Efficiënte circuitspecialisatie voor dynamische herconfiguratie van FPGA's

Efficient Circuit Specialization for Dynamic Reconfiguration of FPGAs

Karel Bruneel

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

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2010 - 2011

UNIVERSITEIT
GENT

ISBN 978-90-8578-443-2
NUR 959, 987
Wettelijk depot: D/2011/10.500/47

Dankwoord

Dit proefschrift was er nooit gekomen zonder de hulp en steun van heel wat mensen.

Vooreerst wil ik mijn collega's bedanken die steeds zorgden voor een aangename en uitdagende werksfeer. Brahim, Fatma, Harald, Peter, Tom Davidson, Tom Degryse en Wim Heirman wil ik bedanken voor het wetenschappelijke werk dat we samen realiseerden. Daarbij mag ik zeker de technische ondersteuning van Michiel Ronsse, Ronny en Wim Meeus niet vergeten. Mijn andere collega's wil ik danken voor hun goede raad, de aangename middagpauzes, de veelzijdige discussies, de WELEK-avonden, de hilarische robotcompetities en nog veel meer. Benjamin, David, Eric, Francis, Hendrik, Mark, Michiel D'Haene, Peter, Philippe, Pierre, Sean, Wim Heirman, ... Bedankt!

Dankzij een ongelofelijke groep vrienden kon ik mij tijdens mijn vrije tijd ontspannen op fantastische feestjes, etentjes, uitstappen, heroïsche fietstochten en memorabele vakanties. Dank aan Christophe (Boelty), Jean (Pepeke), Koen (Dikke), Koen (Poeky), Marlies (Speldje), Nina, Stefaan (Pleetje), Stijn (C'tje), Sven (Lange), ... Ik dank ook mijn ouders en mijn broers, Wouter en Bartel, voor al hun hulp en steun tijdens de voorbije jaren.

Financieel werd dit werk mogelijk gemaakt door een doctoraatsbeurs van het Bijzonder Onderzoeksfonds van de universiteit Gent. Ook mijn promotor, Dirk Stroobandt, wordt bedankt voor de kans die hij mij gaf om in alle vrijheid mijn onderzoek op te bouwen. Tot slot wens ik graag de leden van mijn doctoraatsjury te bedanken voor hun opbouwende kritiek. In het bijzonder prof. Jan Van Campenhout voor het op korte tijd nalezen en bekritiseren van een ruwe versie van dit proefschrift.

Karel Bruneel
Gent, 22 augustus 2011

Examination Commission

prof. dr. ir.	Daniël De Zutter, chairman Department of Information Technology - INTEC Faculty of Engineering Ghent University
prof. dr. ir.	Jan Van Campenhout, secretary Dept. of Electronics and Information Systems - ELIS Faculty of Engineering Ghent University
dr. ing.	Michael Hübner Institute for Information Processing - ITIV Karlsruhe Institute of Technology
prof. dr. ir.	Ingrid Verbauwhede Department of Electrical Engineering - ESAT Faculty of Engineering Katholieke Universiteit Leuven
dr. ir.	Peter Veelaert Department of Applied Engineering Hogeschool Gent
prof. dr. ir.	Dirk Stroobandt, advisor Dept. of Electronics and Information Systems - ELIS Faculty of Engineering Ghent University

Samenvatting

Field programmabale gate arrays (FPGA's) zijn heel aantrekkelijk als ontwerpplatform voor digitale systemen. FPGA's zijn geïntegreerde schakelingen die geprogrammeerd of geconfigureerd kunnen worden nadat ze geproduceerd zijn. Door een FPGA te configureren, kan deze het gedrag vertonen van om het even welke digitale schakeling. Zoals ook elke digitale schakeling kan gebouwd worden als een ASIC (Application Specific Integrated Circuit). Door hun *flexibiliteit* zijn FPGA-implementaties een aantrekkelijk alternatief voor ASIC-implementaties met een kleine oplage, omdat de NRE-kost en dus het economisch risico voor FPGA's veel lager ligt dan voor ASIC's. Voor FPGA's moeten namelijk geen dure maskers geproduceerd worden zoals het geval is voor ASIC's. Verder kan het ontwerp op een hoger niveau gebeuren door de beschikbaarheid van goede ontwerpsoftware en eenvoudige verificatie. Daarenboven wordt de time-to-market verkort omdat het ontwerp de flexibiliteit van de FPGA overerft en het ontwerp dus kan aangepast worden na verkoop. Natuurlijk is de flexibiliteit van een FPGA niet kosteloos. Een FPGA-implementatie zal altijd meer chipoppervlak verbruiken, een lagere prestatie hebben en meer vermogen verbruiken dan zijn ASIC-tegenhanger.

Dynamische herconfiguratie. Dit laatste is vooral het geval wanneer statische configuratie van de FPGA gebruikt wordt. Bij statische configuratie wordt de FPGA enkel geconfigureerd wanneer het systeem wordt aangezet. Daarna wordt de configuratie niet meer veranderd. Het is echter mogelijk om de configuratie van een FPGA te wijzigen tijdens de werking. Deze techniek heet *Dynamische Herconfiguratie*. Dynamische herconfiguratie maakt het mogelijk verschillende functionaliteiten die niet gelijktijdig worden gebruikt te multiplexen in de tijd. Op een ASIC, zou elk van deze functio-

naliteiten geïmplementeerd worden in een apart gebied van de chip dat een groot deel van de tijd zou stil liggen, omdat slechts een van de functionaliteiten nodig is op elk moment. Omdat de functionaliteiten nooit gelijktijdig gebruikt worden, kan op een FPGA, met behulp van dynamische herconfiguratie, hetzelfde gebied gebruikt worden voor al de functionaliteiten. Wanneer een andere functionaliteit nodig is, kan deze functionaliteit eenvoudig ingeladen worden in het gebied. Aangezien de ASIC-chipoppervlakte groeit met het aantal functionaliteiten, terwijl de FPGA-oppervlakte constant blijft, kan door gebruik te maken van dynamische herconfiguratie de oppervlakte-efficiëntie verbeterd worden. Er is zelfs een punt waar de oppervlakte-efficientie van de FPGA de oppervlakte-efficiëntie van de ASIC overstijgt. Dit zou het geval kunnen zijn voor toepassingen als software defined radio.

Hoewel FPGA's in principe al sinds hun ontstaan, in de jaren 80, kunnen geherconfigureerd worden, wordt dynamische herconfiguratie nog nauwelijks toegepast buiten het laboratorium. Dit komt door de hogere complexiteit van het ontwerp en de zeer beperkte software-ondersteuning.

Deze doctoraatsthesis heeft als doel om dynamische herconfiguratie uit het onderzoekslaboratorium te halen en het gebruik ervan mogelijk te maken in de echte wereld, waar krappe budgetten het ontwerpen op een laag ontwerpsniveau niet toelaten en software-ondersteuning dus absoluut noodzakelijk is. Ik concentreerde mij op een veelbelovende vorm van dynamische herconfiguratie, namelijk *Dynamische Circuit-specialisatie* (DCS), waarvoor tot nu toe geen software-ondersteuning beschikbaar was. Bij conventionele dynamische herconfiguratie, die ik *Configuration Swapping* noem, is het mogelijk om te wisselen tussen een beperkte verzameling van discrete functionaliteiten. Bij DCS daarentegen, wordt de configuratie van de FPGA gespecialiseerd voor specifieke parameterwaarden, wat niet mogelijk is met configuration swapping. Het toepassingsdomein wordt dus uitbreid. Voorheen moesten DCS-implementaties zonder software-ondersteuning vervaardigd worden op het niveau van de FPGA-architectuur. Dit impliceert een grote ontwikkelingstijd door zeer gespecialiseerde ontwikkelaars en is dus erg duur. In dit werk, beschrijf ik een methodologie die het ontwerp van DCS-systemen naar het RT-niveau tilt en de verdere verfijning automatiseert. De automatisch gegenereerde DCS-systemen

hebben een kwaliteit die vergelijkbaar is met die van systemen die handmatig werden gerealiseerd op het architecturaal ontwerpsniveau. Door het ontwerpsniveau naar het RT-niveau te tillen worden de ontwikkelkosten van het DCS-systeem sterk gereduceerd.

Dynamische Circuit-specialisatie is een techniek die het mogelijk maakt een FPGA-configuratie dynamisch te specialiseren voor de waarden van een aantal *parameters*. Het algemene idee van DCS is dat elke keer dat de parameters van waarde veranderen, de FPGA geconfigureerd wordt met een configuratie die gespecialiseerd is voor de nieuwe waarden van de parameters. Aangezien gespecialiseerde configuraties kleiner en sneller zijn dan hun generieke tegenhangers, is de hoop dat de implementatie van het volledige systeem efficiënter wordt door het gebruik van DCS. De belangrijkste uitdaging bij het bouwen van een DCS-systeem is het feit dat de gespecialiseerde configuraties snel moeten worden gegenereerd tijdens de uitvoering van het systeem en dat deze configuraties tegelijkertijd van goede kwaliteit (grootte en snelheid) moeten zijn. Anders zal de efficiëntie van een DCS-implementatie lager zijn dan de efficiëntie van de klassieke implementatie.

Geparametriseerde configuraties. De technieken voor het snel genereren van gespecialiseerde configuraties die in dit werk beschreven worden, zijn allen gebaseerd op het nieuwe concept van *Geparametriseerde Configuraties*. Dit zijn meerwaardige Boolese functies die de gespecialiseerde FPGA-configuratie uitdrukken als een functie van de parameterwaarden. Zodra een geparametriseerde configuratie gevonden is, houdt het genereren van een gespecialiseerde configuratie voor een bepaalde parameterwaarde niet meer in dan het evalueren van de geparametriseerde configuratie voor die parameterwaarde. Deze configuratie kan vervolgens gebruikt worden om de FPGA te herconfigureren. Het is belangrijk om hier op te merken dat, in tegenstelling tot het genereren van een FPGA-configuratie met conventionele methodes, het evalueren van een geparametriseerde configuratie niet NP-compleet is.

