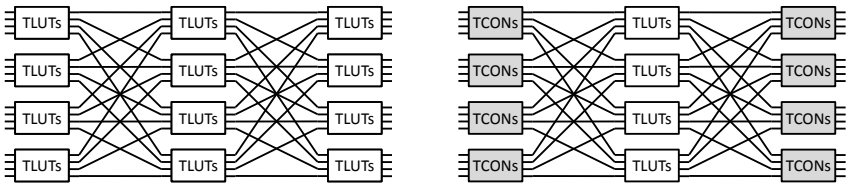
256 Clos networks.

The runtime of placement decreases with a factor 2 to 9 in comparison with the Conv implementation but increases with a factor of 1.7 to 2.0 in comparison with the TLUT implementation. So TPLACE needs more time to place the smaller TCON-TLUT circuits than the VPR wire length driven placer needs to place the larger TLUT circuit. This is because the hybrid TCON-TLUT implementations have relatively more interconnections than the TLUT implementations, so during placement there are more interconnections for which the placer needs to estimate the routing resource usage. This is also the reason why the total wire length increases for the hybrid implementation in comparison with the TLUT implementation. The same applies for TROUTE, but more extreme. The runtime of the routing step decreases with a factor 2.8 to 12 in comparison with the Conv implementation but increases with a factor of 1.5 to 4 in comparison with the TLUT implementation. The main reason is that the number of connections per CLB increases and more congestion occurs at routing time. The increase in runtime relative to the size of the circuit is acceptable, because the parameterized configuration generated by TPAR can be used many times to generate a specialized configuration.

7.7.5 Runtime comparison

The runtime of the place and route tools can be compared on two different levels. The first level is the runtime to place and route the same functionality but different implementations. The runtime for TPLACE and TROUTE to place and route the TCON netlist of a TCON implementation versus the runtime for VPR to place and route the netlist of a TLUT implementation versus the runtime for VPR to place and route the netlist of a conventional implementation. This comparison was described separately in the Clos Networks section and the section on the VCGRA for Regular Expression Matching. We can conclude that the runtime difference of TPLACE and TROUTE greatly depends on the type of application, and on how dense the functional blocks in the circuit are interconnected with TCONs. The runtime of TPLACE and TROUTE scale very well in comparison with the runtimes for the conventional and TLUT implementation for the Regular Expression matcher, but not so good for the Clos switch networks. The TCON implementation of the 4×4 grid has 577 LUTs and 1830 connections. The TCON implementation of the 64×64 Clos switch on the other hand has only 128 LUTs, but 2304 connections. So clearly the routing problem for 64×64 Clos switch is harder than the 4×4 grid, which reflects in larger run-

times for the place and especially the route step.



(a) (b)

Figure 7.14: The two implementations of the 16×16 Clos network, that use reconfiguration to control the crossbar switches

7.7.6 Specialization Overhead

The specialization overhead consists of an area overhead and a runtime overhead. The runtime overhead is defined by how many times the application needs to be reconfigured and by the reconfiguration overhead, the time to evaluate the PConf and reconfigure the bits that changed. The Clos switch needs to be reconfigured when the settings of the switch, which define what outputs are connected to which inputs, change. The VCGRA needs to be reconfigured when the specific application implemented on it changes. The frequency of reconfiguration therefore greatly depends on the application in which the Clos switch or VCGRA is used. For example, the generic regular expression matcher can be used in a Network Intrusion Detection System (NIDS). It is used to rapidly implement the new regular expressions released in the last update. Updates for NIDS are typically released with intervals of a few days to a few hours. If the Clos switch is used in a telecom application, then the time a connection has to be active ranges between a few seconds to a few hours. The reconfiguration overhead is typically a few milliseconds. This is an acceptable overhead in the applications mentioned here as an example, but could be unacceptable for other applications.

The online specialization stage of the TCON tool flow also requires extra processing power to evaluate the Boolean functions in the parameterized configuration produced by the offline generic stage of the TCON tool flow. An embedded processor can be used to evaluate the Boolean functions. The Xilinx' Virtex family has embedded IBM PowerPC cores. Zynq, a recent Xilinx device family, has ARM multi-cores. If no processor is available, a softcore can be implemented, such as the microblaze processor ($\sim$1200 LUTs) or picoblaze ($\sim$200 LUTs). If the

processor is only used to evaluate the parameterized configuration, a faster and smaller customized processor can be designed [6]. To alter the configuration from within the FPGA fabric, an interface is required. In Xilinx devices this is called the Internal Configuration Access Port (ICAP). For example, the Virtex-6 devices offer two ICAPs in the center of the device, one at the top and one at the bottom. The processor has to be able to communicate with the ICAP, so a bus has to be provided. The overhead of this bus is small ($\sim$150 LUTs).

7.8 Conclusion

In this chapter we introduced new placement and routing algorithms, TPLACE and TROUTE. They are part of the easy-to-use TCON tool flow, that enables designers to use dynamic partial reconfiguration to specialize not only the lookup tables, but also the routing infrastructure of the FPGA for the runtime needs of their designs.

The TCON tool flow generates implementations that need 50% up to 92% less LUTs and 36% to 81% less wiring than conventional implementations. The logic depth of the TCON implementations decreases with 63% up to 80% in comparison with their corresponding conventional implementations. The downside is that the FPGA needs a factor 1.3 to 2.8 more tracks per channel to be able to implement the configurations generated by the TCON tool flow. This could be improved by giving up some of the area gain to spread the routing.

In the future, we want to investigate the timing behaviour of the solutions produced by TPLACE and TROUTE. Other optimisation goals can be implemented, such as a timing-driven placement and routing.

8

Logic Gates in the Routing Nodes of the FPGA

In this chapter we investigate a new kind of FPGA architecture with a routing network that not only provides interconnections between the functional blocks but also performs some logic operation. More specifically we replaced the routing multiplexer in the conventional architecture with an element that can be used as both a two input AND/OR gate and a multiplexer. The aim of this study is to determine if this kind of architecture is feasible. We developed a new technology mapping algorithm and sized the transistors in these new architectures to evaluate the area, delay and power consumption. The mapping results indicated two promising two promising architectures. We implemented packing algorithms for these architecture to be able to place and route benchmark designs on these new architectures and to evaluate the net results.

8.1 Overview

Traditionally, the part of the infrastructure of an FPGA that provides the functionality is called logic blocks and the part that provides interconnections between those blocks is called the routing network. These parts are considered and optimised separately. Past studies have focused almost exclusively on the best architecture for the routing network or for the logic blocks [111, 32, 79, 7]. For example, in [79] the au-

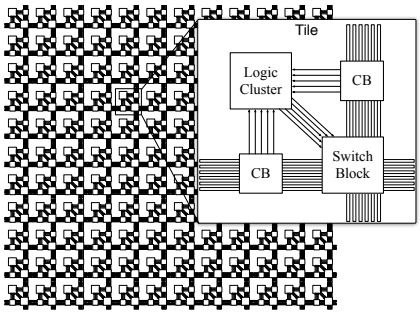


Figure 8.1: The tiled fabric of an FPGA architecture

thors describe the advantage of using a routing architecture with unidirectional and single-driven wires. Another example is the research for the optimal size of the LUTs and the optimal number LUTs in a logic block [7]. These studies never deviate from the paradigm that the blocks provide functionality and the routing network provides interconnection between those blocks.

In this chapter we loosen this assumption and take a look at the cost and performance of inserting some logic functionality into the routing network. The routing network is built up by programmable multiplexers (MUXes) and buffers. The buffers drive wires that route the output signal of the preceding MUX to other MUXes and input pins. Buffers typically consist of two subsequent inverters. In the new FPGA architecture the N:1 MUXes are replaced with N:2 MUXes and the first stage inverter is replaced by a NAND gate. This new routing node architecture can be used either as an AND gate or as a MUX.

The routing network is built up by a large number of routing MUXes, so in case something is added in each multiplexer the area overhead could increase enormously. To measure the area, delay and power consumption overhead of the new routing node architecture, COFFE [30], an automated transistor-sizing tool, was adapted to be able to size the new architecture and optimise it for area-delay-power product [137].

8.2 FPGA Architecture

8.2.1 High-level Overview

To design an FPGA, the FPGA is divided into tiles [82]. Each tile consists of a functional block and a small part of the routing network. Those tiles are replicated and stitched together to obtain a complete

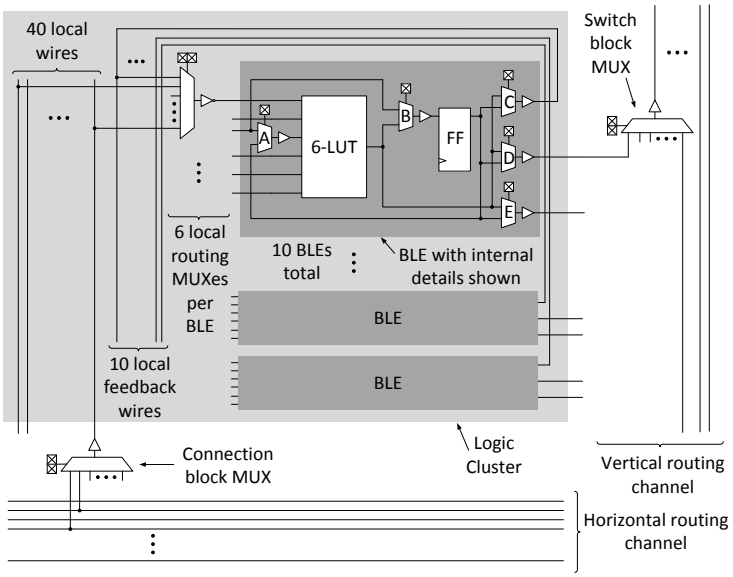


Figure 8.2: Logic cluster architecture and interconnectivity provided by the connection block and switch block multiplexers in the baseline architecture, figure reproduced from [30].

FPGA. Fig. 8.1 illustrates how the FPGA fabric is divided in tiles. In this work we focus on a tile that contains a soft block, also called logic cluster. Each tile is built up by a logic cluster (LC), a connection block (CB) and a switch block (SB). An LC provides a small amount of logic functionality and local routing. The connection blocks allow signals to be routed from the wires within the routing channels to the logic cluster. The switch block allows the output signals of the logic cluster to be connected with the wires in the routing channels and it also provides connectivity between wires within the routing channels.

8.2.2 Baseline Architecture

The baseline FPGA architecture used in the experiments is the example architecture present in COFFE [30] and depicted in Fig. 8.1. The routing channel width W is set to 320. COFFE currently only supports directional, single-driver routing wires and they span 4 logic tiles. In the switch block, each incoming wire can be programmed to connect to 3 different routing MUX inputs. A logic cluster contains 10 basic logic entities (BLEs). Each BLE has one 6-input LUT, a FF, several 2-input MUXes to use the LUT and FF simultaneously and three outputs. Two outputs are able to connect to $0.025W = 8$ routing MUXes in the switch

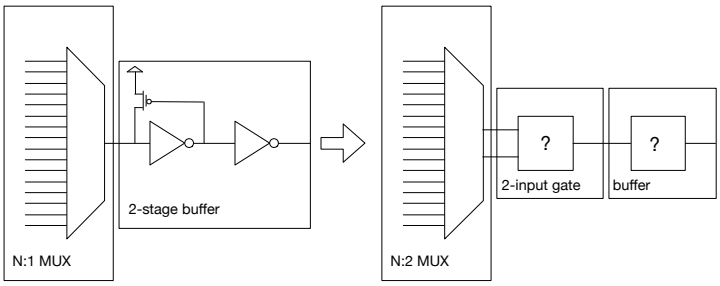


Figure 8.3: The conventional routing node consists of a N:1 MUX and a buffer. The new routing architecture has a N:2 MUX, 2-input gate and a buffer

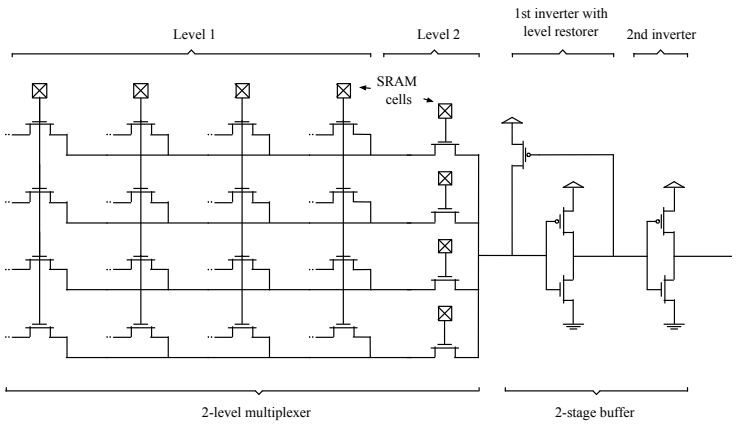


Figure 8.4: Transistor-level design of a routing node. A 16:1 two-level multiplexer and a two-stage buffer.

block, thus output signals from the basic logic entity can be routed to 8 wires in the adjacent channels. The other output drives a feedback wire. Each logic cluster has 40 inputs and each input can be driven by signals routed from $0.2W = 64$ wires in the adjacent channel. Local routing MUXes select the BLE inputs from the 40 input and 10 feedback wires. These MUXes are sparsely populated (at 50%). More information on the motivation for the different architectural parameters in the baseline architecture can be found in [30] and [31].

8.2.3 Routing Node

We define a routing node as a many-to-one MUX and a buffer that drives the subsequent wire. This is one of the basic building blocks used to build the connection block, switch block and the local interconnect in the logic cluster. In Figure 8.3 a schematic of the routing node is depicted and in Figure 8.4 the corresponding transistor-level design is shown. Routing nodes are extensively used to implement FPGAs and are an interesting point to insert logic.

A MUX in the baseline architecture is built up by NMOS pass gates. If the gate of an NMOS transistor is driven high, the NMOS transistor will conduct the signal. These pass gates can be connected in several different topologies to form a MUX, each of which possesses a different area-delay trade-off. The MUXes in the example architecture of COFFE are implemented as two-level MUXes because they have been shown to give the best area-delay product [24] and are used in commercial architectures. An important parameter in the design of two-level MUXes is the size of each level. If S_{L1} and S_{L2} are the sizes of the first and second level respectively, any combination of S_{L1} and S_{L2} such that $S_{L1}.S_{L2}$ = MUX size is a possible MUX topology. Since SRAM cells occupy 35-40% of tile area [30], we choose a MUX topology that minimizes the number of SRAM cells, this leads to $S_{L1} \approx S_{L2}$ as illustrated in the 16:1 MUX example in Fig. 8.4. Let n be the size of the MUX. The number of SRAM cells in the MUX is $\lceil\sqrt{n}\rceil + \lceil n/\lceil\sqrt{n}\rceil\rceil$ and the number of pass gates is $n + \lceil\sqrt{n}\rceil$. The 16:1 MUX example has 8 SRAM cells and 20 pass gates. The output of each MUX is driven by a two-stage buffer, built up by two inverters in series. The buffer enables faster charging of a possible large downstream capacitance. We optimized the baseline architecture by investigating the optimal number of inverter stages in the buffers and the performance in terms of area-delay-power product improved with 36% in case we added two extra inverter stages in the local interconnection routing nodes. See Table 8.1 for more detailed results. Our motivation to investigate buffer optimizations is that it will lead to a more fair comparison. Inserting logic gates in the routing nodes will give more buffering capacity to the routing nodes and this may distort the results if we do not optimize the buffering strength of the routing nodes in the baseline architecture.