Het nieuwe concept van een geparametriseerde configuratie vormt de basis van dit proefschrift, maar een grote uitdaging is het genereren van een geparametriseerde configuratie startende vanuit een RT-niveau beschrijving van de generieke functionaliteit die moet worden geïmplementeerd. De beschrij-

ving wordt een geparametriseerde HDL-beschrijving genoemd. Een geparametriseerde HDL-beschrijving is een normale HDL-beschrijving waarin een onderscheid wordt gemaakt tussen reguliere ingangen en parameteringangen. In dit werk, beschrijf ik twee methodes die automatische generatie van geparametriseerde configuraties mogelijk maken vanuit een geparametriseerde HDL-beschrijving. De methodes maken gebruik van dezelfde stappen als de methodes die gebruikt worden voor het genereren van conventionele FPGA-configuraties: synthese, technology mapping, plaatsing en routering.

TLUT-methode. De eerste methode wordt de TLUT-methode genoemd. In de geparametriseerde configuraties geproduceerd door de TLUT-methode worden alleen de configuratiebits van de waarheidstabellen uitgedrukt als een Boolese functie van de parameterwaarden. Alle andere configuratiebits zijn statisch. Alleen de LUT's herconfigureren kan leiden tot zeer snelle herconfiguratie, maar aan de andere kant zal dit niet altijd leiden tot de meest compacte implementatie.

Voor de TLUT-methode moet enkel de technology mapping sterk aangepast worden. In dit proefschrift geef ik een gedetailleerde beschrijving van een aangepaste technologie mapper genaamd TMAP. TMAP is een nieuwe technology mapper die een circuit van logische poorten afbeeldt op een circuit van *Tunable LUT's* (TLUTs). In plaats van een statische waarheidstabel, hebben deze LUT's een waarheidstabel die wordt uitgedrukt als een Boolese functie van de parameters. Ik toon aan dat TMAP even goed schaalt als conventionele LUT-mappers en dat de complexiteit van de evaluatie van de waarheidstabellen lineair schaalt met de grootte van het logische circuit waarvan vertrokken wordt.

Naast de theoretische TLUT-methode, heb ik tevens een aantal praktische implementaties gemaakt die zich richten op commerciële FPGA's, in mijn geval de Xilinx Virtex-II Pro. Deze implementaties tonen aan dat DCS niet alleen in theorie mogelijk is, maar ook in de praktijk, met commercieel beschikbare FPGA's.

TCON-methode. Voor een tweede methode, heb ik de eerste stappen gezet naar het genereren van geparametriseerde configuraties waarin ook de routeringsbits van de FPGA worden uitgedrukt als een functie van de parameters. Deze methode heet de TCON-

methode. Voor bepaalde toepassingen, kan het herconfigureren van interconnecties leiden tot verdere vermindering van de vereiste FPGA-middelen, verhoging van de prestatie en het verlagen van het energieverbruik. Aan de andere kant heeft dit als nadeel dat de herconfiguratie meer tijd kan vergen omdat er meer bits moeten geherconfigureerd worden, en in tegenstelling tot de bits van de waarheidstabellen, zijn deze bits willekeurig verspreid in de configuratie-geheugenruimte.

De TCON-methode vereist drastische veranderingen aan zowel de technology mapper, de plaatser en de routeerder. De fundamentele verandering in de TCON-methode is de vervanging van het net, een interconnectie met een bron en meerdere doelen, met de *Tunable Connection* (TCON). Een TCON heeft een aantal bronnen en een aantal doelen en zijn parameteringangen bepalen hoe deze doelen en bronnen met elkaar verbonden zijn. De technology mapper moet het logisch circuit nu afbeelden op een mengsel van TLUT's en TCON's en de routeerder moet nu TCON's routeren in plaats van netten.

Dit werk bevat een schets van een TCON technology mapper en een volledig geïmplementeerde TCON-routeerder, de pattern router. De pattern router werd gebruikt om grote herconfigureerbare schakelnetwerken te implementeren met veelbelovende resultaten op het gebied van de gebruikte FPGA-middelen en de prestatie. Behalve de pattern router bevat dit werk ook een beschrijving van een andere, beter schaalbare TCON-routeerder, de Connection Router genaamd.

Samengevat toont dit werk aan dat het ontwerp van DCS-systemen naar een hoger abstractieniveau, met name het RT-niveau, getild kan worden, zonder afbreuk te doen aan de kwaliteit van de uiteindelijke implementatie en dat dit kan worden gedaan voor commercieel verkrijgbare FPGA's. Op deze manier wordt de toepasbaarheid van FPGA's en meer specifiek die van dynamische herconfiguratie uitgebreid.

Summary

Field programmabale gate arrays (FPGAs) are very attractive as a design platform for digital systems. FPGAs are off-the-shelf integrated circuits that can be programmed or configured many times after being manufactured. By configuring an FPGA, the user can make it behave like any digital system. As such, FPGAs intrinsically offer similar benefits as ASICs (Application Specific Integrated Circuits). The *flexibility* of FPGAs makes them an attractive alternative for low volume ASIC implementations, because it greatly reduces the NRE cost and the economic risk. Indeed, there is no need for the application developer to manufacture a expensive mask set when using FPGAs, as would be the case when using an ASIC. Also, the design of the system is a lot cheaper because of good tool support and easy verification. Besides this, the time-to-market is reduced because the flexibility of the FPGA is inherited by the design it implements which enables updating after deployment. Of course, the flexibility does not come for free. An FPGA implementation will always require more silicon area, have a lower performance and consume more power than its ASIC counterpart.

Dynamic reconfiguration. This last statement is mostly true when a static configuration of the FPGA is used. In a static configuration the FPGA is configured when the system is powered up and its configuration is never changed afterwards. However, it is possible to change the configuration of an FPGA while it is running. This technique is called *dynamic reconfiguration*. Dynamic reconfiguration enables time multiplexing of several functionalities that are not used simultaneously. In an ASIC, each of these functionalities would need to be implemented on a separate piece of silicon area that would sit idle most of the time because only one of the functionalities is needed at every moment. On an FPGA, using dynamic reconfiguration, the

same piece of FPGA area could be used for all these functionalities, because they are never needed at the same time. When a new functionality is needed, the FPGA area is simply reconfigured. Since the ASIC area grows with the number of functionalities while the FPGA area is constant, dynamic reconfiguration can boost the area efficiency of FPGAs in certain applications. There is even a point where an FPGA could outperform an ASIC in terms of area efficiency. This is for example believed to be the case for software defined radio.

Although FPGA devices could in principle be reconfigured since the early days of FPGAs in the 80's, dynamic reconfiguration is still hardly ever used outside the research lab, because of its higher complexity and very limited tool support.

In this Ph.D. thesis, my goal is to pull dynamic reconfiguration out of the research lab and into the real world, where tight budgets don't allow low level design and tool support is an absolute must. I focused on a promising form of dynamic reconfiguration called *Dynamic Circuit Specialization* (DCS) for which no tool support was available so far. Where conventional dynamic reconfiguration, which I call *configuration swapping*, allows the designer to switch between a small set of discrete functionalities, DCS enables the designer to specialize the configuration of the FPGA for a whole range of specific data values, thus enlarging the application domain of FPGA reconfiguration. Previously, DCS implementations needed to be hand-crafted at the architectural level. This implies long development times for highly specialized developers and thus is very expensive. In this work, I provide a methodology that enables automatic implementation of a DCS systems starting from an RT-level design while achieving similar quality as the hand-crafted implementation. By lifting the design of DCS systems to the RT level, my work greatly reduces the development cost of these systems.

Dynamic Circuit Specialization is a technique to dynamically specialize an FPGA configuration according to the values of a set of *parameters*. The general idea of DCS is that each time the parameter values change, the device is reconfigured with a configuration that is specialized for the new parameter values. Since specialized configurations are smaller and faster than their generic counterparts, the hope is that the system implementation will be more cost efficient when using DCS. The main difficulty when building a DCS system

is the fact that the specialized configurations need to be rapidly generated on the fly and at the same time, they need to be of good quality (size and speed). Otherwise, the efficiency of a DCS implementation will be lower than the efficiency of a classic implementation.

Parameterized Configurations. The fast configuration generation techniques described in this work all make use of the new concept of *parameterized configurations*. These are multivalued Boolean functions that express the specialized FPGA configuration as a function of the parameter values. Once a parameterized configuration is found, generating a specialized configuration for a given parameter value only requires the evaluation of the parameterized configuration for that parameter value. This configuration can then be used to reconfigure the FPGA. It is important to note here that in contrast to generating an FPGA configuration with a conventional tool flow, evaluating the parameterized configuration is not computationally hard.

The new concept of a parameterized configuration forms the basis of this thesis, but the challenge is to generate a parameterized configuration starting from a RT-level description of the functionality that needs to be implemented. This description is called a parameterized HDL description. A parameterized HDL description is a regular HDL description in which a distinction is made between regular inputs and parameter inputs. In this work, I present two methods that enable automatic generation of parameterized configurations starting from parameterized HDL descriptions. The methods use the same steps as conventional FPGA tool flows: synthesis, technology mapping, placement and routing.

TLUT method. The first method described in this work is called the TLUT method. In the parameterized configuration produced by the TLUT method, only the truth tables' configuration bits are expressed as a Boolean function of the parameter inputs. All other configuration bits are static. Only reconfiguring the LUTs can lead to very fast reconfiguration, but on the other hand might not lead to the most compact implementation.

In the TLUT method only the technology mapping step undergoes a major change. In this thesis, I give a detailed description of an adapted technology mapper called TMAP. TMAP is a new technology mapper that maps a gate-level circuit to *Tunable LUTs* (TLUTs). Instead of a static truth table, these LUTs have a truth table that is ex-

pressed as a Boolean function of the parameters. I show that TMAP scales equally well as conventional LUT mappers and that the complexity of evaluating the truth tables scales favorably with the size of the input gate-level circuit.

Besides the theoretical TLUT method, I have also built several practical tool flows that target commercial FPGAs, in my case the Xilinx Virtex-II Pro. These tool flows show that dynamic circuit specialization is not only possible in theory, but also in practice on commercially available FPGAs.

TCON method. For the second method, I have made the first steps towards parameterized configuration where also the routing configuration bits of the FPGA are expressed as a function of the parameters. This method is called the TCON method. For certain applications, also allowing reconfiguration of interconnect can lead to further reduction of the required FPGA area, can increase the performance and reduce power consumption. The disadvantage on the other hand is that the reconfiguration time may increase because more bits need to be reconfigured and unlike the truth table bits, these bits are typically scattered over the configuration memory space.

The TCON method requires drastic changes to technology mapping, placement and routing. The fundamental difference in the TCON method is the replacement of the net, an interconnection with one source and multiple sinks, with the *Tunable Connection* (TCON). A TCON has a number of sources and a number of sinks and its parameter inputs control how these sinks and sources are interconnected. The technology mapper now needs to map to a mixture of TLUTs and TCONs and the router will now need to route TCONs instead of nets.

This work contains an algorithm sketch of a TCON technology mapper and a fully implemented TCON router, called the pattern router. The pattern router is used to implement large reconfigurable switches with very promising results in terms of area usage and performance. Apart from the pattern router there is also a description of an other better scalable TCON router, called the connection router.

To conclude, this work shows how the design of DCS systems can be lifted to a higher abstraction level, namely the RT level, without compromising the quality of the final implementation and that this

can be done for commercially available FPGAs. In this way, the applicability of FPGAs and more specifically that of dynamic reconfiguration is extended.

Contents

List of Acronyms

AIG	And-Inverter Graph
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CAM	Content-Addressable Memory
CB	Connection Block
CI	Combinational Input
CLB	Configurable Logic Block
CO	Combinational Output
CPCIR	Constant Propagation, Compaction and Incremental Routing
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAG	Directed Acyclic Graph
DCS	Dynamic Circuit Specialization
DPGA	Dynamically Programmable Gate Array
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processor
FF	Flip-Flop
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port
IOB	Input/Output Block
IP	Intellectual Property
ISP	Instruction Set Processor
JVM	Java Virtual Machine

KCM	Constant (K) Coefficient Multiplier
LB	Logic Block
LUT	Lookup table
NOP	No Operation
NRE	Non-Recurring Engineering
OPB	On-Chip Peripheral Bus
PBS	Parameterized Bitstream
PI	Parameter Input
PLB	Processor Local Bus
PN	Parameter Node
PPC	Partial Parameterized Configuration
QUIP	Quartus II University Interface Program
RAM	Random-Access Memory
RI	Regular Input
RN	Regular Node
RTL	Register-Transfer Level
SAT	Boolean Satisfiability
SB	Switch Block
SRAM	Static Random-Access Memory
SRL	Shift Register LUT
TC	Template Configuration
TCAM	Ternary Content-Addressable Memory
TCON	Tunable Connection
TLUT	Tunable LUT
TOS	Top Of Stack
VHDL	Very-high-speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
VM	Virtual Machine
VPR	Versatile Placement and Routing
WELEK	Werkgroep Elektronica
XDL	Xilinx Design Language
XUP	Xilinx University Program

Chapter 1

Introduction

In the development of computing systems, one of the first things that need to be decided is which computing platform will be used. Will we use an off-the-shelf general purpose processor (GPP) or do we need to build a custom computing system to meet the specifications of our product? GPPs are a good choice, if they are able to reach the performance and power requirements. GPPs are cheap because they are mass produced and the development of the software that will run on the processor is relatively easy and thus cheap, certainly if compared to the development of an Application Specific Integrated Circuit (ASIC). If the needed performance or power specifications require an ASIC, things get a lot more expensive and thus risky. Because of the huge rise in development cost, only very high volume products justify the development of an ASIC. Luckily, GPPs and ASICs aren't the only available choices. There is a whole range of intermediate platforms (DSPs, GPUs, VLIWs,...), which each offer a specific trade-off between performance, power consumption and development cost. One of these intermediate platforms is the Field Programmable Gate Array (FPGA).

An FPGA is an off-the-shelf integrated circuit that can be programmed or configured after being manufactured. By configuring an FPGA, the user can make it behave like any digital system. This flexibility allows the same FPGA to be used by a wide variety of developers for a wide variety of products. FPGA manufacturers can thus produce large volumes of an FPGA even if the individual products it is used in have low volumes. For low volume products, the FPGA device cost is thus lower than the device cost of an ASIC. Besides the lower device cost of FPGAs compared to ASICs, there is also

a huge reduction in development cost. The design tools provided by the FPGA manufacturer hide almost all low level design aspects. The designer basically writes the RT-level HDL code and the design tools take care of the rest. Verification also becomes a lot easier since the design can simply be loaded in the FPGA and verified in-circuit without any additional cost. Of course, the reconfigurability of an FPGA also comes at a cost. On the one hand, FPGA implementations can easily outperform GPP software implementations in terms of speed and power consumption for applications that have lots of parallelism available, but on the other hand, an FPGA implementation will always require more area, have lower performance and consume more power than its ASIC counterpart [43].

This last statement is mostly true when a static configuration of the FPGA is used. When static configuration is used, the FPGA is configured when the system is powered up and its configuration is never changed afterwards. However, it is possible to change the configuration of an FPGA while it is running. This technique is called *dynamic reconfiguration*. Dynamic reconfiguration enables time multiplexing of several functionalities that are not used simultaneously. In an ASIC each of these functionalities would need to be implemented on a separate piece of silicon area that would sit idle most of the time because only one of the functionalities is needed at every moment. On an FPGA, using dynamic reconfiguration, the same piece of FPGA area could be used for all these functionalities, because they are never needed at the same time. When a new functionality is needed, the FPGA area is simply reconfigured. Since the ASIC area grows with the number of functionalities while the FPGA area is constant, dynamic reconfiguration can boost the area efficiency of FPGAs in certain applications, e.g. software defined radio [27]. There is even a point where an FPGA could outperform an ASIC in terms of area efficiency.

Although FPGA devices could in principle be reconfigured since the early days of FPGA in the 80's, dynamic reconfiguration is still hardly ever used outside the research lab, because of its higher complexity and very limited tool support. However, the past few years things have improved for dynamic reconfiguration. Xilinx was the first manufacturer that started offering tool support for dynamic reconfiguration and now Altera is also offering dynamic reconfiguration for its new 28 nm devices. Moreover, in early 2010, a new player in the FPGA market called Tabula released an FPGA tech-

nology (spacetime architecture) and a corresponding tool flow that implicitly uses dynamic reconfiguration.

The goal of my work is to pull dynamic reconfiguration out of the research lab and into the real world, where tight budgets don't allow low level design and tool support is an absolute must. In this thesis, I focused on a promising form of dynamic reconfiguration called *Dynamic Circuit Specialization* for which no tool support was available so far. While conventional dynamic reconfiguration, which I call *configuration swapping*, allows the designer to switch between a small set of discrete functionalities, DCS enables the designer to specialize the configuration of the FPGA for a whole range of specific data values, thus enlarging the application domain of FPGA reconfiguration. Previously, DCS implementations needed to be hand-crafted at the architectural level. This implies long development times by highly specialized developers and thus is very expensive. In this thesis, I provide a methodology for automating the implementation of a DCS system starting from an RT-level design while achieving similar quality as the hand-crafted implementation. By lifting the design of DCS systems to the RT level, my work greatly reduces the development cost of these systems.

1.1 Dynamic Reconfiguration

In view of reconfiguration, the FPGA fabric can be used in several ways. The most commonly used form of dynamic reconfiguration is configuration swapping as is supported by the two major FPGA manufacturers Xilinx [86] and Altera [3]. Configuration swapping is however not the only way in which dynamic reconfiguration can be used. Dynamic circuit specialization for example is an orthogonal way of using dynamic reconfiguration. Figure 1.1 shows the different ways in which configuration swapping and dynamic circuit specialization can be combined. Each of the subfigures shows a behavioral model of a system that makes use of a GPP and an FPGA. The behavior is represented as a network of communicating processes [35]. These processes are mapped to a set of resources: processing elements (in this case the GPP and the FPGA resources) and communication channels (buses, FIFOs, ...).

In the conventional FPGA use model, FPGA fabric is used in the same way as silicon area in an ASIC. A process is assigned to a dedicated set of FPGA resources. These resources are configured

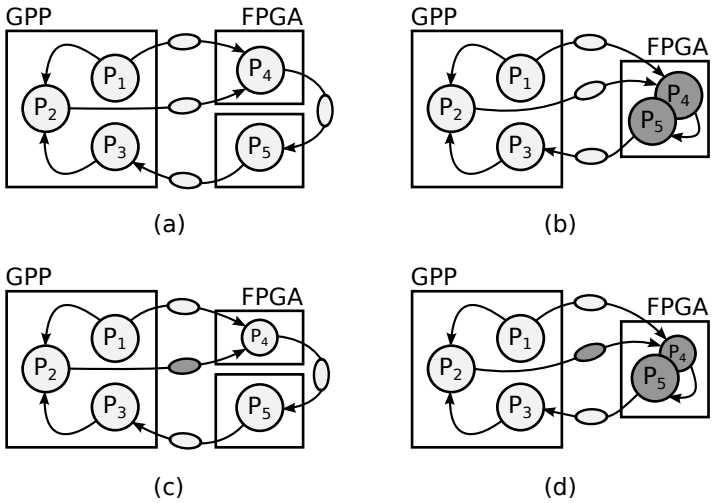


Figure 1.1: Options for using reconfiguration: (a) No reconfiguration used, (b) P_4 and P_5 timeshare FPGA resources, (c) P_2 communicates with P_4 by means of reconfiguration (d) combination of timesharing and communication by means of reconfiguration.

when the system starts and keep the same configuration during the run time. This first mapping option is illustrated in Figure 1.1 (a). Processes P_4 and P_5 are both assigned to a dedicated set of FPGA resources and the communication channels are implemented using conventional techniques like buses, FIFOs, etc.

1.1.1 Configuration Swapping

A first option to introduce dynamic reconfiguration is to have multiple processes timeshare a set of FPGA resources. This option is illustrated in Figuur 1.1 (b). Process P_4 and P_5 are both assigned to the same set of FPGA resources. If P_4 and P_5 don't require the performance given by dedicated resources, this technique leads to a more cost efficient implementation because only one set of FPGA resources is needed. Communication with P_4 and P_5 is implemented with conventional methods like buses and buffers, but of course the necessary precautions should be taken so that the communication channels can handle the configuration swaps.

Tool support for configuration swapping can simply be realized by slightly adapting a conventional tool flow. Xilinx has adapted

its modular design flow to enable configuration swapping [86] and Altera will probably create a similar tool flow [3].