NMOS pass gates are incapable of passing a full logic-high voltage. Their output voltage saturates at approximately $V_g - V_{th}$, where V_g is the gate voltage and V_{th} is the threshold voltage of the transistor. This gate-source voltage drop causes a static power dissipation in downstream inverters. This problem worsens as technology scales because V_{DD} is lowered more rapidly than V_{th} to keep the power consump-

tion within the limits. To overcome this issue, there are two techniques that are commonly used, pass gate boosting and PMOS level restorers. Both have their disadvantage. When applying gate boosting, the output voltage of the SRAM cells that drive the gate of the NMOS pass transistor is chosen slightly higher than the general voltage source in the circuit. In this way the voltage drop reduces, but it accelerates device ageing. In the example architecture NMOS pass gate voltages are boosted 200mV above the nominal V_{DD} of 0.8V. Another solution is illustrated in Fig. 8.4, a level-restoring PMOS transistor is connected across the first-stage inverter of the buffer. It creates a positive feedback loop which pulls up the degraded voltage. The main issue with PMOS level-restorers is robustness. A V_{th} that is a larger fraction of V_{DD} means it takes longer for level-restorers to turn on (which increases short-circuit currents) or, in the extreme case, they might not turn on at all.

In the following sections we evaluate if it is advantageous to insert basic logic gates in between the MUX and the driver of the routing nodes, so we are also confronted with the gate-source voltage drop problem. We also have to note that commercial FPGA designs have likely moved towards a transmission-gate based implementation for increased reliability, especially at smaller process nodes operating with lower voltages as hinted in [112]. In the initial phase of this research we were biased by the results reported in [31]. Chiasson and Betz reported that transmission gate based FPGAs were 2% worse in terms of area-delay product for the 22nm process. The results were produced by using the high performance technology process. However, most vendors use the low power technology process, because the static power consumption is the lion's share of the total power consumption for an FPGA. This changes the results in favor of the transmission gate. Although we consider pass-transistor based FPGAs in this chapter, we believe that this work can be easily adapted to consider transmission-gate based FPGAs. Additionally the recent FinFet process technology advances may allow FPGA vendors to shift back to pass gates [14].

8.3 Transistor-level Design

The schematic in Fig. 8.3 gives an overview of the general idea of the new routing node architecture. The N:1 MUX in the conventional routing node architecture is replaced by a N:2 MUX and a gate is introduced between the multiplexer and the buffer. This section focuses on the motivations for the different choices made when replacing the routing node architecture and the effects on the area and delay of the sized tile.

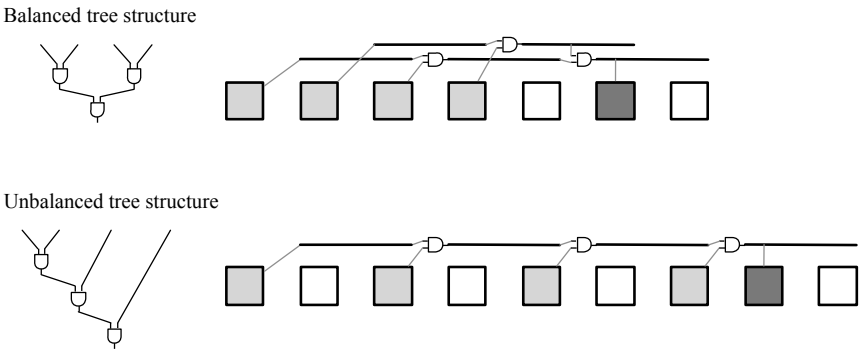


Figure 8.5: Associativity and commutativity of the AND/OR gate creates extra degrees of freedom for packing, placement and routing. The light grey boxes are the logic clusters that have an output that drives an input of the AND network and the dark grey box is the logic cluster that has an input pin connected to the output of the AND network.

To size the buffers and the gates in the new FPGA architecture we used the transistor sizing algorithm implemented in the COFFE tool [30]. COFFE's transistor sizing algorithm maintains circuit non-linearities by relying on HSPICE simulations to measure the delay. However, FPGAs are user programmable and hence they have application dependent critical paths which implies that at the time of designing an FPGA, there is no clear critical path to optimize. To deal with this issue, a representative critical path is optimised in COFFE. It contains one of each type of the FPGA's subcircuits and the delay is taken as a weighted sum of the delay of each subcircuit. The weighting scheme is chosen based on the frequency with which each subcircuit was found on the critical paths of placed and routed benchmark circuits [73]. The different weights used in COFFE are plotted in Fig. 8.8. The area is estimated using the 22nm HP Predictive Technology device Models [3].

Table 8.1 contains the sizing results of all the variations on the new architecture considered in this work. These results will be used to motivate different choices.

8.3.1 Selecting the Type of Logic Gate

The tile areas in Table 8.1 indicate that inserting logic gates in the routing adds an area and in most cases also a delay overhead. In order to minimize the area and delay overhead only non-reconfigurable 2-input gates are considered. To facilitate packing, placement and routing, the choice of gates is further restricted to only associative and commutative

gates, because associativity and commutativity leave extra degrees of freedom for packing, placement and routing. For example in Fig. 8.5, a four-input AND gate has to be realised in the routing using the smaller 2-input AND gates present in the routing nodes. There are two possibilities depicted in Fig. 8.5: a balanced and an unbalanced tree topology. Both are possible because the AND gate has the commutative and associative property. There are four eligible 2-input gates with the associative and commutative property, the AND, OR, XOR and XNOR gate. There are three compelling reasons for not considering XOR and XNOR gates. The first and most important reason is that mapping the VTR [88] and the 20 largest MCNC benchmarks to LUTs and X(N)OR gates showed that there is little to no gain in both area and logic depth of the mapped circuits compared with LUT-only mapping. Secondly, the CMOS implementations of the X(N)OR gates are larger and slower than the AND and OR gates. Thirdly, the X(N)OR gates also have to support the MUX functionality present in the original routing node. This is not straightforward, because extra transistors have to be added to be able to pass a signal. In contrast, passing a signal can be easily achieved when using AND and OR gates by connecting the supply voltage or the ground respectively to the unused input.

So the only candidates left are the AND and OR gates. The typical CMOS AND gate implementation is built up by a NAND gate with a subsequent inverter, similarly the typical CMOS implementation for the OR gate is a NOR gate with a subsequent inverter. The NAND gate in CMOS is smaller and faster because the NOR gate has two PMOS in series hence it has to be sized larger to have good rise and fall times. The NAND gate also has less leakage current than the NOR gate while implemented in CMOS. So in what follows the NAND gate and a subsequent inverter will be used as a basic building block.

8.3.2 The N:2 Multiplexer

In the new routing node all the incoming wires have to be multiplexed to two inputs of the NAND gate. Let N be the number of incoming wires of the routing node. In a first attempt we considered a new routing node architecture with the ability to provide any combination of two of the incoming signals to the AND gate. We will call this architecture the "maximum architecture". To realise the maximum architecture, all but two of the incoming wires have to be routed to two different MUXes, as depicted in Fig. 8.6. Two wires are only routed to one of the MUXes, because it is not necessary to be able to program all the possible combinations, due to the commutativity of the subsequent

Table 8.1: Tile Area, Representative Critical Path (RCP) and Power Consumption (PC) and Area-Delay-Power product (A*D*P) for the different transistor level designs of the tile.

Design					Tile Area		RCP		PC		A*D*P	
Gate	SB	CB	LI	Conn	(μm^2)	(%)	(ps)	(%)	(μW)	(%)	($\mu m^2.ns.nW$)	(%)
AND	x	x	x	max	1555	68	149	30	263	35	60,9	193
AND	x	x		max	1165	26	146	27	262	34	44,6	114
AND		x	x	max	1322	42	129	12	257	32	43,8	111
AND	x	x	x	min	1161	25	126	10	235	21	34,4	65
AND	x	x		min	1104	19	125	9	238	22	32,8	58
AND		x	x	min	1092	18	123	7	241	24	32,4	56
AND		x		min	1086	17	121	5	237	22	31,1	50
AND			x	min	1069	15	119	3	224	15	28,5	37
OR	x	x	x	max	1565	69	128	11	253	30	50,7	144
OR	x	x		max	1432	54	125	9	260	33	46,5	124
OR		x	x	max	1296	40	127	10	256	31	42,1	102
OR	x	x	x	min	1200	29	114	-1	228	17	31,2	50
OR	x	x		min	1134	22	112	-3	231	18	29,3	41
OR		x	x	min	1028	11	114	-1	189	-3	22,1	6
OR		x		min	1048	13	100	-13	242	24	25,4	22
OR			x	min	1005	8	119	3	221	13	26,4	27
Baseline COFFE arch.					986	6	121	5	237	22	28,3	36
Baseline opt. buffers					928		115		195		20,8	

The relative difference with the sizing results of the baseline architecture with optimized buffers is reported for each measure.

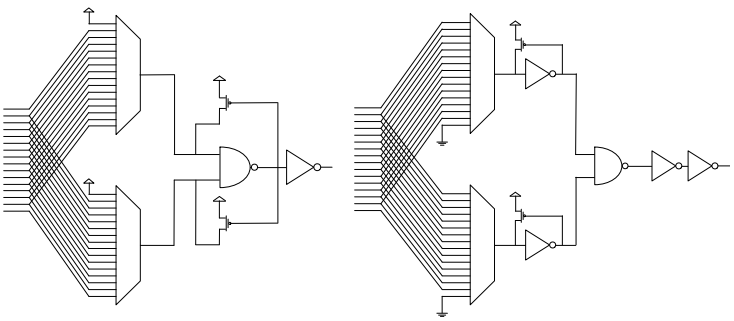


Figure 8.6: The maximum node architectures. These nodes are able to select any possible combination of two incoming signals and perform a basic logic operation or to let any incoming signal pass through. The left routing node contains an AND gate in which level restorer is switched across the NAND gate and the right routing node contains an OR gate in which the signal is restored before entering the NAND gate.

AND gate. The new routing node also has to provide the ability to pass any of the incoming signals. To achieve this both MUXes have one input connected to the voltage source. If the logic high input is selected in the upper MUX, the subsequent AND gate will pass any signal from the lower MUX and vice versa. The two MUXes have a size of N, so the resource usage is doubled compared with a single N:1 MUX in the original node. For example the 16:2 MUX in Fig. 8.6 has 16 SRAM cells and 40 pass gates, but the 16:1 MUX in Fig. 8.4 has only 8 SRAM cells and 20 pass gates. The 16:2 MUX will also create an extra wiring load for the buffers upstream that drive the incoming wires, compared with the 16:1 MUX in the original node. Taking into account a completely sized tile, see Table 8.1, the maximum architectures have a large area overhead of 23-65%, a delay overhead of 10-27% and a consistent 30-35% increase in power consumption compared with the original architecture. This results in a 2-3x larger area-delay-power product. This overhead is too high to be compensated by the gain in area and logic depth achieved by mapping to the new architecture.

To lower the area overhead, the number of combination possibilities could be restricted by only connecting each incoming wire to only one MUX, as depicted in Fig. 8.7. We call this the "minimum architecture", because the number of combinations is halved, but it still has the ability to pass any of the incoming signals. The area overhead would decrease to $2\left(\lceil\sqrt{N/2+1}\rceil + \lceil(N/2+1)/\lceil\sqrt{N/2+1}\rceil\rceil\right)$ SRAM cells and $N + 2 + 2\lceil\sqrt{N/2+1}\rceil$ pass gates. For example, in the 16:2 example

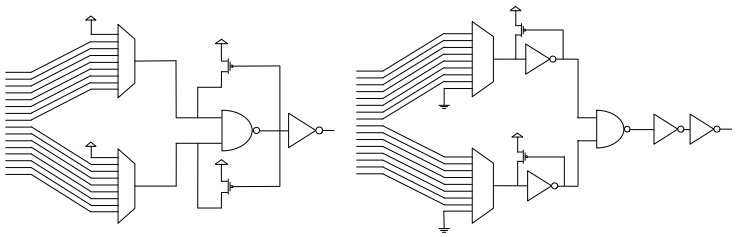


Figure 8.7: The minimum node architectures. These nodes are able to perform a basic logic operation of two incoming signals with a minimal overhead, while still able to multiplex any incoming signal. The left routing node contains an AND gate in which level restorer is switched across the NAND gate and the right routing node contains an OR gate in which the signal is restored before entering the NAND gate.

in Fig. 8.7 the area usage amounts to only 12 SRAM cells and 24 pass gates in contrast to the maximal configuration with 16 SRAM cells and 40 pass gates. This is also reflected in the sizing results. The tile area overhead of the sized minimum architectures reduces to 6-25%, as reported in Table 8.1. Another advantage is that the wiring load will not only decrease compared with the maximal configuration, it will also be smaller compared with the original routing node. The upstream buffers will only have to push the transition through MUXes half the size of the original one. This is also reflected in the representative critical path delays of the sized minimum architectures. The delay ranges from a 10% increase to a 13% decrease, depending on the level restoring tactic and the locations where the gates are inserted.

A downside of the minimum architectures is that they can only be programmed to provide about half of the number of possible combinations, from $N^2/2 - N/2$ combinations for the maximum architecture to $N^2/4$ for the minimum architecture. This will have consequences for packing and post-routing performance, which is described in Section 8.6 and 8.7. In-between these extreme routing node architectures it is possible to develop architectures in which some of the incoming wires are connected to both MUXes and some to only one.