The main difference for designers is the fact that they will need to prepare one or more parts of the FPGA for dynamic reconfiguration and they will need to describe all the functionalities that might be implemented in these areas 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.

1.1.2 Dynamic Circuit Specialization

Dynamic Circuit Specialization [34, 74] is a technique to dynamically specialize an FPGA configuration according to the values of a set of *parameters*. DCS is closely related to partial evaluation [39], which is a technique used in software to generate specialized programs from a generic program, by fixing one of the arguments of the generic program. In mathematics fixing one of the arguments of a function is called restriction and in logic this is called currying.

The general idea of DCS is that each time the parameter values change, the device is reconfigured with a configuration that is specialized for these new values. Since specialized configurations are smaller and faster than their generic counterparts, the hope is that the system implementation will be more cost efficient when using DCS, but of course, there is a catch. Indeed, every time the parameters change value, a new configuration needs to be generated and the FPGA needs to be reconfigured. This is called the specialization overhead. When the parameters change too frequently this overhead can negate the advantage of using the smaller and faster specialized configuration and thus render DCS useless. Therefore, the objective of this thesis is not only to automate the design of DCS systems, but to do this while keeping the specialization overhead of the automatically generated DCS implementations at the same level as the specialization overhead of hand-crafted implementations.

A good example of where DCS can be used is adaptive filtering. In adaptive filtering the filter coefficients are the parameters. Every time the filter characteristic needs to change, the values of the coefficients (parameters) have to be changed. In order to do this, a specialized filter configuration is generated and this configuration is loaded

in the FPGA's configuration memory. The main advantage here is that the specialized FIR filters are typically 40% smaller than their generic counterparts and can be clocked about 30% faster (see Section 5.6.1). This is because constant multipliers can be used instead of bulky generic multipliers. On the other hand, generating specialized filters introduces a specialization overhead that is only acceptable when the coefficients don't change to often. Using the techniques described in this thesis, specialization frequencies in the order of tens of reconfigurations per second can be achieved for adaptive FIR filters (see Section 5.6.1).

At the system level, DCS can be abstracted to a communication channel between two processes of a system, called the sender and the receiver, where the receiver is implemented on an FPGA. The parameters are the data that are sent over the DCS channel. This is illustrated in Figure 1.1 (c). Process P_4 and P_5 are assigned to dedicated sets of resources, but now communication between P_2 and P_4 is done by means of reconfiguration.

A DCS channel is an abstraction of functionality that will reconfigure a set of FPGA resources every time a new parameter value is sent over the DCS channel, so that the input/output relation of the set of FPGA resources is specialized for the new parameter. To implement these channels it will, on the one hand, be necessary to select a set of FPGA resources that can implement the specialized functionality for each possible parameter value. On the other hand, it will be necessary to implement a subsystem that, given a parameter value, determines a configuration for these resources. These two communicating subsystems are shown in Figure 1.2. This can be seen as a partitioning of the functionality of process P into two subprocesses $P_{res.}$ and $P_{spec.}$. The subprocess $P_{spec.}$ is responsible for generating a configuration for the selected FPGA resources given a parameter value, while the subprocess $P_{res.}$ represents the functionality of these specialized FPGA resources.

1.1.3 Combining Configuration Swapping and DCS

Both reconfiguration abstractions are orthogonal and can thus be used simultaneously as is shown in Figure 1.1 (d). P_4 and P_5 timeshare the same set of resources and communication between P_2 and P_4 is done by means of DCS. Each time P_2 sends new data, d, to P_4 a configuration (P_4^d), which is a configuration specialized for d, is gen-

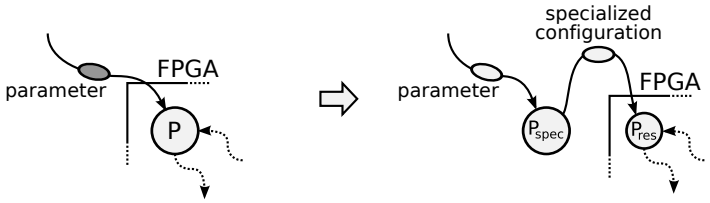


Figure 1.2: Refinement of communication by means of dynamic circuit specialization.

erated and this configuration is timeshared with the configuration of P_5.

1.2 Contributions

The main contribution of this work is a methodology that enable the automated design of DCS systems at the RT level. More concretely, starting from an RT-level description of the generic functionality (the functionality of P in Figure 1.2), the methods described in this thesis automatically implements the specialization process ($P_{spec.}$) and at the same time select the set of FPGA resource that will be reconfigured ($P_{res.}$). Previously, efficient DCS system needed to be handcrafted at the architectural level of the FPGA, requiring information (the organization of the FPGA's configuration memory) that is not even published by FPGA manufacturers.

1.2.1 Parameterized Configurations

The methods described in this work are all based on the observation that the behavior of the specialization process can be abstracted to a multivalued Boolean function, which I call a *Parameterized Configuration*. This is true since both the input (a parameter value) and the output (a specialized configuration for the FPGA) of the specialization process are bit vectors.

Once the parameterized configuration for a certain process is found, the specialization process only needs to evaluate the parameterized configuration every time the parameter values change. The result of evaluating the parameterized configuration is a configuration that is specialized for the parameter value under consideration. This configuration can then be used to reconfigure the FPGA.

It is important to note here that, in contrast to generating an FPGA configuration with a conventional tool flow, evaluating the parameterized configuration is not computationally hard. In this way, the specialization overhead is kept under control.

1.2.2 Generating Parameterized Configurations

The new concept of a parameterized configuration forms the basis of this thesis, but equally as important are two methods that enable automatic generation of a parameterized configuration starting from a RT-level description of the generic functionality.

The generic functionality is presented in the form of a parameterized HDL description. This is a regular HDL description in which a distinction is made between regular inputs and parameter inputs. The only other requirement for the HDL description is that it is synthesizable by a conventional synthesis tool when the parameter inputs are regarded as regular inputs.

The software counterpart of generating parameterized configurations is generating program generators [39]. The program that generates a program generator from a generic program and one of its arguments is the software counterpart for the methods described in this thesis. The resulting program generator is a program that produces specialized versions of the generic program, when given a fixed value for the argument. This is similar to a parameterized configurations which produce specialized versions of the generic functionality described as a parameterized HDL description, when given new parameter values.

Dynamic Reconfiguration of LUTs

In the parameterized configuration produced by the first method, only the truth tables' configuration bits are expressed as a Boolean function of the parameter inputs. All other configuration bits, mainly routing bits, are static. This first method is called the TLUT method. As I will explain, only reconfiguring the LUTs can lead to very fast reconfiguration, but on the other hand might not lead to the most compact implementation.

Similar to conventional FPGA tool flows, the TLUT method is also divided into four steps: synthesis, technology mapping, placement and routing. This can be seen in Figure 1.3. The algorithms

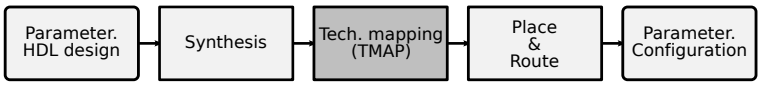


Figure 1.3: Overview of the TLUT method.

used to solve the subproblems in the TLUT method are adapted versions of the conventional algorithms. In the TLUT method only the technology mapping step undergoes a major change.

In this thesis, I present a detailed description of a technology mapper called TMAP. TMAP is a new technology mapper, which maps a gate-level circuit to Tunable LUTs (TLUTs). Instead of a static truth table, these LUTs have a truth table that is expressed as a Boolean function of the parameters. I show that TMAP scales equally well as conventional LUT mappers and that the complexity of evaluating the truth tables scales favorably with the size of the input gate-level circuit.

Besides the theoretical TLUT method, I have also built practical tool flows that target commercial FPGAs, in my case the Xilinx Virtex-II Pro. These tool flows show that dynamic circuit specialization is not only possible in theory, but also in practice on commercially available FPGAs.

Dynamic Reconfiguration of Routing

I have also made the first steps towards a method to generate parameterized configuration where the routing configuration bits of the FPGA are also expressed as a function of the parameters. The implementation of this method is called the TCON method. For certain applications, e.g. crossbars, also allowing reconfiguration of interconnect can lead to further reduction of the required FPGA area, increases the performance and reduces power consumption. On the other hand, the disadvantage is that the specialization overhead may increase because more bits need to be reconfigured and unlike the truth table bits, these bits are typically scattered over the configuration memory space.

As can be seen in Figure 1.4, the mapping process is again divided in the conventional way. The TCON method however requires drastic changes to technology mapping, placement and routing. The main difference in the TCON method is the replacement of the net, an interconnection with one source and multiple sinks, with the TCON.

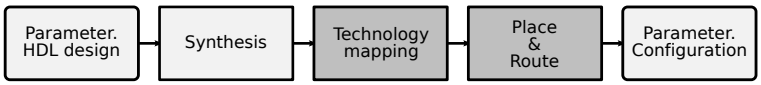


Figure 1.4: Overview of the TCON method.

A TCON has a number of sources and a number of sinks and its parameter inputs control how these sinks and sources are interconnected. The technology mapper now needs to map a gate-level circuit to a mixture of TLUTs and TCONs and the router will now need to route TCONs instead of nets.

This work contains an algorithm sketch of a TCON technology mapper and a fully implemented TCON router, called the pattern router. Apart from the pattern router there is also a description of an other better scalable TCON router, called the connection router. The pattern router is used to implement large reconfigurable switches with very promising results in terms of area usage and performance.

To summarize, this work introduces the concept of parameterized configurations and shows that parameterized configurations can be generated automatically from an RT-level design. In order to do this, I developed two methods. One that only allows reconfiguration of LUTs, called the TLUT method and an extended version that also allows reconfiguration of the routing infrastructure, called the TCON method. Furthermore, I show how to use parameterized configurations as a means to implement DCS on commercially available FPGAs.

1.3 Structure of the Thesis

This thesis is organized as follows:

In Chapter 2, I explain in more detail how contemporary FPGAs are built. I especially concentrate on the aspects of FPGA devices that are important for dynamic reconfiguration and more specifically, dynamic circuit specialization. I also describe a commonly used tool flow for generating static FPGA configurations and configuration swapping applications. I describe omnipresent algorithms for technology mapping, placement and routing. These algorithms will be adapted to incorporate parameterizability in the later chapters.