8.3.3 Level Restoring Tactics

In the original routing node a level-restoring PMOS transistor is connected across the first-stage inverter of the buffer. This creates a positive feedback loop which pulls up the degraded voltage. There are two options to achieve the same effect in the new routing node architecture. Add two level-restoring PMOS transistors connected across the NAND

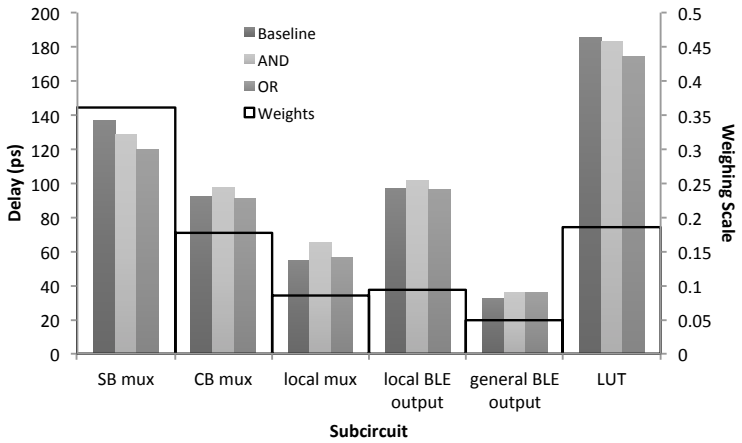


Figure 8.8: The delays for the subcircuits of an FPGA with the minimum routing node architecture implemented in each routing node. The relative importance of the different subcircuits is given by the weights of each subcircuit in the representative critical path

gate, one for each input, as depicted in the left schematics of Fig. 8.6 and Fig. 8.7. The second option is to restore the two input signals' logic high level before they are fed to the NAND gate. To realise this two inverters with accompanying level restorers are added for each input of the NAND gate. The extra inverters at the inputs change the logic functionality to a NOR gate, and an extra inverter has to be added to keep the logic functionality associative, as depicted in the right schematics of Fig. 8.6 and Fig. 8.7. The first option implements a logic AND2 gate and the second option represents a logic OR2 gate. The level restoring tactic used in the OR2 gate needs three extra inverters in the routing node compared with the level restoring for the AND2 gate, so the OR2 gate implementations are supposed to have more area overhead, but they are more reliable which leads to a similar area overhead between OR2 and AND2 gates. The sized tiles of the minimum OR2 gates architectures are typically faster than the AND2 gate architectures and in most cases even faster than the baseline with a delay that ranges between a 3% increase and a 13% decrease, despite adding extra logic into the nodes. To investigate this we plotted the delays for each subcircuit in Fig. 8.8 for the minimum OR2 gate architecture alongside the minimum AND2 architecture and the baseline architecture. A first glance shows that the different delays are very similar. The delays are slightly lower for the OR2 gate implementation and slightly higher for the AND2 gate implementation for the two most important subcircuits, the SB MUX

subcircuit and the LUT subcircuit. These are also the circuits where the wire loads are most relevant.

8.3.4 Routing Nodes in Different Locations

The routing node architecture can be modified in different locations, the connection block (CB), switch block (SB) and the intra-cluster crossbar (local interconnect, LI). An FPGA design was sized for any combination of the locations and the sizing results are tabulated in Table 8.1. Adding logic gates in every SB, CB and LI creates a large area overhead, 25-29-% for the minimum architectures, but it has a lot of flexibility to logically combine signals. For example, a frequently used subcircuit is one in which an output signal of a BLE in one cluster is routed to an input of a BLE in another cluster. In this subcircuit the signal traverses at least three routing nodes. First the routing node in at least one SB, then the one in a CB and finally the one in the LI that drives the input of the BLE. So it is possible to have three subsequent gates in-between BLEs for connections between clusters without having to detour. A three-level tree of two-input gates allows up to eight different signals processed if a balanced tree structure is used (see Fig. 8.5 for a balanced tree with 2-input AND gates). Feedback connections inside the logic cluster can also use the gate in the LI routing node, but this is limited to only one gate in-between LUTs. Only adding gates to the routing nodes in the CB and the LI or in the CB and the SB reduces the area overhead with 7-18% for the minimum architectures, but it will restrict the possibility of adding gates in-between LUTs. In case gates are only added in the CB and LI routing nodes, the number of subsequent gates in between BLEs for inter-cluster connections is restricted to two, so only four different signals can be processed. In case gates are added only in the CB and SB routing nodes, feedback connections between BLEs in the same cluster can't have a gate in between.

8.3.5 Concluding Remarks on the Sizing Results

The most promising architectures considering the sizing results are the architectures with the minimum OR routing nodes in the connection block and/or in the local interconnect multiplexers, because of their lower area-delay-power product. Additionally conventional place and route algorithms can be used, because only the routing nodes inside the CLB are changed. The results of these architectures are highlighted in green in Table 8.1. In what follows, we focus on these architectures. Mapping to an FPGA with LUTs and OR gates is in essence the same problem as mapping to an FPGA with LUTs and AND gates. The OR

network in between LUTs can easily be replaced by an AND network if we invert both the input and outputs of the network and vice versa. Explaining the technology mapping algorithm for mapping to LUTs and AND gates is somewhat simpler, because an AIG is built up by AND gates and inverting edges, so in Section 8.5 we present a novel technology mapping algorithm which targets an FPGA with LUTs and AND gates, but we start with explaining the techniques behind conventional technology mapping first.

In section 8.6 modifications to the packing algorithm are described in order to exploit the commutativity and associativity of the gates.

8.4 Conventional Technology Mapping

During technology mapping, the Boolean network generated by the synthesis step is mapped onto the resource primitives available in the target FPGA architecture, e.g. LUTs. The result of the technology mapping step is a circuit of these primitives that is functionally equivalent to this Boolean network. We focus on the technology mapping to LUT primitives, since the other primitives, such as DSPs, are typically directly inferred from HDL. An example of a LUT circuit can be found in Figure 8.9.

In this section, we first explain the optimisation criteria of a technology mapping algorithm in more detail. Then, we introduce a number of concepts used in technology mapping algorithms such as Boolean networks and cuts. Finally, we describe the conventional technology mapping algorithm.

8.4.1 Optimisation Criteria

A technology mapping algorithm typically tries to find a LUT circuit with minimal logic depth, i.e. the number of LUTs on the longest combinational path from sequential primitive to another. Sequential primitives are flip-flops, Block RAM, in- and outputs (Figure 8.9). As a secondary optimisation criterion, the area of the circuit, i.e. the total number of LUTs, is minimized.

The number of LUTs on the longest combinational path is used as an approximation of the path with the longest propagation delay, which determines the maximum clock frequency of the implemented design. Each LUT on this path contributes to the delay of the path. The interconnection between LUTs also contributes significantly to the delay, but because little is known about it at this stage, the delay is assumed

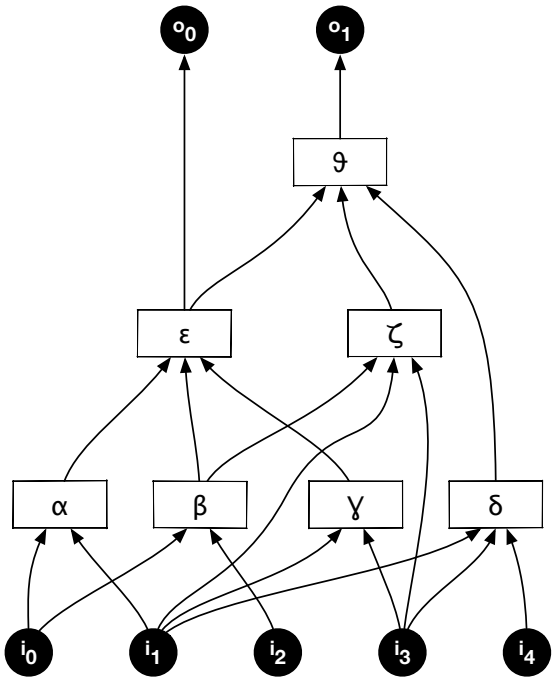


Figure 8.9: A toy LUT circuit with the functions of the LUTs shown as Greek letters. The logic depth of this circuit is 3 LUTs and the area is 7 LUTs.

to be constant per connection. In this way, the total delay of a combinational path is approximated as being proportional to the number of LUTs on the path, which is why the number of LUTs on the longest combinational path is minimized.

However, not all combinational paths should be mapped with minimal logic depth. Once the longest combinational path in a circuit is found, all shorter paths can be relaxed and allowed to have an equal logic depth. This gives the technology mapper more freedom to try to reduce the area of the circuit by extracting more common LUTs and reducing duplication of logic. When the desired clock frequency can be achieved with a longer combinational path than the minimal, an even less stringent logic depth constraint can be used to achieve a lower area cost.

The number of LUTs, in the context of technology mapping is used as a measure for area. It correlates strongly with the number of CLBs used in the final design. Although the number of available LUTs on a specific FPGA is fixed and the number of different FPGA sizes is limited, the number of LUTs in a design is typically minimized as much as possible. Besides maximising the amount of functionality that fits on the target FPGA, fewer LUTs typically result in fewer connections and less congestion. Less congestion typically leads to a lower routing runtime and a higher maximum operating frequency.

8.4.2 Definitions

Boolean network A Boolean network is a directed acyclic graph of which the nodes represent logic gates and the directed edges the wires that connect them. For technology mapping, And-Inverter Graphs (AIG) are used to represent the combinational circuits that form the output of the synthesis step. AIGs are Boolean networks containing only 2-input AND gates and inverted or non inverted edges. AIGs can represent any combinational circuit. Sequential circuits can also be handled by transforming them into combinational circuits by turning the registers into additional inputs and outputs of the circuit.

Primary inputs and outputs Primary inputs (PI) and primary outputs (PO) are the inputs and outputs of the combinational circuit.

Cut A cut with a node n as its root is a set of nodes $c = \{n_1, n_2, \dots\}$ called leaves, so that every path from the PIs to the node n passes through at least one of the leaves (Figure 8.10). The trivial cut is the set $\{n\}$ containing only the root itself.

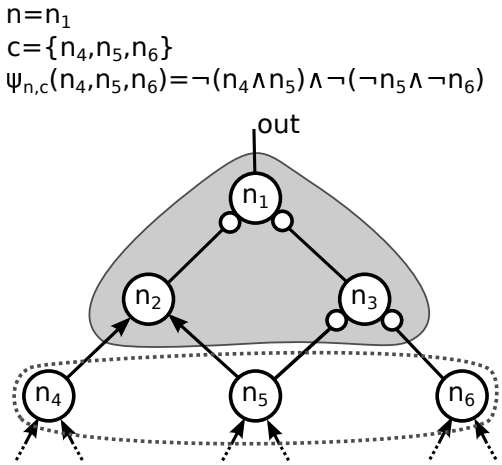


Figure 8.10: A section of an AIG with a cut (dotted line) and corresponding cone (light grey shape) drawn for root n_1. Large circles represent AND nodes, small circles inverters.

The local function of a cut $c = \{n_1, n_2, \dots\}$ with root n is the logic function that expresses the output of the root of the cut in function of the leaves: $\psi_{n,c}(n_1, n_2, \dots)$.

The cone corresponding to a cut with root n is the set of all the nodes on the paths from the leaves to the root, including the root and excluding the leaves.

8.4.3 Conventional Technology Mapping Algorithm

During technology mapping, the goal is to find a covering of the nodes of the AIG using cones, so that every cone can be implemented using one of the FPGA's combinatorial primitives. The inputs of every cone have to be available as the roots of other cones or inputs of the AIG. The result of the conventional technology mapping algorithm is a circuit of LUTs that is functionally equivalent to the input AIG. The optimisation criteria described above are used to choose one of the many possible coverings of the AIG.

The pseudocode of the conventional technology mapping algorithm [25, 35] is listed in Listing 8.1. It consists of 3 main steps: cut enumeration, cut ranking and node selection. It can be proven that after these steps a depth-optimal mapping result is found. The proof is based on mathematical induction and is almost identical to the proof given in [35]. Additional steps are performed afterwards to optimise the area of the result without increasing the depth [25]. Many varia-

Listing 8.1: Overview of the conventional technology mapping algorithm.

```
1  //Input: AIG graph
2  //Output: LUT netlist
3  function technologyMapping():
4      foreach node in aigGraph.nodesInTopologicalOrder():
5          node.enumerateAllCuts()
6      foreach node in aigGraph.nodesInTopologicalOrder():
7          node.selectBestCut()
8      foreach node in aigGraph.nodesInReverseTopologicalOrder():
9          node.determineIfSelected()
10     return aigGraph.netlistOfSelectedBestCuts()
```

tions of the algorithm exist but the main idea of the algorithm can be described as follows:

Cut Enumeration A cut is called K-feasible if its local function can be implemented using a K-input LUT, i.e. when the cut has at most K leaves. In the first step, all K-feasible cuts $\Phi(n)$ of every node n in the AIG are computed using Equations 8.1, 8.2 and 8.3.

$$\Phi(n) = \begin{cases} \{\{n\}\}, & \text{if } n \in PI \\ \{\{n_{in_1}\}\}, & \text{if } n \in PO \\ \{\{n\}\} \cup M(n), & \text{otherwise} \end{cases} \tag{8.1}$$

$$M(n) = \{c = c_1 \cup c_2 \mid c_i \in \Phi(n_{in_i}), \theta(n,c)\} \tag{8.2}$$

$$\theta(n,c) = |c| \leq K \tag{8.3}$$

The function $M(n)$ generates all possibly feasible cuts of n by combining a feasible cut of each of its inputs n_{in_1} and n_{in_2}. The function $\theta(n,c)$ evaluates if a cut can be mapped to the K-LUT primitive. Cut enumeration is done for every node in topological order from the PIs to the POs, i.e. in such a way that the inputs of each node are processed before the node itself.

In Table 8.2 you can find the results of this process applied to the AIG example in Figure 8.12.

In the cone enumeration process, redundant cuts are generated. A cut c of node n is redundant if some proper subset of c is also a cut of n (Equation 8.4, Figure 8.11). These cuts do not contribute to the quality of the mapping and can therefore be removed.

$$c \text{ is redundant in } \Phi(n) \iff \exists c_s \in \Phi(n) : c_s \subset c \wedge c_s \neq c \tag{8.4}$$

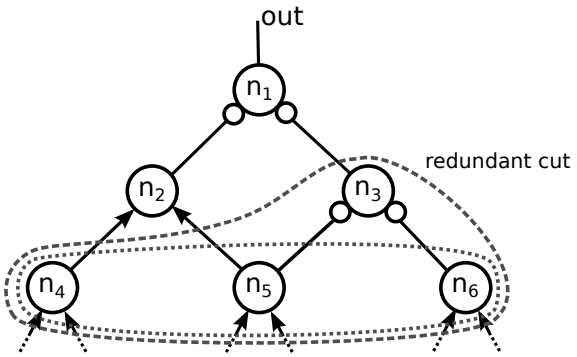


Figure 8.11: Cut $c_1 = \{n_3, n_4, n_5, n_6\}$ with root n_1 is redundant because of cut $c_2 = \{n_4, n_5, n_6\}$ and can be removed.

Cut Ranking For each node in topological order, the cut with the lowest depth is selected as the best cut. This ensures an end result with minimal number of LUTs on the longest path from any output to any input. The depth of a mapped circuit is the maximum depth of all primary outputs.

The $delay$ of a non-trivial and non-PO cut is 1, the delay of a LUT.

$$depth(n,c) = \begin{cases} 0, & \text{if } n \in PI \\ \max\limits_{m \in c} depth(bestCut(m)) & \\ \quad +delay(n,c), & \text{otherwise} \end{cases} \tag{8.5}$$

$$areaFlow(n,c) = \begin{cases} 0, & \text{if } n \in PI \\ \sum\limits_{m \in c} \frac{areaflow(bestCut(m))}{numFanouts(m)} & \\ \quad +area(n,c), & \text{otherwise} \end{cases} \tag{8.6}$$

$$bestCut(n) = \begin{cases} \{n\}, & \text{if } n \in PI \\ \operatorname{argmin}\limits_{c \in \Phi(n)}(depth(n,c), area\_flow(n,c)) & \text{otherwise} \end{cases} \tag{8.7}$$

With "$\operatorname{argmin}_a(f(a), g(a))$" the operator that returns the argument a which minimizes the function f. In case two or more arguments give the same value for f, then the argument which minimizes the function g is returned. In case the depth of two cuts is equal, the area flow [95] is used as a tie breaker. Area flow is a measure for the area of the mapped circuit. Unfortunately at the moment of ranking the area of the mapping solution is unknown. Area flow estimates the area of the subgraph below it and can be calculated simultaneously with the depth in the same forward pass of the AIG, because it can be defined recursively.

Area flow of the PIs is set to 0. The area of the best cut of the incoming edges is summed and the area of the current node is added. The sum is evenly divided among the number of edges (numFanout) flowing out of the node. This promotes the reuse of signals and gives a measure for how useful a signal is.

The results of this process for the example in Figure 8.12 can also be found in Table 8.2.

Node Selection This step starts by selecting the POs and recursively selecting all nodes in the best cuts of the previously selected nodes until the PIs are reached (Equation 8.8). The local function of the best cut of each selected node, except the POs, will be implemented using a LUT. After this step a depth-optimal mapping has been defined.

$$S = PO \cup \bigcup_{n \in S} bestCut(n) \tag{8.8}$$

Applied to the example in Table 8.2 this gives $S = \{o_0, o_1, a_{11}, a_9, a_8, a_7, a_2, a_1, a_0, i_0, i_1, i_2, i_3, i_4\}$. The corresponding cones are also shown in Figure 8.12.

Additional Steps Additional, heuristic steps can reduce the area without increasing the depth of the mapped circuit [95], this is called Area Recovery. Other heuristics try to minimize the number of connections to improve routability [64].

Complexity

Technology mapping an AIG for minimal logic depth can be done in polynomial time with reference to the size of the AIG [35]. The above algorithm guarantees that all feasible cuts will be enumerated and that those nodes will be selected that result in a covering of the AIG with minimal logic depth. Note, however, that this does not overall guarantee a circuit with minimal logic depth because other, functionally equivalent AIGs may exist that result in a better mapping. This is called structural bias [23]. Optimisation for area, on the other hand, is NP-hard and is therefore always performed using heuristics [45]. Cut enumeration also has a worst-case complexity that is exponential with reference to the number of inputs of the LUTs [36].

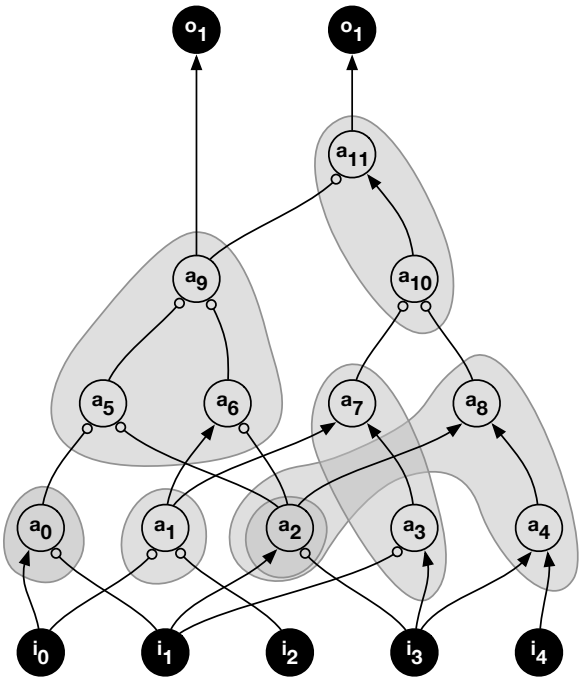


Figure 8.12: AIG of a toy circuit with the resulting cones of conventional technology mapping to 3-input LUTs. This mapping corresponds to the LUT circuit in Figure 8.9.

Table 8.2: Intermediate results of the conventional technology mapping of the AIG in Figure 8.12 to 3-input LUTs.

Node	K-feasible cuts with best cut in bold	Best cut	
n	$\Phi(n)$	$depth(n, bc)$	$area\_flow(n, bc)$
a_0	$\{\{a_0\}, \mathbf{\{i_0 i_1\}}\}$	1	1
a_1	$\{\{a_1\}, \mathbf{\{i_0 i_2\}}\}$	1	1
a_2	$\{\{a_2\}, \mathbf{\{i_1 i_3\}}\}$	1	1
a_3	$\{\{a_3\}, \mathbf{\{i_1 i_3\}}\}$	1	1
a_4	$\{\{a_4\}, \mathbf{\{i_3 i_4\}}\}$	1	1
a_5	$\{\{a_5\}, \{a_0 a_2\}, \{a_0 i_1 i_3\}, \{a_2 i_0 i_1\}, \mathbf{\{i_0 i_1 i_3\}}\}$	1	1
a_6	$\{\{a_6\}, \{a_1 a_2\}, \{a_1 i_1 i_3\}, \mathbf{\{a_2 i_0 i_2\}}\}$	2	4/3
a_7	$\{\{a_7\}, \{a_1 a_3\}, \mathbf{\{a_1 i_1 i_3\}}, \{a_3 i_0 i_2\}\}$	2	3/2
a_8	$\{\{a_8\}, \{a_2 a_4\}, \mathbf{\{i_1 i_3 i_4\}}, \{a_2 i_3 i_4\}, \{a_4 i_1 i_3\}\}$	1	1
a_9	$\{\{a_9\}, \{a_5 a_6\}, \mathbf{\{a_5 a_1 a_2\}}, \{a_6 a_0 a_2\}, \mathbf{\{a_0 a_1 a_2\}}\}$	2	17/6
a_{10}	$\{\{a_{10}\}, \{a_7 a_8\}, \{a_7 a_2 a_4\}, \mathbf{\{a_8 a_1 a_3\}}\}$	2	7/2
a_{11}	$\{\{a_{11}\}, \{a_9 a_{10}\}, \mathbf{\{a_9 a_7 a_8\}}, \{a_5 a_6 a_{10}\}\}$	3	59/12
o_0	$\{\mathbf{\{a_9\}}\}$	2	
o_1	$\{\mathbf{\{a_{11}\}}\}$	3	

8.5 Mapping to LUTs and AND Gates

In case of mapping to LUTs and AND gates, a node in the AIG can be implemented by a LUT or an AND gate if and only if both input edges of the node in the AIG are not inverted. In that case one extra cut is added to the cut set of the node. This cut has a lower depth than a cut which represents a LUT. This opportunity can be used to reduce the total depth. However this first approach is not depth-optimal. To achieve depth-optimal mapping to LUTs and AND gates, an extra degree of freedom should be used: the costless inversion of inputs and outputs of LUTs. An example AIG is depicted in Figure 8.13. If the implementation of node a_2 outputs an inverted signal, then node a_6 can be implemented by an AND gate. However for the AND gate implementation of node a_8, the output of node a_2 can't be inverted. This example shows that by introducing the inversion of the inputs and outputs of LUTs more degrees of freedom can be used to optimise the depth of the circuit, but the mapping problem clearly becomes more complex. We designed a depth-optimal algorithm with area recovery, which is described in the following sections.
In a first attempt we assumed that the extra delay introduced by the AND gate is negligible compared to the delay of a LUT and the routing delay of the connection between the output pin of that LUT and the input pin of the subsequent block. However the modifications to the mapping algorithm are generic and allow an arbitrary AND gate delay.

Listing 8.2: Pseudo code for the Technology Mapping algorithm for LUTs and AND gates

```
1  //Input: AIG graph
2  //Output: LUT netlist
3  function technologyMapping():
4    //Cut Enumeration
5    foreach node in aigGraph.nodesInTopologicalOrder():
6      node.enumerateAllCuts()
7      node.addAndCut() //always one cut with two direct inputs
8    //Cut Ranking
9    foreach node in aigGraph.nodesInTopologicalOrder():
10     foreach cut in allLutCuts(node):
11       cut.depth = maxDepthOfLeaves(minDepth(invDepth, nonInvDepth)) + 1
12     node.andCut.depth = maxDepthOfLeaves(appropriateInvOrNonInvDepth)
13     node.bestCutInverted = bestCut(allLutCuts)
14     node.bestCutNonInverted = bestCut(node.bestCutInverted, andCut)
15   // CutSelection
16   foreach node in aigGraph.nodesInReverseTopologicalOrder():
17     if node.determineIfSelected():
18       node.setAppropriateInversionVisible()
19   return aigGraph.netlistOfSelectedBestCuts()
```

In Listing 8.2, the pseudocode of the mapping algorithm is listed and in Table 8.2 the intermediate mapping results for the example in Figure 8.13. The new mapping algorithm keeps a depth for an 'inverted' or 'non inverted' implementation of each node. By using mathematical induction, we proved that this method provides a depth-optimal solution for a given input network. To keep the depth minimal it is sometimes necessary that nodes are implemented twice ('inverted' and 'non inverted'), because otherwise it would require an extra LUT to provide the 'inverted' or 'non inverted' signal. On average only 1.2% of the nodes are duplicated for this sake.

8.5.1 Cut Enumeration and Cut Ranking

In cut enumeration all possible cuts are enumerated for each node. The cut that contains the two direct inputs of the current node is copied and marked as an AND gate. In the new mapping algorithm a distinction is made whether the implementation of a node has to be inverted or not. Therefore a 'non inverted depth' and an 'inverted depth' is assigned to each node. An 'inverted implementation' is only LUT-feasible, whereas a 'non inverted implementation' is not only LUT-feasible, but also AND-feasible. The depth calculation of a cut is now as follows: if the cut represents a LUT, the maximal depth of all input

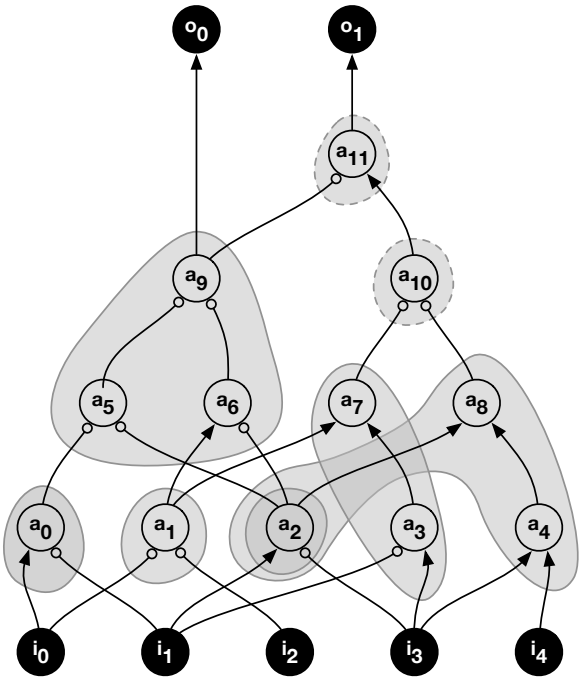


Figure 8.13: AIG of the toy circuit from Figure 8.9 with the resulting cones of technology mapping to AND gates and 3-input LUTs. The cones drawn with a dashed line are supposed to be implemented by AND gates and cones drawn with a solid line are to be implemented by 3-LUTs.

nodes is selected. Here the lowest depth of the node ('inverted' or 'non inverted') can be chosen as the inputs of a LUT can be inverted. Then the maximal depth is incremented. If the cut represents an AND gate, take the appropriate depth of both inputs: if the input edge is inverted, the 'inverted depth' of the input node is taken. Otherwise, the 'non inverted depth' is considered. We proved that this algorithm yields a depth-optimal solution. An example of the cut ranking is given in Table 8.3. For each node the 'inverted' and the 'non inverted depth' is listed.

Table 8.3: Intermediate results of the mapping of the AIG to 2-input AND gates and 3-input LUTs in Figure 8.13.

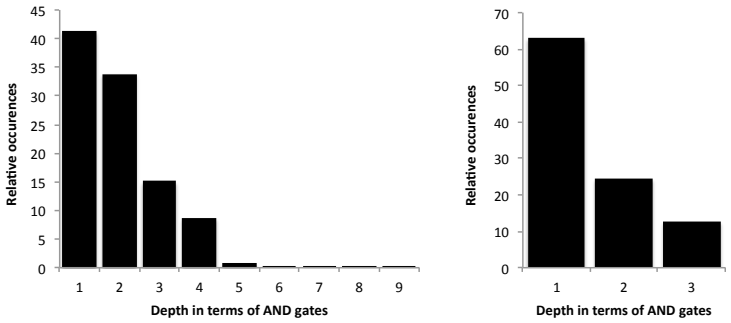
Node		K-feasible cuts with best cut in bold	Best cut			
n	Fanout	$\Phi(n)$	$depth(n, bc)$		$area\_flow(n, bc)$	
			non-inv	inv	non-inv	inv
a_0	1	$\{\{a_0\}, \underline{\mathbf{\{i_0 i_1\}}}\}$	1	1	1	1
a_1	2	$\{\{a_1\}, \underline{\mathbf{\{i_0 i_2\}}}\}$	1	1	1	1
a_2	3	$\{\{a_2\}, \underline{\mathbf{\{i_1 i_3\}}}\}$	1	1	1	1
a_3	1	$\{\{a_3\}, \underline{\mathbf{\{i_1 i_3\}}}\}$	1	1	1	1
a_4	1	$\{\{a_4\}, \underline{\mathbf{\{i_3 i_4\}}}\}$	0	1	0	1
a_5	1	$\{\{a_5\}, \{a_0 a_2\}, \{a_0 i_1 i_3\}, \{a_2 i_0 i_1\}, \underline{\mathbf{\{i_0 i_1 i_3\}}}\}$	1	1	1	1
a_6	1	$\{\{a_6\}, \underline{\{a_1 a_2\}}, \{a_1 i_1 i_3\}, \mathbf{\{a_2 i_0 i_2\}}\}$	1	2	5/6	4/3
a_7	1	$\{\{a_7\}, \underline{\{a_1 a_3\}}, \mathbf{\{a_1 i_1 i_3\}}, \{a_3 i_0 i_2\}\}$	1	2	3/2	3/2
a_8	1	$\{\{a_8\}, \{a_2 a_4\}, \underline{\mathbf{\{i_1 i_3 i_4\}}}, \{a_2 i_3 i_4\}, \{a_4 i_1 i_3\}\}$	1	1	1	1
a_9	2	$\{\{a_9\}, \underline{\mathbf{\{a_5 a_6\}}}, \underline{\mathbf{\{a_5 a_1 a_2\}}}, \{a_6 a_0 a_2\}, \underline{\mathbf{\{a_0 a_1 a_2\}}}\}$	2	2	17/6	17/6
a_{10}	1	$\{\{a_{10}\}, \mathbf{\{a_7 a_8\}}, \{a_7 a_2 a_4\}, \underline{\mathbf{\{a_8 a_1 a_3\}}}\}$	2	2	7/2	7/2
a_{11}	1	$\{\{a_{11}\}, \underline{\mathbf{\{a_9 a_{10}\}}}, \{a_9 a_7 a_8\}, \{a_5 a_6 a_{10}\}\}$	2	3	59/12	71/12
o_0	1	$\{\mathbf{\{a_9\}}\}$	2			
o_1	1	$\{\mathbf{\{a_{11}\}}\}$	2			

The best cut for the non inverted depth is emphasized with a bold font and the best cut for the inverted depth is underlined.