Chapter 3 gives a detailed analysis of dynamic circuit specialization and gives an overview of the state of the art in the field of dy-

namic circuit specialization. I also introduce parameterized configurations as a way to implement dynamic circuit specialization.

An overview of the TLUT method is given in Chapter 4, followed by a detailed description of the adapted technology mapper called TMAP in Chapter 5. Next, the TLUT method is used to build real life DCS systems on a commercial FPGA (Xilinx Virtex-II Pro) in Chapter 6.

Chapter 7 explains the first steps towards an extension of the TLUT method so that not only the truth table bits are reconfigurable but also the routing bits. The chapter explains the concept of a TCON and describes algorithms that are able to process TCONs.

Finally I draw the conclusions of this thesis and discuss future work in Chapter 8.

1.4 Publications

Journal papers

- **Karel Bruneel**, Wim Heirman and Dirk Stroobandt. Dynamic Data Folding with Parameterizable Configurations. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 16, No. 4, 2011.

Conference papers with international peer review (in the ISI data base)

- **Karel Bruneel** and Dirk Stroobandt. TROUTE: a Reconfigurability-aware FPGA Router. In *Lecture Notes in Computer Science*, pp. 207–218, 2010.
- **Karel Bruneel**, Fatma Abouelella and Dirk Stroobandt. Automatically mapping applications to a self-reconfiguring platform. In *Design, Automation and Test in Europe Conference and Expo*, pp. 964–969, 2009.
- **Karel Bruneel** and Dirk Stroobandt. Automatic Generation of Run-time Parameterizable Configurations. In *International Conference on Field Programmable and Logic Applications*, pp. 360–365, 2008.
- **Karel Bruneel**, Peter Bertels and Dirk Stroobandt. A Method for Fast Hardware Specialization at Run-time. In *International*

Conference on Field Programmable and Logic Applications, pp. 35–40, 2007.

Other conference papers with international peer review

- Dirk Stroobandt and **Karel Bruneel**. How Parameterizable Run-time FPGA Reconfiguration can Benefit Adaptive Embedded Systems. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems & Algorithms*, pp. 184–194, 2011.
- Tom Davidson, **Karel Bruneel** and Dirk Stroobandt. Run-time Reconfiguration for Automatic Hardware/Software Partitioning. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs*, pp. 424–429, 2010.
- Fatma Abouelella, **Karel Bruneel** and Dirk Stroobandt. Efficiently Generating FPGA Configurations through a Stack Machine. In *International Conference on Field Programmable and Logic Applications*, pp. 35–39, 2010.
- Fatma Abouelella, **Karel Bruneel** and Dirk Stroobandt. Towards a More Efficient Run-time FPGA Configuration Generation. In *Parallel Computing: From Multicores and GPU's to Petascale*, pp. 624–631, 2010.
- Tom Davidson, **Karel Bruneel**, Harald Devos and Dirk Stroobandt. Applying Parameterizable Dynamic Configurations to Sequence Alignment. In *Parallel Computing: From Multicores and GPU's to Petascale*, pp. 616–623, 2010.
- Fatma Abouelella, **Karel Bruneel** and Dirk Stroobandt. Automatically Mapping Applications to a Self-reconfiguring Platform. In *Proceedings of the 19th Annual ProRISC Workshop*, 2008.
- Tom Degryse, **Karel Bruneel**, Harald Devos and Dirk Stroobandt. Reducing the Dynamic FPGA Reconfiguration Overhead. In *Architecture and Compilers for Embedded Systems*, pp. 53–56, 2008.
- **Karel Bruneel** and Dirk Stroobandt. Reconfigurability-Aware Structural Mapping for LUT-based FPGAs. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs*, pp. 223–228, 2008.

- Tom Degryse, **Karel Bruneel**, Harald Devos and Dirk Stroobandt. Loop Transformations to Reduce the Dynamic FPGA Reconfiguration Overhead. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs*, pp. 133–138, 2008.

Poster presentations / abstracts

- Brahim Al Farisi, Karel Heyse, **Karel Bruneel** and Dirk Stroobandt. Memory-efficient and fast run-time reconfiguration of regularly structured designs. In *International Conference on Field Programmable and Logic Applications*, 2011.

- Robbe Vancayseele, Brahim Al Farisi, Wim Heirman, **Karel Bruneel** and Dirk Stroobandt. RecoNoC : a reconfigurable network-on-chip. In *Proceedings of the International workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2011.

- Brahim Al Farisi, **Karel Bruneel** and Dirk Stroobandt. Automatic Tool Flow for Shift-Register-LUT Reconfiguration : Making Run-time Reconfiguration Fast and Easy. In *Proceedings of the ACM/SIGDA 18th International Symposium on Field Programmable Gate Arrays*, pp. 287–287, 2010.

- Tom Degryse, **Karel Bruneel**, Harald Devos and Dirk Stroobandt. Reducing the dynamic FPGA reconfiguration overhead with loop transformations. In *Fourth International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, pp. 219–222, 2008.

- **Karel Bruneel**. FPGA Architectures for Fast Circuit Compilation. In *Seventh FirW PhD Symposium*, 2006.

Other publications

- Dirk Stroobandt and **Karel Bruneel**. Supersnelle Runtime Herconfiguratie Eindelijk Binnen Handbereik. In *Bits & Chips*, pp. 30–31, 2008.

Chapter 2

FPGA Architecture and Conventional Tool Flow

An FPGA (Field Programmable Gate Array) is an integrated circuit that can be programmed or configured after being manufactured. By configuring an FPGA, the user can make it behave like conventional digital logic. In this chapter, I describe both the architecture of a simple FPGA and the tool flow that is used to map an HDL design to an FPGA.

2.1 The Basic FPGA Architecture

In view of dynamic reconfiguration, an FPGA contains two important parts: the FPGA fabric, and the configuration memory (Figure 2.1). The configuration memory is responsible for storing the data that controls the FPGA fabric. The user can access this configuration memory through the configuration interface of the FPGA. The FPGA fabric is responsible for emulating the functionality that is specified by the configuration stored in the configuration memory.

2.1.1 The FPGA Fabric

All FPGA fabrics contain I/O blocks (IOB), logic blocks (LB) and programable routing. The logic blocks implement small parts of the desired functionality. The I/O blocks act as inputs or outputs and serve as an interface to the outside world. The programmable routing is used to connect the logic blocks to other logic blocks or to the I/O

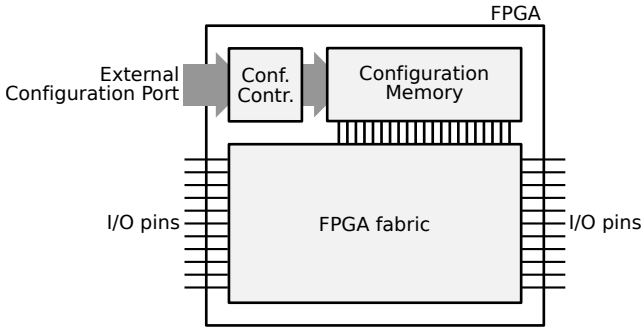


Figure 2.1: High level FPGA architecture representing the configuration memory and the FPGA fabric.

blocks so that the fabric performs the desired functionality. Figure 2.2 illustrates the fabric of a typical FPGA. The logic blocks are arranged in an array which is embedded in the reconfigurable routing.

Logic Blocks

While many types of logic blocks have been used in the past, most of the current commercial FPGAs use logic blocks that are based on Look-up tables or LUTs [94, 92, 81, 83] and flip flops (FF). Figure 2.3 shows the architecture of a simple logic block.

The left hand side of Figure 2.3 shows a 3-input LUT. The LUT is built as a multiplexer of which the data inputs are driven from configuration memory bits and the select inputs are controlled from within the FPGA fabric. The output of the LUT is again available in the FPGA fabric. From the point of view of the FPGA fabric, the LUT appears to be a 3-input gate. The gate can be programmed to implement any 3-input Boolean function by simply writing the truth table of the desired Boolean function in the configuration bits of the LUT.

At the start of my PhD research, the logic blocks of most commercial FPGAs were based on 4-input LUTs, because research [62] had shown that 4-input LUTs lead to the highest area efficiency. More recent research [2] however has shown that this is no longer true for large deep-submicron FPGAs. Therefore, FPGA manufacturers have switched to LUTs with higher input counts [94, 92, 83]. The research in this thesis is mostly based on 4-input LUTs, but the concepts and the algorithms can easily be adapted to larger LUTs.

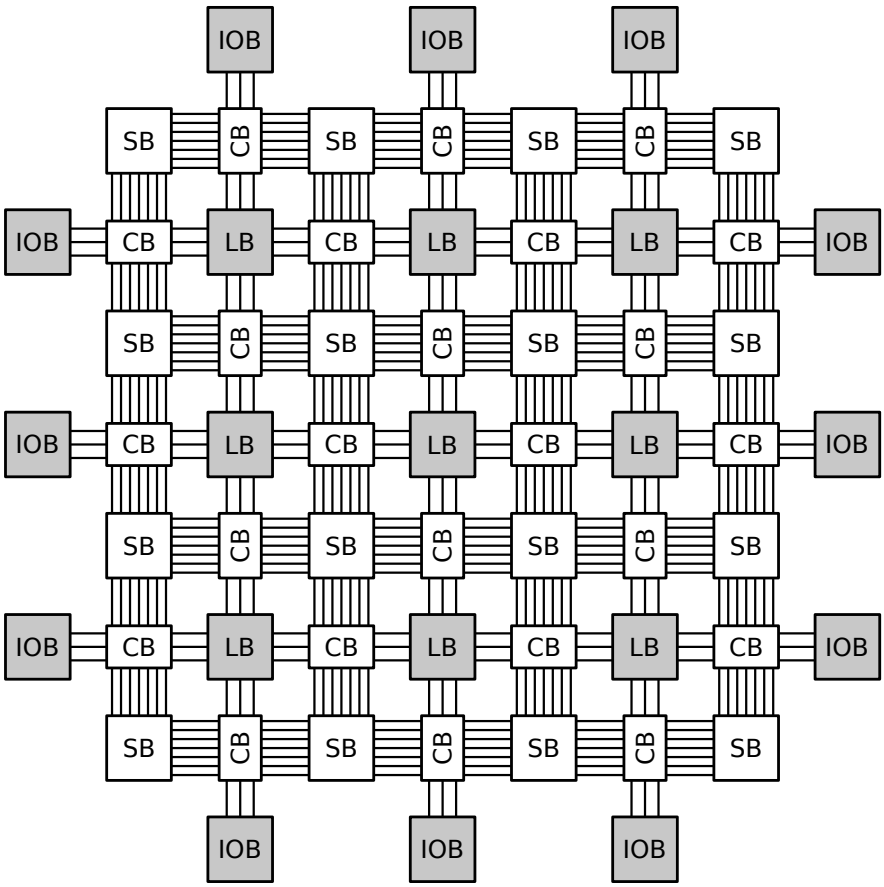


Figure 2.2: Schematic view of an island style FPGA fabric, showing I/O blocks (IOB), logic blocks (LB), switch blocks (SB) and connection blocks (CB).