8.5.2 Cut Selection and Area Recovery

The cut selection decides which nodes are implemented. If a node is implemented, it is marked 'visible'. In our case nodes have to be set 'inverted visible', 'non inverted visible' or even both (in 1.2% of the cases on average). The new algorithm sets the output nodes 'non inverted visible' and traverses the network starting from the outputs towards the inputs. If a node with an AND gate implementation is visible, the inputs of the AND gate are set 'non inverted visible' or 'inverted visible', based on the inversion of the edges in the AIG. For a node with a LUT implementation the inputs of the cut are set 'visible' without further specification. Later on the inversion can be updated to a specific state.

In conventional depth optimal mapping there is only one implementation per node. In the case of mapping to LUTs and AND gates it is possible that a node is implemented twice. We directly adopted the implementation of the area recovery of [102], neglecting this difference. Nevertheless the area recovery still has a high performance: the state-of-the-art area recovery yields a reduction of 17% in number of 6-LUTs on average for the VTR benchmarks. Area recovery for mapping with AND gates yields an average reduction of 24% in number of 6-LUTs.



(a) Unlimited levels (b) Max 3 levels

Figure 8.14: Histogram for the depth of the AND gate network of each input pin

8.5.3 Area and Depth

Extra low-delay logic is added to the solution space, consequently the new circuit depth will always be equal or lower than the original circuit depth. This can be observed in Table 8.4, the geometrical mean of the depth decreases with 21.3% for the VTR benchmarks [88] for mapping to 6-LUTs and AND gates. For the MCNC20 benchmarks an average depth reduction of 32% is reported. The depth-optimal algorithm focuses on depth, therefore the area decreases only slightly with a decrease in geometrical mean of 5.7% for the number of 6-input LUTs, but with a standard deviation of 12.2%, there are some benchmark designs which have an increase in number of LUTs. By implementing AND cuts, the inputs of the gate must be implemented. It often occurs that this increases the number of LUTs locally. The area reduction and the logical depth decrease amounts to an on average 26% decrease in area-delay product.

The number of AND gates in the mapped circuit is on average 22% compared with the number of 6-LUTs for the VTR benchmarks. This low number will result in a relatively low overhead for the FPGA architecture. If we consider an FPGA architecture in which all MUXes in the SB, CB and LI are replaced, then this architecture will have 42 times more candidate routing nodes than LUTs, then only one gate out of 168 physical gates would be used. We expect that it is possible to lower the number of routing nodes with an AND gate compared in the architecture without losing too many degrees of freedom.

Table 8.4: The number of #6-luts, AND gates and the logic depth for mapping the VTR benchmarks to LUTs and AND gates. The number of AND gates is also reported relative to the number of LUTs in the new architecture. The area-delay product (A*D) is calculated as the product of the number of LUTs and the logic depth.

	#LUTs			**#AND2**		**Logic Depth**			**A*D**
Circuit	orig.	new	(%)	gates	(%)	orig.	new	(%)	(%)
bgm	30766	29521	-4	17097	58	34	23	-32,4	-35,1
blob_merge	6171	6196	0,4	1138	18	18	15	-16,7	-16,4
boundtop	2911	2216	-23,9	1139	51	9	7	-22,2	-40,8
ch_intrinsics	392	286	-27	147	51	3	2	-33,3	-51,3
diffeq1	474	489	3,2	95	19	15	13	-13,3	-10,5
diffeq2	325	329	1,2	93	28	14	13	-7,1	-6
LU8PEEng	22050	23058	4,6	6610	29	158	122	-22,8	-19,2
LU32PEEng	73963	72281	-2,3	14085	19	158	122	-22,8	-24,6
mcml	95886	110383	15,1	14586	13	116	99	-14,7	-1,8
mkDelayWorker	5197	4481	-13,8	1122	25	8	5	-37,5	-46,1
mkPktMerge	225	228	1,3	16	7	3	2	-33,3	-32,4
mkSMAdapter	2066	1642	-20,5	594	36	7	5	-28,6	-43,2
or1200	2916	2887	-1	550	19	27	23	-14,8	-15,7
raygentop	2048	1550	-24,3	754	49	8	7	-12,5	-33,8
sha	2197	2055	-6,5	456	22	21	18	-14,3	-19,9
stereovision0	11212	11220	0,1	1203	11	6	4	-33,3	-33,2
stereovision1	10172	10035	-1,3	794	8	7	7	0	-1,3
stereovision2	28423	32700	15	3836	12	17	15	-11,8	1,4
stereovision3	165	148	-10,3	34	23	4	3	-25	-32,7
Geomean	3768	3552	**-5,7**		22	15	12	**-21,3**	**-26**
Std. Dev			**12,2**					**10,4**	**16,1**

8.5.4 AND Gate Chains

Another important metric to consider is the number of subsequent AND gates between two blocks (LUTs, FFs, IOBs, ...). Fig. 8.14a contains a histogram for the depth of the AND network for each input pin that is preceded by AND gates. With 41%, the majority of input pins connected to an AND network have a one level AND network, 35% have a two level AND network and 15% a three level AND network. This means the majority of the AND gate networks have a very low depth and thus chains of AND2 gates rarely occur. The longest chain of AND2 gates that can be found has 9 subsequent AND2 gates. For these few longer chains our assumption that AND gates have negligible delay does not hold, because these long chains put more stringent constraints on the pack, placement and routing process. So we modified the technology mapping algorithm described in the previous sections to be able to limit the number of subsequent AND gates. Before we start our explanation we refer the reader to Figure 8.2 from the background section. The figure has a schematic of the logic cluster tile and it allows the reader to follow the routing pathways for the connections. The number of subsequent AND gates was set to three, two and one levels. Three levels because inter-CLB connections in the proposed architectures have to pass through at least three routing nodes and one level because intra-CLB connections have to pass through one routing node. In the best case a connection between LUTs in the same logic cluster will be implemented by the feedback path provided inside the cluster, so in that case, the connection will be routed through one LI MUX. A connection between LUTs in different logic clusters will be routed through at least 3 MUXes, one or more SB MUX, one CB MUX and one LI MUX. The new architectures have one AND2 gate per MUX, so AND gate networks with up to three levels could be implemented without making detours and putting too much constraints on the placement of the design. The results for limiting the number of subsequent AND2 gates are listed in Table 8.5 and they fit our expectations. Limiting the number of subsequent AND2 gates to 3 only leads to a 2.6% decrease in logic depth gain. Unfortunately we lose the area gain, we now have an overhead of 2.5% in terms of number of LUT gates. As expected the logic depth gain gradually decreases to a 14% decrease and the area overhead increases to a 3.9% increase if we further limit the subsequent AND2 gates to 1.

Table 8.5: Technology mapping results for restricting the AND2 gates in a chain.

Max #	LD	(%)	LUTs	(%)	A*D	(%)	AND2s	(%)
∞	11.8	-21.3	3552	-5.7	42e3	-25.8	784	22.1
3	12.2	-18.7	3863	+2.5	47e3	-16.6	813	21.0
2	12.3	-18.0	3921	+4.1	48e3	-14.7	667	16.0
1	12.9	-14.0	3916	+3.9	51e3	-10.6	390	10.0
0	15.0		3768		56e3			

All results are reported relative to only mapping to LUTs, which is the last row in the table, except for the percentage AND2 gates in the rightmost column which is reported relative to the number of LUTs in the circuit. (LD = Logic Depth, A*D = area-delay product)

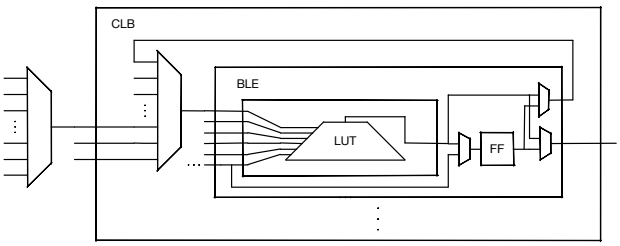
Listing 8.3: Definition of the new local interconnect routing node in the architecture description for VPR

```
1  <pb_type name="mxand" num_pb="6">
2   <input name="in" num_pins="2"/>
3   <output name="out" num_pins="1"/>
4   <mode name="multiplexer">
5    <interconnect>
6     <mux name="mux2and" input="mxand.in[0] mxand.in[1]" output="mxand.out"/>
7    </interconnect>
8   </mode>
9   <mode name="and">
10   <pb_type name="and" blif_model=".subckt and" num_pb="1">
11    <input name="in" num_pins="2"/>
12    <output name="out" num_pins="1"/>
13    <delay_constant max="50.0e−12" in_port="and.in" out_port="and.out"/>
14   </pb_type>
15   <interconnect>
16    <direct name="inand" input="liand.in" output="and.in"/>
17    <direct name="outand" input="and.out" output="liand.out"/>
18   </interconnect>
19  </mode>
20 </pb_type>
```

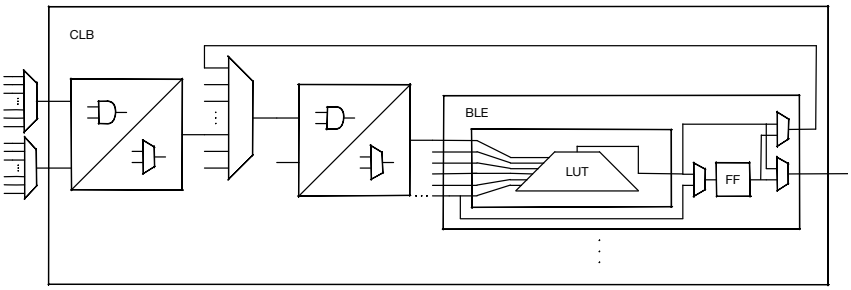
8.6 Packing

8.6.1 Modeling the Architecture

We use VPR to model the new architecture. VPR is built to handle very diverse architectures. An architecture is described hierarchically. Each element of the architecture can be described as a module via the *pb_type* tag. A module typically contains one or more child modules. The interconnection between these child modules and the I/O of the module is described between the interconnection tags. A module can have different modes. For example a DSP block can have a multiplier only mode and a multiplier and accumulate mode. Modes are also used



(a) Baseline



(b) New architecture with AND gates after the local interconnect and connection block multiplexer

Figure 8.15: Schematic of the different blocks described in the architecture file.

to describe a multiplier block that can be fractured in several smaller width multipliers. Different aspect ratios for RAM modules are also described via different modes.

In the new architecture a new basic building block is introduced. In Listing 8.3 the new basic building block is described with VPR's architecture description format. The *mxand* block has two inputs and one output. The block has two modes. In the first mode the block acts like a 2:1 multiplexer and in the second mode the block implements an 2-input AND gate. This basic building block is used to build the architecture. In Figure 8.15 a schematic is depicted for the baseline and the new architecture. It shows how the architecture is built up from basic building blocks. Each rectangle in the schematics correspond to a module. In the new architecture the *mxand* block is introduced after the connection block and local interconnect multiplexers.

Restrictions and Opportunities Since we limit ourselves to AND gates in the local interconnect and connection block multiplexer

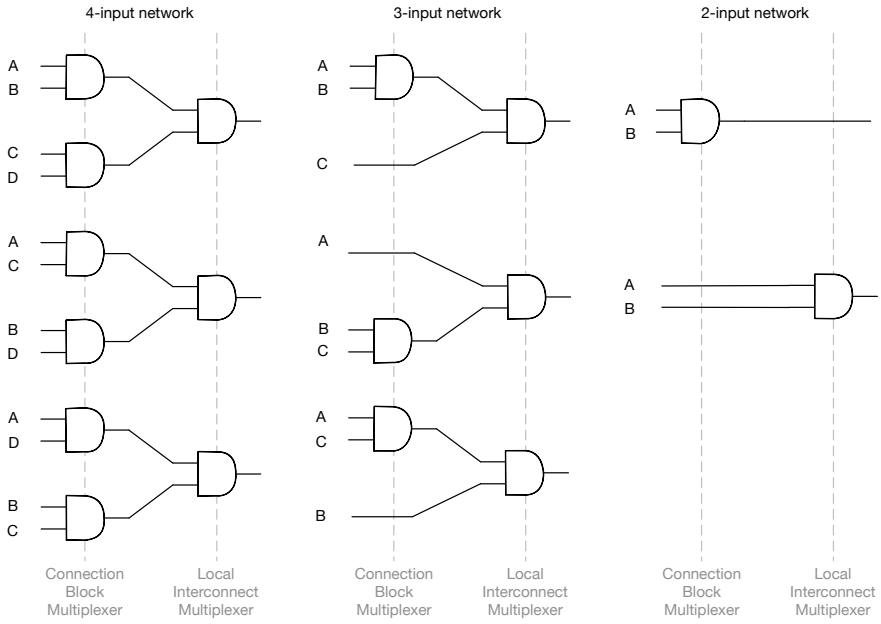


Figure 8.16: The different degrees of freedom for implementing AND networks in the logic cluster .

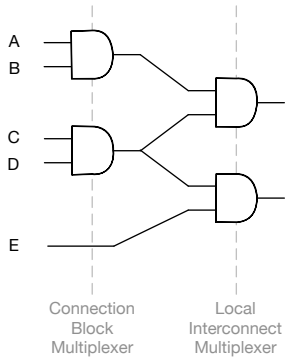


Figure 8.17: Two AND networks can share an intermediate signal, which leads to fewer pins being used to implement both networks.

mapped circuits, the architecture only has 2-level AND gates and LUTs. The first restriction is that the second level AND gates have to be implemented in the local interconnect routing nodes. These AND gates can only have a fanout of one, because the output of the local interconnect routing nodes is fed directly to the BLE input. Consequently, second level AND gates with a fanout higher than one in the input circuit should be duplicated.

Another restriction is that the AND networks with a feedback input signals need to be implemented in the local interconnect multiplexer. In case one of the inputs of the AND network is produced by a block that is already packed in the cluster, then the feedback signal can only be combined in the local interconnect multiplexers. A 2-input AND network with one or two feedback input signals has to be implemented in the local interconnect multiplexer. A 3-input AND network can only have one feedback input signal and a 4-input AND network should not have a feedback input signal, because the local interconnect and connection block routing nodes are necessary to implement the 4-input AND gate.

Next to these restrictions, there are also a few opportunities to optimize the input circuit. The associative and commutative property of the AND gate networks allows some flexibility on how to implement AND networks and decide which signals are combined in the first level. In Figure 8.16 the different ways to implement 4, 3 and 2-input AND networks are depicted.

Another opportunity is intermediate signal sharing. In Figure 8.17 a 4-input and a 3-input AND network are depicted. They share the intermediate signal after the AND gate that combines input C and D. In that way signal sharing can lead to tighter packing results.

To be able to handle these restrictions and opportunities, we slightly modified the packing algorithm. The AND networks are resynthesized during the cluster feasibility check in order to produce tighter packing results. We used a greedy algorithm to resynthesize the AND networks, because the cluster feasibility check is part of the inner loop of the packing algorithm. In order to explain our modification we start with a brief description of the conventional packing algorithms.

8.6.2 Conventional Packing

Conventional packing algorithms can be divided into two main classes, seed based and partitioning based packers. The packer implemented in VPR is based on the seed based packer AAPack, which was described

first in [91]. We focus on AAPack, because we use VPR to model our new architecture.

Seed-based Packing Seed based packing is a greedy algorithm. It starts with an unpacked elementary block called the seed. In AAPack this seed is the unpacked block with the highest number of nets attached. Subsequently an affinity metric between the seed block and its surrounding blocks is calculated. Several cost functions for the affinity metric are proposed by various authors [88, 20, 121] and are designed to improve the quality for a specific optimisation criterium. The block that scores the highest on this affinity metric is considered as a candidate to be packed into the cluster. Next the block is checked if it can be legally packed into the cluster. Once the cluster is full, the algorithm "closes" that cluster and opens up a new cluster to be filled by selecting a new seed block. In AAPack, for example, the new seed is the unpacked block with the most nets attached. Packing finishes when all blocks in the netlist have been packed into clusters.

Cluster Feasibility Check All elementary blocks in the cluster are placed and the intra cluster connections are routed in order to check if an elementary block can be legally packed in a cluster. This part of the packing algorithm will be slightly modified to enable VPR to pack circuits with AND gates more efficiently.

Speculative Packing In the process of filling the cluster, the packer does not do a complete feasibility check, but only a fast check based on the easy to calculate feasibility statistics, such as the number of cluster inputs/outputs and the number of blocks of each type. Once a cluster is completely filled the extensive feasibility check is performed. If it succeeds, a lot of runtime is saved. In case the final feasibility check fails, the cluster is repacked with the extensive cluster feasibility check for each addition.

8.6.3 Resynthesis during Cluster Feasibility Check

For the new architecture we extended the cluster feasibility check with a resynthesis step. Each time a block is considered as candidate to add to the cluster, the AND networks attached to the blocks inside the cluster are resynthesized in order to maximize signal sharing. In order to decide which intermediate signals are chosen to be implemented, a greedy algorithm is used. We will first give a high level overview of the algorithm and then apply it to an example. In the first step of

the resynthesis algorithm a histogram of all the possible intermediate signals is built. For each input AND network several different intermediate signals can be used to implement the network. The intermediate signals for the example networks from Figure 8.16 are listed here:

- 4-input AND network with inputs A,B,C,D:
  AB, AC, AD, BC, BD, CD
- 3-input AND networks with inputs A, B, C:
  A, BC, B, AC, AB, C
- 2-input networks with inputs A, B:
  A, B, AB

The second step of the algorithm consists of a decision loop. In each iteration all the possibilities for each AND input network are ranked and the highest scoring option is chosen and implemented. Subsequently the histogram is adapted accordingly. All the possibilities for each of the example networks in Figure 8.16 are listed here:

- 4-input AND networks: AB-CD, AC-BD, AD-BC
- 3-input AND networks: A-BC, AB-C, AC-B
- 2-input networks: A-B, AB

The decision loop ends when for each network a possibility is chosen and no decisions need to be made anymore.

The resynthesis algorithm is applied to the example in Table 8.6. The example consists of a cluster with 3 LUTs. The AND networks for each LUT input are listed in the first column of Table 8.6. Each net or signal is represented by a single character. There are 9 different nets from *a* to *i*. In Table 8.7 the histogram with all the possible intermediate signals and their ranking scores are listed. Before the iteration loop starts, there are a few preliminary decisions that have to be made. The preliminary decisions are shown in the second column of Table 8.6, marked with the decision loop number 0 as a header. There are two types of preliminary decisions. The first type is input networks that only have one signal, they are implemented as intermediary signals. These are regular inputs to the cluster and they have to be available to the local interconnect multiplexers. Examples in Table 8.6 are net *a* and *c* as inputs to *LUT 0* and net *i* to *LUT 2*. The second type of decisions are forced by the fact that feedback paths are only connected to the local interconnect multiplexers. Examples in Table 8.6 are *k-o* for *LUT 1* and *ij-k*, *k-ef*, *im-k* for *LUT 2*, because signal *k* is an output of *LUT 0* and *o* of *LUT 2*.

Table 8.6: Decision table for the resynthesis of a cluster with 3 LUTs. The input networks for each LUT are listed in the first column. This table corresponds with the histogram in Table 8.7.

Decision loop #	0	1	2	3	4	5	6	7
Input Netwerk								
LUT0 - output net: k								
a	a	-	-	-	-	-	-	-
c	c	-	-	-	-	-	-	-
abcd	5|2|3	5|2|3	5|2|3	**5**|2|3	**ab-cd**	-	-	-
ad	**7**|2	**6**|2	5|2	4|2	4|1	**4**|1	**a-d**	-
LUT1 - output net: n								
cde	5|6|5	4|**6**|5	**cd-e**	-	-	-	-	-
cdf	5|**7**|5	**cd-f**	-	-	-	-	-	-
aef	5|4|3	5|4|3	5|4|3	**5**|4|3	**5**|4|3	**a-ef**	-	-
fgh	4|2|2	4|2|2	4|2|2	4|2|2	4|2|2	3|2|2	**3**|2|2	f-gh
ko	k-o	-	-	-	-	-	-	
LUT2 - output net: o								
ijk	ij-k	-	-	-	-	-	-	-
kef	k-ef	-	-	-	-	-	-	-
imk	im-k	-	-	-	-	-	-	-
i	i	-	-	-	-	-	-	-
cdi	5|6|5	4|**6**|4	3|**6**|3	**cd-i**	-	-	-	-

Next the decision loop is started. The highest ranking possibilities in iteration 1 are *a-d* and *cd-f* with a score of seven. *cd-f* is chosen because it has the highest number of inputs. The histogram in Table 8.7 is adapted by subtracting the counter for the intermediary signals that are not implemented by taking this decision, which is *c*, *d*, *cf* and *df* in our case. In the second iteration we have three intermediary signals and of these three signals *cd-e* and *cd-i* are ranked highest and have the same number of inputs. In this case we just implement the first one. The subsequent decisions are made in a similar manner. The algorithm ends after the implementation for each AND network in the cluster is decided. The resulting cluster uses 6 regular inputs and 12 inputs to realize 6 AND gates in the connection block multiplexers

8.6.4 Performance Improvement

Without resynthesis during clustering, the post route performance is worse than the baseline for the architecture with AND2 gates in the local interconnect and connection block routing nodes. The critical path delay is similar with a 1% improvement, but there is a 25% area overhead in terms of CLBs on average. Resynthesizing AND networks during packing allows us to recover some of the area overhead, but a considerable area overhead remains. The area overhead reduces to 11%. The average critical path delay improves greatly to a 9% decrease. In

Table 8.7: The changes made to the histogram of intermediate signals during the decision loop of the resynthesis algorithm applied on the example from Table 8.6

Decision loop #	0	1	2	3	4	5	6	7
Intermediate signal								
a	3	=	=	=	=	=	=	=
c	4	3	2	1	=	=	=	=
ab	1	=	=	=	=	=	=	=
ac	1	=	=	=	0	-	-	-
ad	2	=	=	=	1	=	0	-
bc	1	=	=	=	0	-	-	-
bd	1	=	=	=	0	-	-	-
cd	4	=	=	=	=	=	=	=
d	4	3	2	1	=	=	=	=
e	2	=	=	=	=	1	=	=
ce	1	=	0	-	-	-	-	-
de	1	=	0	-	-	-	-	-
f	3	=	=	=	=	2	=	=
cf	1	0	-	-	-	-	-	-
df	1	0	-	-	-	-	-	-
ae	1	=	=	=	=	0	-	-
af	1	=	=	=	=	0	-	-
ef	2	=	=	=	=	=	=	=
g	1	=	=	=	=	=	=	0
h	1	=	=	=	=	=	=	0
fg	1	=	=	=	=	=	=	0
fh	1	=	=	=	=	=	=	0
gh	1	=	=	=	=	=	=	=
ij	1	=	=	=	=	=	=	=
im	1	=	=	=	=	=	=	=
i	2	=	=	=	=	=	=	=
ci	1	=	=	0	-	-	-	-
di	1	=	=	0	-	-	-	-
Number of regular inputs:								6
Number of AND2:								6

An equal sign indicates that the count for the specific intermediate signal is not changed in this iteration.

the next section the results with the modified pack algorithm are discussed in more detail.

8.7 Post-route Performance

The VTR benchmark designs are mapped while restricting the mapping algorithm from Section 8.5 to only one or two AND2 gates. The resulting netlists are packed by the modified packing algorithm described in the previous section. Subsequently, the packed designs are placed and routed with the default settings in VPR. The architecture used in COFFE [30] is used as a baseline. See Section 8.2.2 for a more detailed

Table 8.8: Post route performance of the new architectures

	LI Muxes only		LI and CB Muxes	
Circuit	CPD	Area (#CLBs)	CPD	Area (#CLBs)
bgm	1,02	1,40	0,96	1,48
blob_merge	0,93	1,00	0,81	1,02
boundtop	0,96	1,05	0,88	1,08
ch_intrinsics	1,12	0,91	1,06	0,96
diffeq1	0,89	1,20	0,86	1,29
diffeq2	1,03	1,36	0,99	1,41
LU8PEEng	1,05	1,04	0,95	1,09
LU32PEEng	1,03	1,08	1,02	1,16
mcml	0,99	1,29	0,97	1,30
mkDelayWorker32B	0,81	0,99	0,78	1,03
mkPktMerge	0,83	0,99	0,81	1,01
mkSMAdapter4B	0,74	1,06	0,63	1,13
or1200	0,84	1,04	0,82	1,10
raygentop	0,86	1,14	0,76	1,15
sha	0,96	1,19	0,94	1,22
stereovision0	0,87	1,02	0,79	0,98
stereovision1	1,00	0,96	1,00	0,96
stereovision2	0,96	1,05	0,87	1,07
stereovision3	1,88	0,84	1,86	0,86
Geomean	**0,97**	**1,08**	**0,91**	**1,11**
Std. Dev.	**0,06**	**0,02**	**0,06**	**0,03**

CPD = Critical Path Delay

All results are reported relative to the baseline architecture without extra gates in the routing nodes

description. The description is available in the VTR project [88] under the name *k6_N10_gate_boost_0,2V_22nm*. New descriptions were made for the new architectures with new blocks, see Listing 8.3 and the delays are taken from the sizing results, see Section 8.3. The final post-routing results are reported relative to the baseline. The ratios are listed in Table 8.8.

The results show a clear trend. The new architecture performs slightly better in terms of speed-performance at the cost of a larger area overhead. For the architecture with the AND2 gates in LI routing nodes only the critical path delay improves with 3% on average and the number of CLBs increases with 8%. The same applies to the architecture with AND2 gates in the LI and SB routing nodes, but more pronounced. The critical path delay improves with 9% at an area overhead cost of 11%. For both architectures the standard deviation is similar. The critical path delay improves for the majority of the benchmarks, but for some designs the critical path delay worsens. This is reflected in the standard deviation of 6% for the critical path delay. The area overhead is more consistent with a standard deviation of 2-3%

In section 8.3.2 we described that the maximum architectures are not feasible, because of the large tile area overhead. The minimum architectures have less tile area overhead, but less flexibililty to combine signals. Using the minimum architectures instead of the maximum architectures comes at a cost and that is clearly observable if we look at the post route area overhead. There is a higher area overhead in terms of number of CLBs than in terms of LUTs, see Table 8.5 and 8.8. For restricting the AND2 gates in series to only one AND2 gate the technology mapping algorithm achieved solutions with a 3.9% increase in LUTs on average, but the post routing area overhead in terms of CLBs increases to 7%. For restricting the AND2 gates in series to two AND2 gates the technology mapping algorithm achieved solutions with a 4.1% increase in LUTs on average, but the post routing area overhead in terms of CLBs increases to 11%.

8.8 Concluding Remarks

We investigated a new kind of FPGA architecture with routing nodes that contain N:2 instead of N:1 multiplexers, and an AND or OR gate. The new routing nodes have two modi. They are able to pass through signals like a conventional routing node or apply an AND or OR operation on two signals. By sizing the tile of the new architectures and developing a new technology mapping algorithm for the new architecture we are able to estimate the performance of the new architecture.

Adding gates to the routing nodes increased the area of the tile but the gates in the routing nodes are used to reduce the logic levels in the circuits. The minimum architecture with OR gates in the local interconnect and/or connection block routing nodes leads to the most promising result. To further investigate these architectures we used VPR to model these architectures and pack, place and route the benchmark designs. The post-route performance of the designs on the new architecture improves in term of critical path delay with a 3-9% decrease in post-route critical path delay. Unfortunately the large depth reduction of 14-18% achieved during technology mapping doesn't completely translate post-routing. Additionally the downside of these architectures is the large area overhead with a 7-11% increase in number of CLBs and a 8-11% increase in tile area.

For the main FPGA manufacturers, the relatively small increase in maximum clock frequency probably will not justify the larger area overhead, but this technique may be applicable for niche products in which speed performance is the highest good. To decrease the area overhead of the logic gates, we would like to investigate depopulation in the future. The architecture does not need extra gates in every local interconnect block and connection block multiplexer.

We think improvements are possible if we would implement a hierarchical packing algorithm for these architectures. Seed-based packing greedily selects the most attracted neighbour as a candidate to include in the cluster, which leads to suboptimal solutions because the solution is easily stuck in a local minimum. The new architecture suffers more from seed based packing, because the clusters are more difficult to pack. In the future we want to apply hierarchical packing strategies discussed in Chapter 4 to improve performance. At the moment of writing we are still improving our hierarchical packing algorithms.

We are also interested in how the results would change if we consider Altera's Stratix 10 fabric. Altera has inserted simple latches in the routing network of the Stratix 10 FPGAs

Lastly, we noticed that investigating new architectures requires a lot of engineering work. The CAD tool flow has to be modified and optimised for the new target architecture. The new architectures have to be sized to extract wire delays and area usage. An additional hurdle is the computational intensity involved with sizing new architectures. This is a significant problem and in future work we want to investigate how we can shorten this hardware design cycle.

9 Conclusions and Future Work

We start with repeating the fundamental problems we described in the introduction and the goals we proposed. The first problem is the slow FPGA design cycle. The bottleneck of the design cycle is FPGA compilation. To shorten the design cycle the goal is to introduce new compilation techniques to speed up FPGA compilation. The second problem is the gap between the academic and commercial results. The goal was to investigate the gap and find out what we can do to reduce it as much as possible. We also tried to improve the efficiency of FPGA devices by investigating new architectures and micro-reconfiguration.

9.1 Conclusions

In this section, I present the overall conclusions of this dissertation and summarise how the goals of this dissertation were achieved. The conclusions of each chapter separately can be found at the end of the respective chapters.