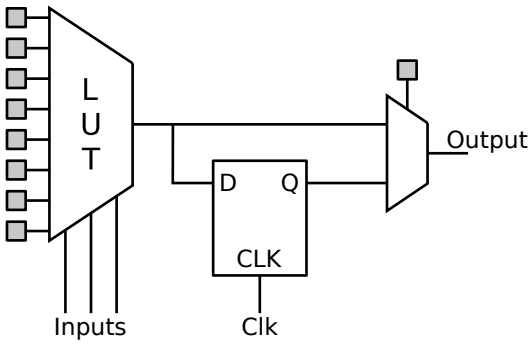


Figure 2.3: Basic logic block containing a 3-input LUT and a FF. Configuration memory elements are shown in grey.

Besides LUTs, logic blocks contain FFs so that sequential circuits can be implemented. Typically every LUT is accompanied by a FF, as shown in Figure 2.3. The FF can be bypassed, so that several LUTs can be combined to form larger blocks of combinational logic.

Modern FPGAs have complex logic blocks that contain several LUT/FF combinations [2] that are connected by programmable local interconnect. The architecture targeted by the placement and routing tools described in this thesis, however, contains only one LUT/FF combination. Again, it should be easy to adapt the concepts and algorithms described in this thesis to more complex logic blocks.

Programable Routing

The programmable interconnect basically consists of a number of wires that are connected by routing switches. The switches are controlled by bits in the configuration memory. There are many possible ways of organizing the programable interconnect. Figure 2.2 shows an island style FPGA, as is for example used in Xilinx FPGAs [88]. Altera FPGAs use a more hierarchical interconnect structure [82]. In this thesis I will use island style FPGAs in the examples, but the techniques that will be described are not limited to this organization.

As can be seen from Figure 2.2, the wires in an island style FPGA are organized as channels in between the logic blocks. The switches that connect the wires and the logic block pins are aggregated as connection blocks and switch blocks. The connection blocks connect the logic block pins to the wires in their neighboring channel while the

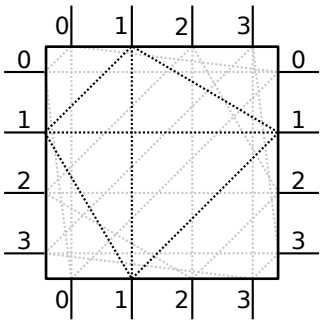


Figure 2.4: Schematic view of a disjoint switch block. Possible connections between incoming wires are shown in dotted lines. The possible connections of wires with rank 1 are highlighted.

switch blocks connect the wires from one channel to wires from an adjacent channel.

The design of switch blocks and connection blocks is an area of research by itself [61, 69, 19, 72, 47]. In this thesis, I use fully connected connection blocks and disjoint switch blocks. Fully connected connection blocks allow a logic block pin to connect to every wire of its neighboring channel. If the wires in a channel are numbered, a disjoint switch block [47] only connects wires with the same number (Figure 2.4). Again, our work can be easily extended to other routing switch configurations.

Finally, the distribution of the lengths of the wire segments can be chosen [7]. However, our simple architecture makes use of only length 1 wire segments, although this research work is not restricted to such wire architectures.

2.1.2 The Configuration Memory

In this section, I discuss several aspects of the configuration memory that are important in the light of dynamic reconfiguration.

Memory Technology

There are many technologies possible for implementing the configuration of an FPGA. In most commercial FPGAs (Xilinx Virtex family [94] and Altera Stratix [83]) SRAM is used to implement the configuration memory, but other technologies like flash memory (Actel

Igloo [78]) and antifuse (Actel Axcelerator [77]) are also used. Antifuse FPGAs are of course not an option for use with dynamic reconfiguration since they can be programmed only once. Both flash-based and SRAM-based FPGAs can be used with dynamic reconfiguration. SRAM-based FPGAs are preferred because their reconfiguration speed is higher and they are more widely available.

Dynamic Partial Reconfiguration

The reconfiguration granularity of an FPGA refers to the smallest part of its FPGA fabric that can be reconfigured without reconfiguring the rest of the fabric. FPGAs that don't need to be reconfigured completely are partially reconfigurable. If an FPGA can be partially reconfigured while other parts of the FPGA continue execution, this is called dynamic partial reconfiguration (DPR). DPR has been available for the Xilinx Virtex family for several years. Early 2010 Altera announced that DPR will be possible in there new 28 nm FPGAs [3], which are now available.

In the Virtex family, the grains of reconfiguration often don't coincide with the grains of the FPGA fabric. This is because of the organization of the configuration memory. In the Virtex family, the smallest addressable group of configuration bits is called a frame. The frames of a Virtex-II Pro contains configuration bits that span the entire height of the FPGA [85]. Thus, if one wants to reconfigure a specific logic block of this FPGA one will automatically need to reconfigure an entire column of logic blocks. In the Virtex-5, this was improved by limiting the height of a frame to 20 logic blocks [93].

This different reconfiguration granularity does not mean that it is impossible to change part of a column without interfering with the operation of the rest of the column. If configuration bits in the Virtex family are overwritten with the same value that is already stored in configuration memory, there are no transitions or glitches on the associated signal that controls the FPGA fabric. This means that when one wants to reconfigure a logic block the other logic blocks in the same column can stay operational although they will also be reconfigured. One just needs to make sure that the configuration bits of the other logic blocks are not changed. It is thus possible to change part of the configuration of a column but it comes at the cost of writing a bigger number of configuration bits than strictly needed or in other words a longer reconfiguration time.

There is one problem with what is described in the previous paragraph. Volatile bits that are actually part of the FPGA fabric are also mapped in the configuration memory of the Virtex FPGAs. This is for example the case for the FFs of the logic blocks, but also for distributed RAM, shift registers and BRAM [93]. By mapping these bits in the configuration memory space, they can be initialized during configuration. The problem occurs when part of a frame needs to be reconfigured while it contains a volatile bit. The volatile bit should be overwritten with its current value, but there is no way of obtaining the current value. It should therefore be avoided that there are volatile bits mapped in frames that need to be reconfigured. This can be done with AREA_GROUP constraints [86]. In the Virtex-5 it is possible to mask all volatile configuration bits so that they are not rewritten during reconfiguration. Thus, this problem is solved for the Virtex-5.

The Configuration Interface

In the ideal situation for dynamic reconfiguration, the configuration memory would be organized as a conventional memory with a standard bus interface so that the configuration memory is directly addressable by a CPU. This was the case for old FPGAs like the Xilinx XC6200 [24]. However, because of the limited success of dynamic reconfiguration at that time, FPGA manufacturers have focussed on configuration interfaces that are optimized for low cost instead of fast and flexible reconfiguration. Today, there are no commercial FPGAs available with a standard bus interface as configuration interface.

Contemporary FPGAs are configured by writing a bitstream to the configuration interface. Normally this is done bit-serial, but many FPGAs have parallel configuration modes (8-bit for Virtex-II Pro [85] and 32-bit for Virtex-5 [93]) that allow faster configuration.

A bitstream is a sequence of commands and data that is processed by the configuration control logic of the FPGA. In its simplest form, a configuration bitstream starts with a command that sets the address of the first configuration word that should be written, followed by a command that sets the number of configuration words that will be written and ending with the actual data that needs to be written. In the Virtex family, there is an important difference between regular bitstreams and bitstreams that are intended for dynamic partial reconfiguration [85, 93]. Regular bitstreams contain commands

that shutdown the entire FPGA fabric before the actual reconfiguration starts and restart it afterwards. For obvious reasons, these commands are not present in bitstreams that are intended for dynamic partial reconfiguration.

The configuration memory of the Virtex family FPGAs can also be accessed from within the FPGA fabric through a configuration interface which is called the Internal Configuration Access Port (ICAP). The ICAP can be used to build a complete dynamically reconfiguring system using one Virtex FPGA, which is the approach I used in my experiments. The same thing can be done with FPGAs that don't have an ICAP, but in that case this should be taken into account during the PCB layout. This is explained in [4] for Spartan-III FPGAs.

Shift Register LUTs

The basic building block of the Virtex-II Pro FPGA is not a simple LUT, as described in Section 2.1.1, but a Shift Register LUT (SRL). An SRL is a LUT in which the 16 configuration bits are arranged as a shift register of which the input and the output are accessible from within the FPGA fabric (Figure 2.5). Therefore the truth table configuration bits can not only be changed through the configuration interface of the FPGA but also by shifting a new truth table in the SRL. As will be explained in detail in Section 6.3, this form of reconfiguration naturally fits the tool flow described in Chapter 4. This technique enables very fast reconfiguration because of two reasons. First, fewer bits need to be reconfigured because the smallest addressable group of configuration bits now becomes one truth table instead of an entire frame. Second, multiple SRLs can be configured at the same time. Unfortunately the fraction of LUTs that can be used as SRL has dropped from 100% in the Virtex-II Pro to only 25% in the Virtex-5.