9.1.1 The Gap between the Academic and Commercial Results

We measured the gap between academic and commercial results and found it considerably large. Certainly for speed-performance and runtime there is more than a factor of 2x difference in results. We hope

by measuring and publishing about this divide that we increase the awareness amongst other academic researchers.

By introducing new compilation techniques we effectively reduced the gap, but we did not close it and we predict that the gap will never close completely. Academics will always lag behind and there are clear causes for this statement. FPGA vendors have capital for hiring more research engineers to investigate new tool and architecture ideas. They also have access to sensitive information that researchers don't have. The designs of their customers and the parameters and characteristics of the latest process technologies are sensitive information FPGA vendors have at their disposal, but can't share with the academic community because they don't own it. Nevertheless we have to strive to keep the academic work relevant in this area. We suggest to try to use hybrid evaluation frameworks as much as possible. We used hybrid frameworks in Chapter 4, Chapter 5 and Chapter 6.

9.1.2 New FPGA Compilation Techniques

We successfully shortened the design cycle by speeding up the compilation. We described new techniques for the packing, placement and routing steps of the compilation flow. The hierarchical multi-threaded packing approach from Chapter 4 reached speedup factors of 3.6 on a four core CPU while reducing wiring and reducing routing runtime by half. We reduced the placement time by half with our steepest gradient descent placement technique, LIQUID from Chapter 5. The routing runtime is further reduced by 23% by the connection-based routing mechanism from Chapter 5.

9.1.3 Dynamic Reconfiguration of the Routing Network

We designed place and route techniques for the dynamic reconfiguration of the routing network. In this way we contributed to an easy to use compilation flow for micro-reconfiguration. The flow exploits both the reconfigurablility of the LUTs and the routing network in an automatic fashion by compiling parameterized configurations. The designer only has to annotate which input signals in his design should be processed as parameters. By doing so, we hope to unlock the potential of dynamically reconfiguring the routing network of the FPGA. The main problems that remain are the limited commercial tool support. There is no public information about the low level configuration details and timing characteristics of commercial devices, which prevents us to generate competitive solutions.

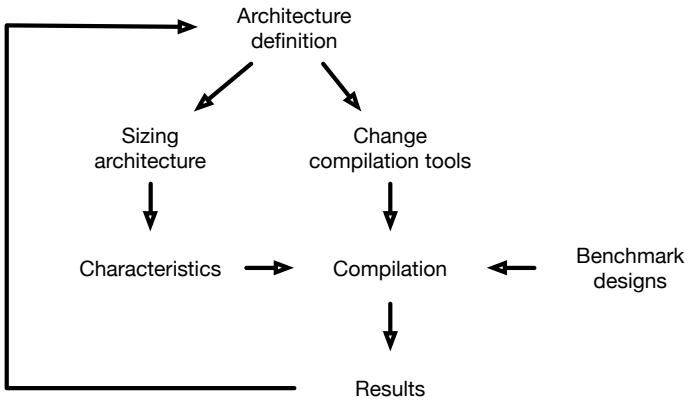


Figure 9.1: The FPGA architecture design cycle.

9.1.4 FPGA Architectures with Logic Gates in the Routing Network

Although mapping results seemed very promising at first glance, an architecture with logic gates in the routing network is not unilaterally better. It performs slightly better in speed-performance and power consumption at the cost of a substantial area increase. It is therefore not a cost effective architecture. The most important takeaway from the architecture investigation was the observation that the investigation process is slow and tedious. We observed the FPGA architecture design cycle, which is illustrated in Figure 9.1 and not to be confused with the FPGA design cycle. An architecture definition is where the cycle starts. The architecture needs to be sized with the help of several electronic circuit simulations. Several characteristics are extracted, such as delay, area and power consumption of the basic blocks. The compilation tools need to be changed to be able to compile designs to the new architectures. Once these two steps are finished the architecture can be tested by compiling benchmark designs. The results will indicate the performance of the new architecture and suggest what to change in the architecture to improve performance. Almost all steps in this cycle are time consuming and prohibit a wide exploration.

9.2 Future Work

I finish this chapter with a number of suggestions for the future of research on FPGA compilation techniques and architectures.

9.2.1 Further Acceleration of the FPGA Compilation

The new compilation techniques proposed in this thesis are in their infancy and are not completely fine-tuned in comparison with the conventional compilation techniques which have been refined for many years. There are also some opportunities in better integrating the compilation steps. One example is using the partitioning tree information from the hierarchical partitioning based packer from Chapter 4 to quickly build a placement prototype and use the placement algorithm from Chapter 5, LIQUID, to further optimize the placement prototype. Another example is soft packing. Soft packing is a technique in which the placer is allowed to move primitives from one packed cluster to another if it improves the placement.

To achieve very short compilation runtimes researchers try to use GPU acceleration. LIQUID is an excellent candidate for GPU acceleration, but now that routing becomes the main bottleneck it is paramount that we investigate how we can accelerate the routing process. Speeding up routing is not straightforward on a GPU because it is not designed to accelerate the pointer chasing operations that are common in routing algorithms.

We also predict that FPGA compilation will be performed mainly in the cloud in the near future. Setting up workstations for compilation is a hassle and a costly diversion for a lot of small companies, it is not their core business. A startup pioneer in this field is Plunify. The business model of this provider allows to design specific accelerators for the compilation techniques, because they have more interest in speeding up the compilation. It is more closely related to their product. There have already been articles published about machine learning techniques used to speed up the search for optimal compilation parameter values [113]. Now Plunify is a small player that still has to gain trust before the incumbent FPGA design companies will come on board. In case Xilinx or Intel Altera start with cloud services this could gain traction much faster, because the important customers already trust Altera/Xilinx to let them use their design to benchmark their tool flow and architecture. At the last moment of writing, Xilinx together with Amazon Web Services introduced a cloud instance with UltraScale+ VU9P FPGAs, instant access to the Xilinx' design tools and a web store to sell FPGA designs. This allows startups to avoid the high hardware setup cost and the high license costs for the design tools.

9.2.2 Generic Method to Investigate New FPGA Architectures

In the future we want to investigate how we can shorten the FPGA architecture design cycle, as schematized in Figure 9.1. This was the main motivation for developing the partitioning based packer. We wanted to investigate the natural hierarchy levels in the benchmark circuits and map this natural hierarchy to our architecture automatically. In this way we don't follow the traditional cycle for the prototype of the architecture. The partitioning based packer turned out to be a more qualitative packing tool and it could be easily adapted to exploit the multi-core environment in commodity hardware. We plan to get back to our initial goal.

Bibliography

[1] ABC: A System for Sequential Synthesis and Verification. [Online] http://www.eecs.berkeley.edu/~alanmi/abc.

[2] AutoESL Acquisition a Great Move for Xilinx. http://www.eetimes.com/author.asp?section_id=36&doc_id=1284904, Jan. 2011. Accessed: 2016-01-10.

[3] Predictive Technology Model (PTM). http://ptm.asu.edu/, 2014.

[4] Technology Quarterly: After Moore's Law, March 2016. last accessed on 2 November 2016.

[5] A. M. A. Petkovska, D. Novo and P. Ienne. Constrained interpolation for guided logic synthesis. In *IEEE/ACM International Conference on Computer-Aided Design*, 2014.

[6] F. Abouelella, T. Davidson, W. Meeus, K. Bruneel, and D. Stroobandt. How to Efficiently Implement Dynamic Circuit Specialization Systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(3):35:1–35:38, 2013.

[7] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *IEEE TVLSI*, 12(3):288–298, March 2004.

[8] B. Al Farisi, E. Vansteenkiste, K. Bruneel, and D. Stroobandt. A novel tool flow for increased routing configuration similarity in multi-mode circuits. *Proceedings of the 50th Annual Design Automation Conference*, page to appear, 2013.

[9] Altera. Enabling High-Performance DSP Applications with Stratix V Variable-Precision DSP Blocks. White Paper, May 2011.

[10] Altera. White Paper: Implementing FPGA Design with the OpenCL Standard. https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf, November 2013.

[11] Altera Corporation. *Increasing Design Functionality with Partial and Dynamic Reconfiguration in 28-nm FPGAs*, 2010.

[12] Altera Corporation. Design Planning for Partial Reconfiguration. 2013.

[13] Altera Corporation. *Stratix V Device Handbook Volume 1: Device Interfaces and Integration*, January 2016.

[14] R. Arunya, S. Ranjith, P. Umarani, A. Ramya, and T. Ravi. Energy Efficient Multiplexer and De-multiplexer Using FINFET Technology. *Research Journal of Applied Sciences, Engineering and Technology*, 10(8):923–931, 2015.

[15] V. Betz. The FPGA Place-and-Route Challenge, Aug. 2011.

[16] V. Betz and J. Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *International Conference on Field-Programmable Logic and Applications (FPL), Proceedings of the*, pages 1–10, London, UK, 1997. Springer-Verlag.

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

[18] V. Betz, J. Swartz, and V. Gouterman. Method and apparatus for performing parallel routing using a multi-threaded routing procedure, Sept. 10 2013. US Patent 8,533,652.

[19] E. Bozorgzadeh, S. O. Memik, X. Yang, and M. Sarrafzadeh. Routability-driven packing: Metrics and algorithms for cluster-based FPGAs. *Journal of Circuits, Systems, and Computers*, 13(01):77–100, 2004.

[20] E. Bozorgzadeh, S. Ogrenci-Memik, and M. Sarrafzadeh. RPack: routability-driven packing for cluster-based FPGAs. In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 629–634. ACM, 2001.

[21] K. Bruneel, W. Heirman, and D. Stroobandt. Dynamic Data Folding with Parameterizable FPGA Configurations. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(4):43:1–43:29, 2011.

[22] P. K. Chan and M. D. F. Schlag. Parallel Placement for Field Programmable Gate Arrays. In *FPGA '03: Proceedings of the*

2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, pages 43–50, New York, NY, USA, 2003. ACM.

[23] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. Reducing Structural Bias in Technology Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(12):2894–2903, Dec. 2006.

[24] C. Chen, R. Parsa, N. Patil, S. Chong, K. Akarvardar, J. Provine, D. Lewis, J. Watt, R. T. Howe, H.-S. P. Wong, et al. Efficient FPGAs using nanoelectromechanical relays. In *FPGA*, pages 273–282. ACM/SIGDA, 2010.

[25] D. Chen and J. Cong. DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs. In *ICCAD*, pages 752–759. IEEE, 2004.

[26] D. T. Chen, K. Vorwerk, and A. Kennings. Improving timing-driven FPGA packing with physical information. In *2007 International Conference on Field Programmable Logic and Applications*, pages 117–123. IEEE, 2007.

[27] S.-Y. Chen and Y.-W. Chang. Routing-architecture-aware analytical placement for heterogeneous fpgas. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 27:1–27:6, New York, NY, USA, 2015. ACM.

[28] Y.-C. Chen, S.-Y. Chen, and Y.-W. Chang. Efficient and effective packing and analytical placement for large-scale heterogeneous FPGAs. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 647–654. IEEE Press, 2014.

[29] C.-L. E. Cheng. RISA: Accurate And Efficient Placement Routability Modeling. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Proceedings of the*, pages 690–695, Los Alamitos, CA, USA, 1994.

[30] C. Chiasson and V. Betz. COFFE: Fully-automated transistor sizing for FPGAs. In *ICFPT*, pages 34–41. IEEE, Dec 2013.

[31] C. Chiasson and V. Betz. Should FPGAs abandon the pass-gate? In *FPL*, pages 1–8. IEEE, 2013.

[32] S. A. Chin and J. H. Anderson. A case for hardened multiplexers in fpgas. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 42–49. IEEE, 2013.

[33] A. Choong, R. Beidas, and J. Zhu. Parallelizing simulated annealing-based placement using GPGPU. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 31–34. IEEE, 2010.

[34] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, XXXII:406–424, 1953.

[35] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(1):1–12, 1994.

[36] J. Cong and Y. Ding. On Area/Depth Trade-off in LUT-based FPGA Technology Mapping. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(2):137–148, 1994.

[37] J. Cong and M. Romesis. Performance-driven multi-level clustering with application to hierarchical FPGA mapping. In *Design Automation Conference, 2001. Proceedings*, pages 389–394. IEEE, 2001.

[38] A. Corporation. *Stratix IV Device Handbook*. Altera Corporation, 2008. Available online: `https://www.altera.com/en_US/pdfs/literature/hb/stratix-iv/stratix4_handbook.pdf`.

[39] M. E. Dehkordi and S. D. Brown. Performance-driven recursive multi-level clustering. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 262–269. IEEE, 2003.

[40] A. DeHon and N. Mehta. Exploiting partially defective luts: Why you don't need perfect fabrication. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 12–19, 2013.

[41] S. Dieleman, J. Schlueter, and D. Nouri. Lasagne: Lightweight library to build and train neural networks in Theano. `https://github.ugent.be/Lasagne`, 2014-2016.

[42] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[43] J. Divyasree, H. Rajashekar, and K. Varghese. Dynamically Reconfigurable Regular Expression Matching Architecture. *International Conference on Application-Specific Systems, Architectures and Processors*, pages 120–125, July 2008.

[44] L. Easwaran and A. Akoglu. Net-length-based routability-driven power-aware clustering. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):38, 2011.

[45] A. Farrahi and M. Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(11):1319–1332, Nov 1994.

[46] T. Feist. Vivado design suite. *White Paper*, 5, 2012.

[47] W. Feng. K-way partitioning based packing for FPGA logic blocks without input bandwidth constraint. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 8–15. IEEE, 2012.

[48] W. Feng, J. Greene, K. Vorwerk, V. Pevzner, and A. Kundu. Rent's rule based FPGA packing for routability optimization. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 31–34. ACM, 2014.

[49] C. Fobel, G. Grewal, and D. Stacey. A scalable, serially-equivalent, high-quality parallel placement methodology suitable for modern multicore and gpu architectures. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[50] P.-E. Gaillardon, X. Tang, G. Kim, and G. De Micheli. A novel fpga architecture based on ultrafine grain reconfigurable logic cells. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 23(10):2187–2197, 2015.

[51] M. Gort and J. Anderson. Accelerating FPGA Routing Through Parallelization and Engineering Enhancements Special Section on PAR-CAD 2010. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):61–74, Jan 2012.

[52] M. Gort and J. H. Anderson. Deterministic multi-core parallel routing for FPGAs. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 78–86. IEEE, 2010.

[53] M. Gort and J. H. Anderson. Analytical placement for heterogeneous fpgas. In D. Koch, S. Singh, and J. Torresen, editors, *FPL*, pages 143–150. IEEE, 2012.

[54] S. Gupta, J. H. Anderson, L. Farragher, and Q. Wang. CAD techniques for power optimization in Virtex-5 FPGAs. In *CICC*, pages 85–88. Citeseer, 2007.

[55] Hardware and E. S. group Computer Systems Lab ELIS department Ghent University. The TLUT tool flow, Oct. 2012.