Multi-context Configuration Memories

Up to now I have discussed the configuration memory of conventional FPGAs that are not optimized for dynamic reconfiguration. In the light of dynamic reconfiguration, an attractive improvement of these architectures would be the introduction of multi-context configuration memories [68, 29, 18, 65]. In multi-context FPGAs, the configuration inputs of the fabric are not directly connected to a bit in the

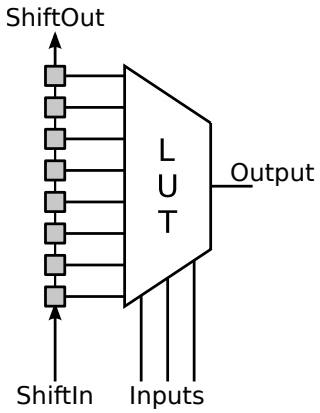


Figure 2.5: Schematic view of a 3-input Shift Register LUT (SRL).

configuration memory. Instead they are connected to multiple configuration bits by means of a multiplexer. In this way, the functionality implemented by the FPGA can be changed by simply changing the values on the select inputs of the multiplexers. Not only does this allow rapid switching between functionalities, but more importantly for this thesis, a new configuration can be loaded while an other is still operational, thus hiding the reconfiguration overhead (see Section 3.2). Of course this is only possible if the next configuration is known before the current one ends. The drawbacks of multi-context FPGAs are the silicon area needed to implement the extra context [29] and the possible peak of dynamic power upon a context switch.

Recently, a company called Tabula released the first commercial multi-context FPGA [84], which is similar to the much older DPGA [30]. The Tabula FPGA has eight contexts or folds. Every clock cycle the next context is loaded. One cycle of the eight contexts is called a user cycle. Basically, instead of evaluating the boolean logic that needs to be implemented by the FPGA in one step it is evaluated in eight steps. For this type of reconfiguration it is relatively easy to automate the tool flow by adapting the place and route tools of a conventional tool flow. The concepts described in this work are orthogonal to Tabula's reconfiguration. It should thus be easy to transfer the concept of parameterized configurations to Tabula FPGAs.

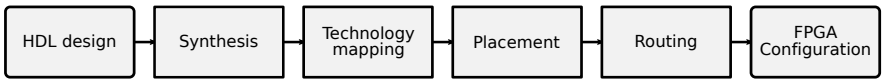


Figure 2.6: Conventional FPGA tool flow.

2.2 Overview of the Conventional FPGA Tool Flow

The aim of an FPGA tool flow is to produce a configuration for the target FPGA that implements the functionality described in the HDL design that serves as input to the flow. This process of mapping an HDL design to an FPGA is typically done in 4 steps: synthesis, technology mapping, placement and routing (see Figure 2.6). In the synthesis step the HDL code is translated from a human readable form to a gate-level logic circuit. The synthesis tool is also responsible for optimizing the circuits depending on the needs of the designer. During my research I used `quartus_map` for the translation of the HDL design to a gate-level circuit and ABC [5] for further optimization of the circuit. In contrast to other commercial synthesis tools, Quartus can produce a .blif file [79] that can be used as input for academic tools like ABC. Technology mapping translates the gate-level circuit to a circuit that only contains logic blocks that are available in the target technology. In the case of FPGAs these blocks are basically LUTs and FFs. The tool ABC contains several technology mappers that can be used for this purpose. During placement the abstract logic blocks in the mapped circuit are assigned to physical logic blocks on the FPGA and during routing, the switches in the routing infrastructure are controlled in such a way that the interconnection between these logic blocks is made as is described by the mapped circuit. The academic tool mostly used for place and route is called VPR [6, 7].

Except for the synthesis step, the algorithms used in all these steps are adapted for dynamic circuit specialization in the following chapters. Therefore, it is necessary to have a detailed understanding of the basic algorithms used in these design steps. In Section 2.3, Section 2.4 and Section 2.5.2, I give detailed descriptions of the algorithms used for technology mapping, placement and routing. These algorithms are used as a basis for the adapted algorithms that are described in later chapters. The algorithms are not the most optimized algorithms available, but they contain the basic concepts that are also

used in state of the art algorithms. The reason why I chose these simple algorithms was to control the complexity of my endeavor, which had the goal to prove that it is possible to automatically generate good quality parameterized configurations in a reasonable time and not in the first place in the most efficient way possible. Moreover, up to now I have not seen any reason why the optimizations that are applied to the basic algorithms could not also be applied to the adapted algorithms that are described later.

2.3 Conventional Structural Mapping

In this section I review the basics of LUT mapping. I first define and introduce the data structures that are used in technology mapping (Section 2.3.1), followed by the problem definition (Section 2.3.2) and a description of a simple structural mapping algorithm (Section 2.3.3).

2.3.1 Definitions and Data Structures

The input of our structural mapper is the combinational part of the circuit produced by the synthesis step. It is represented as an and-inverter graph (AIG) [42]. This is a directed acyclic graph (DAG) $G = (N, E)$. A node in the graph $n \in N$ represents a two-input AND gate, a combinational input (CI) or a combinational output (CO). A directed edge in the graph $(u \in N, v \in N, i \in \{0, 1\}) \in E$, represents a connection in the logic circuit with an input of gate v as sink and the output of gate u as source. i indicates whether the connection is inverted (1) or not (0).

Figure 2.7 depicts the AIG for a 4-to-1 multiplexer example. The internal nodes that represent two-input AND gates are represented by large circles. The smaller circles represent the CIs and COs of the AIG. Inverted edges have small circles as arrowhead while non-inverted edges have a standard arrowhead. The AIG shown in the figure is not the minimal size AIG for a 4-to-1 multiplexer, but it is better suited to illustrate the concepts and reasoning in what follows.

A cone of node n, C_n, is a subgraph of the AIG consisting of n and some of its predecessors that are not CIs, such that any node $u \in C_n$ has a path to n that lies entirely in C_n [52]. The set of input edges $iedge(C_n)$ of a cone C_n, is the set of edges with head in C_n and tail outside C_n, and the set of output edges $oedge(C_n)$ is the set of edges

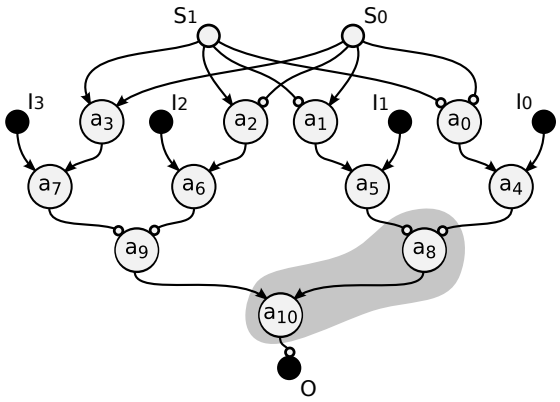


Figure 2.7: AIG of a 4-to-1 multiplexer. Large circles represent internal nodes. Smaller circles represent combinational inputs and output. Inverted edges have a circular arrowhead while non-inverted edges have a standard arrowhead.

with n as a tail. The set of input nodes $inode(C_n)$ of a cone C_n, is the set of distinct nodes that are the tails of $iedge(C_n)$. A cone C_n is called K-feasible if the number of input nodes $|inode(C_n)|$ is smaller than or equal to K.

A cone C_n of a node n is uniquely specified by the ordered pair $(n, inode(C_n))$. $inode(C_n)$ is also called the cut of cone C_n. When n and $inode(C_n)$ are given, the nodes that are part of C_n can be found by backtracking from n towards the nodes in $inode(C_n)$. All the nodes that are visited during this process are part of C_n.

In Figure 2.7, the subgraph consisting of nodes a_8 and a_{10} is a cone of node a_{10}. The input edges of this cone are $(a_4, a_8, 1)$, $(a_5, a_8, 1)$ and $(a_9, a_{10}, 0)$. The output edge of the cone is $(a_{10}, O, 1)$. The cut of this cone is $\{a_4, a_5, a_9\}$. This cone of node a_{10} is uniquely specified by the ordered pair $(a_{10}, \{a_4, a_5, a_9\})$. Because the number of its input nodes is smaller than or equal to 3 the cone is 3-feasible.

2.3.2 Problem Definition

During technology mapping the gate-level circuit produced by the synthesis step is mapped on the resources (K-LUTs) available in the target FPGA architecture.

Since a K-input LUT can implement any Boolean function with up to K arguments, the conventional structural mapping problem

reduces to selecting a minimal-cost set of K-feasible cones to cover the input graph. This means that every edge of the input DAG lies entirely within one of the selected cones or is an output edge of one of the selected cones [52]. There are many possible coverings for each input DAG. The task of the mapper is to find a covering with near to minimum cost. In depth-oriented mapping this cost is the longest path through the cones of the covering. In area-oriented mapping the cost is the number of cones in the covering. In this work, I focus on depth-oriented mapping, but the techniques described can be easily extended to other mapping criteria.

2.3.3 The Algorithm

Just like most structural mapping algorithms [36], the mapper described here is based on a dynamic programming technique. These algorithms typically consist of the following steps:

Cone enumeration: Calculates all K-feasible cones of the input circuit.

Cone ranking: Selects the lowest-cost cone for each node.

Cone selection: Selects a subset of the best cones to form the final covering.

Mapping solution generation: Generates the LUT structure and the truth tables.

Cone Enumeration

During cone enumeration all K-feasible cones of every node in the DAG are enumerated. This is done with a dynamic programming algorithm [60] that traverses all nodes of the input circuit in topological order starting from the CIs and going towards the COs. This means that for every edge in the input DAG, $(u, v) \in E$, u is visited before v is visited.

The set of K-feasible cones of node n is denoted $\zeta(n)$. Since all cones in $\zeta(n)$ have the same root, it is not efficient to store $\zeta(n)$ as a set of cones. Therefore $\zeta(n)$ is stored as an ordered pair $(n, \Phi(n))$ where $\Phi(n)$ is the set of cuts of the cones of n.

$\Phi(n)$ is generated by combining the cut sets of n's left and right predecessors (n_l and n_r), adding the trivial cut and retaining only the

Table 2.1: Intermediate results of mapping the AIG represented in Figure 2.7 to an FPGA architecture containing 3-input LUTs with a conventional mapper.