[56] K. Heyse. *Improving the Gain and Reducing the Overhead of Dynamic Circuit Specialisation and Micro-reconfiguration*. PhD thesis, University of Ghent, Faculty of Engineering Science, 2015.

[57] K. Heyse, K. Bruneel, and D. Stroobandt. Mapping logic to reconfigurable FPGA routing. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 315 –321, aug. 2012.

[58] K. Heyse, T. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt. Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAs. *Proceedings of the 50th Annual Design Automation Conference*, page to appear, 2013.

[59] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In *ICCAD*, pages 381–384, 1986.

[60] E. Hung. Mind The (Synthesis) Gap: Examining Where Academic FPGA Tools Lag Behind Industry. In *25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[61] E. Hung, F. Eslami, and S. J. E. Wilton. Escaping the academic sandbox: Realizing VPR circuits on Xilinx devices. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21th Annual International Symposium on*. IEEE, 2013.

[62] International Business Strategies, Inc. Rising design costs. *Xcell Journal*, 88(3):10, 2014.

[63] J. Jain, V. Verma, T. Ahmed, S. Kalman, S. Kwatra, C. Kingsley, J. Anderson, and S. Das. Multi-threaded deterministic router, Mar. 11 2014. US Patent 8,671,379.

[64] S. Jang, B. Chan, K. Chung, and A. Mishchenko. WireMap: FPGA Technology Mapping for Improved Routability and Enhanced LUT Merging. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(2):1–24, 2009.

[65] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[66] A. Karpathy and F.-F. Li. CS231n Convolutional Neural Networks for Visual Recognition. `cs231n.github.io/neural-networks-3/#sgd`, 2015.

[67] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[68] G. Karypis and V. Kumar. hMETIS 1.5: A hypergraph partitioning package. Technical report, Technical report, Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL http://www. cs. umn. edu/metis, 1998.

[69] M.-C. Kim, D. Lee, and I. L. Markov. Simpl: An effective placement algorithm. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(1):50–60, 2012.

[70] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[71] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(3):356–365, 1991.

[72] M. Kudlur, V. Vasudevan, and I. Polosukhin. TensorFlow: Computation using data flow graphs for scalable machine learning. `https://github.ugent.be/tensorFlow` and `https://www.tensorflow.org/`, 2015-2016.

[73] I. Kuon and J. Rose. Exploring area and delay tradeoffs in fpgas with architecture and automated transistor design. *IEEE VLSI*, 19(1):71–84, 2011.

[74] M. Y. J. Lai and J. Tong. Yet another many-objective clustering (YAMO-Pack) for FPGA CAD. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.

[75] J. Lamoureux and S. J. Wilton. On the interaction between power-aware FPGA CAD algorithms. In *Proceedings of the 2003*

IEEE/ACM international conference on Computer-aided design, page 701. IEEE Computer Society, 2003.

[76] B. S. Landman and R. L. Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on computers*, 100(12):1469–1479, 1971.

[77] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *21st International Conference on Field-Programmable Logic and Applications (FPL), Proceedings of the*, pages 349–355, sept. 2011.

[78] C. Y. Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, Sept. 1961.

[79] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and Single-Driver Wires in FPGA Interconnect. In *International Conference on Field-Programmable Technology (FPT), Proceedings of the*, pages 41 – 48. IEEE, dec. 2004.

[80] D. Lewis, E. Ahmed, D. Cashman, T. Vanderhoek, C. Lane, A. Lee, and P. Pan. Architectural enhancements in Stratix-III™ and Stratix-IV™. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–42. ACM, 2009.

[81] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang. A high performance fpga-based accelerator for large-scale convolutional neural network. In *2016 International Conference on Field Programmable Logic and Applications*, pages 69–77. IEEE, 2016.

[82] J. Lien, S. Feng, E. Huang, C. Sun, T. Liu, and N. Liao. Tileable field-programmable gate array architecture, Nov. 2002. US Patent 6,476,636.

[83] T.-H. Lin, P. Banerjee, and Y.-W. Chang. An efficient and effective analytical placer for FPGAs. In *Proceedings of the 50th Annual Design Automation Conference*, page 10. ACM, 2013.

[84] H. Liu and A. Akoglu. Timing-driven nonuniform depopulation-based clustering. *International Journal of Reconfigurable Computing*, 2010:3, 2010.

[85] A. Ludwin and V. Betz. Efficient and Deterministic Parallel Placement for FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 16(3):22:1–22:23, June 2011.

[86] A. Ludwin, V. Betz, and K. Padalia. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 14–23. ACM, 2008.

[87] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014.

[88] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. volume 7, pages 6:1–6:30, June 2014.

[89] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, K. Kent, and J. Rose. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):32:1–32:23, Dec. 2011.

[90] J. Luu, J. Rose, and J. Anderson. Towards interconnect-adaptive packing for FPGAs. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 21–30. ACM, 2014.

[91] J. Luu, J. Rose, and J. Anderson. Towards interconnect-adaptive packing for fpgas. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 21–30. ACM, 2014.

[92] P. Lysaght. *Python on Zynq (Pynq) Documentation*. Xilinx, release 1.0 edition, July 5, 2016.

[93] P. Lysaght and D. Levi. Of gates and wires. *Parallel and Distributed Processing Symposium, International*, 4:132a, 2004.

[94] P. Maidee, C. Ababei, and K. Bazargan. Timing-driven partitioning-based placement for island style FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(3):395 – 406, march 2005.

[95] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(11):2331 –2340, Nov. 2006.

[96] A. Marquardt, V. Betz, and J. Rose. Speed and area tradeoffs in cluster-based FPGA architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):84–93, 2000.

[97] A. Marquardt, V. Betz, and J. Rose. Timing-Driven Placement for FPGAs. In *ACM/SIGDA Eighth International Symposium on Field-Programmable Gate Arrays (FPGA), Proceedings of the*, volume 313, pages 203–213, New York, NY, USA, 2000. ACM.

[98] A. S. Marquardt, V. Betz, and J. Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 37–46. ACM, 1999.

[99] Z. Marrakchi, H. Mrabet, and H. Mehrez. Hierarchical FPGA clustering based on a multilevel partitioning approach to improve routability and reduce power dissapation. In *null*, page 25. IEEE, 2005.

[100] L. McMurchie and C. Ebeling. Pathfinder: a negotiation-based performance-driven router for FPGAs. In *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, FPGA '95, pages 111–117, New York, NY, USA, 1995. ACM.

[101] A. Mishchenko. Enumeration of irredundant circuit structures. In *Proceedings of International Workshop on Logic and Synthesis*, 2014.

[102] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for LUT-based FPGAs. *IEEE TCAD*, 26(2):240–253, Feb 2007.

[103] J. Mistry. VSLI Basic: SDC (Synopsys Design Constraints). `http://vlsibasic.blogspot.be/2014/10/sdc-synopsys-design-constraints.html`, 2014.

[104] Y. Moctar, G. Lemieux, and P. Brisk. Routing algorithms for FPGAs with sparse intra-cluster routing crossbars. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 91–98, 2012.

[105] Morningstar. Xilinx is a Key Player in the PLD Segment of the Semiconductor Industry. [Online] http://analysisreport.morningstar.com/stock/research?t=XLNX&region=USA&culture=en-US&productcode=MLE, July 2014. Accessed: 08/05/2015.

[106] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap between Academic and Commercial CAD. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(2):10, 2015.

[107] Y. Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[108] B. New and S. Young. Method and apparatus for incorporating a multiplier into an FPGA, Mar. 26 2002. US Patent 6,362,650.

[109] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[110] F. G. P. Jamieson, K. B. Kent and L. Shannon. ODIN II: An Open-source Verilog HDL Synthesis Tool for CAD Research. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 149–156, 2010.

[111] H. Parandeh-Afshar, H. Benbihi, D. Novo, and P. Ienne. Rethinking fpgas: elude the flexibility excess of luts with and-inverter cones. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 119–128. ACM, 2012.

[112] T. Pi and P. Crotty. FPGA lookup table with transmission gate structure for reliable low-voltage operation, Dec. 2003. US Patent 6,667,635.

[113] H. N. Que Yanghua1, Nachiket Kapre. Boosting convergence of timing closure using feature selection in a learning-driven approach. In *2016 International Conference on Field Programmable Logic and Applications*, pages 69–77. IEEE, 2016.

[114] S. T. Rajavel and A. Akoglu. MO-Pack: Many-objective clustering for FPGA CAD. In *Proceedings of the 48th Design Automation Conference*, pages 818–823. ACM, 2011.

[115] S. Ray, A. Mishchenko, N. Een, R. Brayton, S. Jang, and C. Chen. Mapping into lut structures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1579–1584. EDA Consortium, 2012.

[116] S. U. Rehman, A. Blanchardon, A. Ben Dhia, M. Benabdenbi, R. Chotin-Avot, L. Naviner, L. Anghel, H. Mehrez, E. Amouri, and Z. Marrakchi. Impact of cluster size on routability, testability and robustness of a cluster in a mesh fpga. In *VLSI (ISVLSI), 2014 IEEE Computer Society Annual Symposium on*, pages 553–558. IEEE, 2014.

[117] L. Shannon, V. Cojocaru, C. N. Dao, and P. H. Leong. Technology Scaling in FPGAs: Trends in Applications and Architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 1–8. IEEE, 2015.

[118] H. Sidiropoulos, K. Siozios, P. Figuli, D. Soudris, M. Hübner, and J. Becker. Jitpr: A framework for supporting fast application's implementation onto fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 6(2):7, 2013.

[119] T. Simonite. Intel Puts the Brakes on Moore's Law, March 2016. last accessed on 2 November 2016.

[120] A. Singh, G. Parthasarathy, and M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):643–663, 2002.

[121] A. Singh, G. Parthasarathy, and M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):643–663, 2002.

[122] P. Spindler, U. Schlichtmann, and F. M. Johannes. Kraftwerk - a fast force-directed quadratic placement approach using an accurate net model. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 27(8):1398–1411, Aug. 2008.

[123] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays, FPGA '16, pages 16–25, New York, NY, USA, 2016. ACM.

[124] J. S. Swartz, V. Betz, and J. Rose. A Fast Routability-Driven Router for FPGAs. In *ACM/SIGDA Sixth International Symposium on Field-Programmable Gate Arrays (FPGA), Proceedings of the*, pages 140–149, 1998.

[125] C. Sze, T.-C. Wang, and L.-C. Wang. Multilevel circuit clustering for delay minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1073–1085, 2004.

[126] M. Tom, D. Leong, and G. Lemieux. Un/DoPack: re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 680–687. ACM, 2006.

[127] S. Trimberger. Three Ages of FPGA. `http://www.eecg.toronto.edu/~jayar/FPGAseminar/2014-2/the-three-ages-of-fpga.html`.

[128] S. Trimberger. Three Ages of FPGA: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103:318–331, 2015.

[129] E. Vansteenkiste. An evaluation framework for an academic and commercial comparison. github.com/EliasVansteenkiste/EvaluationFramework, December 2015.

[130] E. Vansteenkiste, B. Al Farisi, K. Bruneel, and D. Stroobandt. TPaR: Place and Route Tools for the Dynamic Reconfiguration of the FPGA's Interconnect Network. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(3):370–383, Mar. 2014.

[131] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. A Connection Router for the Dynamic Reconfiguration of FPGAs. In *LECTURE NOTES IN COMPUTER SCIENCE*, volume 7199, pages 357–365, Berlin, Germany, 2012. Springer Verlag Berlin.

[132] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. Maximizing the reuse of routing resources in a reconfiguration-aware connection router. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 322–329, 2012.

[133] E. Vansteenkiste, K. Bruneel, and D. Stroobandt. A Connection-based Router for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 326–329. IEEE, 2013.

[134] E. Vansteenkiste, H. Fraisse, and A. Kaviani. Analyzing the Divide between FPGA Academic and Commercial Results. In *International Conference on Field Programmable Technology (ICFPT), Proceedings of the*, 2015.

[135] E. Vansteenkiste, A. Kaviani, and H. Fraisse. Analyzing the divide between fpga academic and commercial results. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 96–103. IEEE, 2015.

[136] E. Vansteenkiste and S. Lenders. Liquid: Fast FPGA Placement Via Steepest Gradient Descent Movement. `github.com/EliasVansteenkiste/The-Java-FPGA-Placement-Framework`, 2016.

[137] E. Vansteenkiste and J. Rommens. COFFE with cream: Extension for an automatic transistor sizing tool. `https://github.ugent.be/UGent-HES/COFFE_WITH_CREAM`, 2016.

[138] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt. Runtime-Quality Tradeoff in Partitioning Based Multithreaded Packing. In *26st International Conference on Field-Programmable Logic and Applications (FPL), Proceedings of the*, page 8, Lausanne, Swiss, 2016.

[139] M. M. Waldrop. The chips are down for Moore's law, February 2016. last accessed on 2 November 2016.

[140] K. Weiss, R. Kistner, A. Kunzmann, and W. Rosenstiel. Analysis of the XC6000 architecture for embedded system design. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 245 –252, apr 1998.

[141] Xilinx. *Two Flows for Partial Reconfiguration : Module Based or Small Bit Manipulations*, 2002. UG290.

[142] Xilinx. *Comparing and Contrasting FPGA and Microprocessor*, 2004. WP213.

[143] Xilinx. *Partial Reconfiguration User Guide*, 2010. UG702.

[144] Xilinx. Backgrounder: The Xilinx SDAccel Development Environment. `http://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf`, 2014.

[145] Xilinx. *Vivado Design Suite User Guide: Implementation*, 2014. UG892.

[146] Xilinx. *Vivado Design Suite User Guide: Partial Reconfiguration*, 2014. UG909.

[147] Xilinx. *UltraScale Architecture Configurable Logic Block User Guide UG574*, November 2015.

[148] Xilinx. *UltraScale Architecture DSP Slice User Guide UG579*, November 2015.

[149] Xilinx. *UltraScale Architecture Memory Resources UG573*, November 2015.

[150] Xilinx. *Vivado Design Suite User Guide 910*. Xilinx Inc., Apr. 2015. Available online: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug910-vivado-getting-started.pdf`.

[151] M. Xu, G. Grewal, and S. Areibi. Starplace: A new analytic method for {FPGA} placement. *Integration, the {VLSI} Journal*, 44(3):192 – 204, 2011.

[152] Y. Xu and M. A. S. Khalid. Qpf: Efficient quadratic placement for fpgas. In T. Rissa, S. J. E. Wilton, and P. H. W. Leong, editors, *FPL*, pages 555–558. IEEE, 2005.

[153] S. Young, P. Alfke, C. Fewer, S. McMillan, B. Blodget, and D. Levi. A high I/O reconfigurable crossbar switch. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2003.

[154] G. Zgheib, L. Yang, Z. Huang, D. Novo, H. Parandeh-Afshar, H. Yang, and P. Ienne. Revisiting and-inverter cones. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 45–54. ACM, 2014.

[155] W. Zhao and Y. Cao. New generation of predictive technology model for sub-45nm early design exploration. *IEEE Transactions on Electron Devices*, 53(11):2816–2823, Nov. 2006.