Node n	Cut set $\Phi(n)$	Best cut $bc(n)$	Depth $depth(bc(n))$	Area flow $af(bc(n))$
a_0	$\{\{a_0\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_1	$\{\{a_1\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_2	$\{\{a_2\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_3	$\{\{a_3\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_4	$\{\{a_4\},\{a_0I_0\},\{I_0S_0S_1\}\}$	$\{I_0S_0S_1\}$	1	1
a_5	$\{\{a_5\},\{a_1I_1\},\{I_1S_0S_1\}\}$	$\{I_1S_0S_1\}$	1	1
a_6	$\{\{a_6\},\{a_2I_2\},\{I_2S_0S_1\}\}$	$\{I_2S_0S_1\}$	1	1
a_7	$\{\{a_7\},\{a_3I_3\},\{I_3S_0S_1\}\}$	$\{I_3S_0S_1\}$	1	1
a_8	$\{\{a_8\},\{a_4a_5\},\{a_0a_5I_0\},\{a_1a_4I_1\}\}$	$\{a_4a_5\}$	2	3
a_9	$\{\{a_9\},\{a_6a_7\},\{a_2a_7I_2\},\{a_3a_6I_3\}\}$	$\{a_6a_7\}$	2	3
a_{10}	$\{\{a_{10}\},\{a_8a_9\},\{a_4a_5a_9\},\{a_6a_7a_8\}\}$	$\{a_4a_5a_9\}$	3	6

K-feasible cuts. The trivial cut of a node n is equal to $\{n\}$. The cone enumeration process is formally represented as

$$\Phi(n) = \begin{cases} \{\{n\}\} & \text{if } n \in \text{CI} \\ \{\{n\}\} \cup M(n) & \text{otherwise} \end{cases}, \tag{2.1}$$

where

$$M(n) = \{c_l \cup c_r | c_l \in \Phi(n_l), c_r \in \Phi(n_r), |c_l \cup c_r| \leq K\}. \tag{2.2}$$

The intermediate results of applying conventional structural mapping to the AIG shown in Figure 2.7 to an FPGA architecture containing 3-input LUTs are shown in Table 2.1. The second column shows the cuts of all 3-feasible cones found by the cone enumeration process described above.

Cone Ranking

In the cone ranking step the nodes are again processed in topological order from the CIs to the COs. For each visited node n the best non-trivial cone $bc(n)$ is selected from $\Phi(n)$ according to the mapping criterion. In depth-oriented mapping the cone with the lowest depth (Equation (2.3)) is selected. If several cones have the same depth the one with the lowest area flow (Equation (2.4)) is selected. This mapping criterion leads to a depth optimal mapping solution [26].

$$depth(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ 1 + \max\limits_{u \in inode(C_n)} (depth(bc(u))) & \text{otherwise} \end{cases} \quad (2.3)$$

$$af(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ 1 + \sum\limits_{u \in inode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{otherwise} \end{cases} \quad (2.4)$$

The best cones and their associated depth and area flow for each of the nodes in the multiplexer example of Figure 2.7 are shown in the third, the fourth and the fifth column of Table 2.1, respectively.

Cone Selection

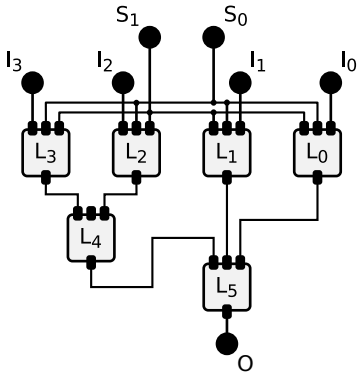
In the cone selection step a subset of the best cones is selected as the final covering of the graph. This is done by traversing the nodes of the graph in topological order but now starting from the COs and moving towards the CIs. First the best cones of the nodes driving the COs are selected and then the best cones of the input nodes, $inode(.)$, of the selected cones are added until the CIs are reached. The selected cones form a cone covering of the input circuit. The selected cones for our example are: $(a_{10}, \{a_4 a_5 a_9\})$, $(a_9, \{a_6 a_7\})$, $(a_7, \{I_3 S_0 S_1\})$, $(a_6, \{I_2 S_0 S_1\})$, $(a_5, \{I_1 S_0 S_1\})$ and $(a_4, \{I_0 S_0 S_1\})$.

Generating the Mapping Solution

During this last step the LUT structure and the truth tables for the LUTs are generated. The LUT structure is found by introducing one LUT for each of the cones in the covering and connecting these LUTs in the same way that their corresponding cones are connected. The truth tables are found by analyzing the functionality of the cone. An entry in the truth table can be found by evaluating the functionality for the entry's corresponding input value. The LUT structure for our example is found in Figure 2.8 together with the truth tables for each of the LUTs.

2.4 Conventional Placement

In this chapter I describe the wirelength-driven placement tool that is implemented in VPR [6, 7]. This simple placer is later used as a starting point to build our own TPLACE placer.



Entry	L_0	L_1	L_2	L_3	L_4	L_5
000	0	0	0	0	1	1
001	1	0	0	0	0	1
010	0	0	0	0	1	1
011	0	1	0	0	0	1
100	0	0	0	0	0	0
101	0	0	1	0	0	1
110	0	0	0	0	0	1
111	0	0	0	1	0	1

Figure 2.8: The final result of mapping the AIG represented in Figure 2.7 with a conventional mapper ($K = 3$).The LUT structure (left) and associated truth tables (right).

2.4.1 The FPGA Placement Problem

An FPGA placement algorithm takes two inputs: the mapped input circuit and a description of the target FPGA architecture. The algorithm searches a legal placement for the functional blocks of the input circuit so that circuit wiring is optimised. In a legal placement every functional block is associated to (placed on) one of the physical blocks (without overlap) that is capable of implementing the functional block.

For our simple target architecture (Section 2.1) there are only three types of functional blocks (inputs, outputs and LUTs) and two types of physical blocks (IOBs and LUTs). The inputs and the outputs can be placed on the IOBs while the functional LUTs can be placed on the physical LUTs.

The main optimisation goal used by placement tools is to minimise 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 [66] and timing-driven placers [53] 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. For the sake of simplicity, the tools used in this thesis are wire-length-driven.

Finding a high quality placement is very important, because poor quality placements generally cannot be routed or lead to low operation frequencies and high power consumption. Worse, the place-

ment 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 [51], analytic placers [17] and simulated annealing placers [6]. Because placers based on simulated annealing are the most used for FPGA placement, the focus lies on this type of placer below.

2.4.2 Simulated Annealing

Simulated annealing [40] is inspired on annealing of metals, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects.

The Basic Algorithm

The psuedo-code for our simulated annealing placer is shown in Figure 2.9. At any time, the algorithm keeps track of the placement cost. In a wire-length-driven placer this is the sum of the estimated wirelengths over all nets. The algorithm starts by randomly, but legally, placing the logic blocks in the input circuit on physical blocks of the FPGA architecture. Afterwards, the placer repeatedly tries to interchange the logic blocks placed on two randomly chosen physical blocks. Such an interchange is called a move. If the move causes a decrease in placement cost, the move is always accepted. If on the other hand, the move causes an increase in placement cost the move is accepted with a probability of $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost due to the move and T is a parameter called the temperature, which controls the probability by which hill-climbing moves are accepted. Initially, T is very high so that most moves are accepted. Gradually T is decreased so that the probability by which hill-climbing moves are accepted decreases. When the temperature is decreased in the proper way the result is a low cost placement. The hill-climbing moves allow the placer to escape from local minima.

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 of the annealer are called the annealing schedule. A good

```
function place(Netlist NL, Architecture A):
  Placement P = randomPlacement(NL, A)
  T = initialTemperature(NL, P)
  R_limit = initialR_limit(A)
  movesPerTemperature = InnerNum(N)^(4/3)

  while (T > 0.005 * averageNetCost(NL, P)):
    repeat movesPerTemperature times:
      move = findLegalMove(P, R_limit)
      ΔC = costAfter(move, NL, P) − currentCost(NL, P)
      if (ΔC < 0 or uniform(0, 1) < e^(−ΔC/T)):
        P = acceptSwap(move, P)
    T = updateTemperature(T)
    R_limit = updateR_limit(R_limit)

  return P
```

Figure 2.9: Pseudo code for a simulated annealing placer.

annealing schedule is crucial for finding a good solution in a reasonable amount of time.

The Annealing Schedule

In this section I describe the annealing schedule which is used in the VPR placement tool [7]. The same annealing schedule is used later to build our new placer called TPLACE.

To calculate the initial temperature, N moves are performed on the initial random placement, where N is the total number of logic blocks in the input circuit. At each move ΔC is calculated as the difference in placement cost before and after the move. The initial temperature is set to 20 times the standard deviation on ΔC. This temperature is so high that almost all moves are accepted at the start of the annealing. This way of calculating the initial temperature was first used by Huang et al [38].

As explained in [67], the number of moves attempted per temperature is set to

$$movesPerTemperature = InnerNum\ N^{\frac{4}{3}}. \tag{2.5}$$

The default value of $InnerNum$ is 10. This number can be used to change the execution time / placement quality tradeoff of the algorithm. Lowering $InnerNum$ reduces the execution time at the cost of a lower placement quality.

In the VPR placement tool the temperature is decreased as a function of the fraction of accepted moves, denoted by α. When α is high the functional blocks are almost moved randomly resulting in a limited improvement of the total cost. On the other hand, when α is low the improvement of the total cost is also low because very few moves are accepted. In [45, 67] it was shown the best results are obtained when α is kept near to 0.44 for as long as possible. Therefore, the temperature is decreased only slightly when α is in the vicinity of 0.44 and strongly otherwise. The authors of [7] realised the best performance with the temperature updating schedule of Equation (2.6).

$$T_{new} = \begin{cases} 0.5 \cdot T_{old} & \text{if } 0.96 < \alpha \\ 0.9 \cdot T_{old} & \text{if } 0.8 < \alpha \leq 0.96 \\ 0.95 \cdot T_{old} & \text{if } 0.15 < \alpha \leq 0.8 \\ 0.8 \cdot T_{old} & \text{if } \alpha \leq 0.15 \end{cases} \tag{2.6}$$

A second instrument to keep the α close to 0.44 is the range limit [45], which is denoted as R_{limit}. The range limit is the maximal distance in the x and y directions between the two locations of a selected move. When R_{limit} is decreased α increases, because the ΔC of a move is more likely to be small when the logic blocks that are interchanged are close together. Initially R_{limit} is set to the maximum FPGA dimension. At every temperature change, R_{limit} is updated as follows:

$$R_{limit}^{new} = R_{limit}^{old}(1 - 0.44 + \alpha), \tag{2.7}$$

which is then limited to the range $1 \leq R_{limit} \leq$ maximum FPGA dimension.

Finally, the annealing is terminated when T is a fraction of the average net cost (set to 0.5% in our case). At this temperature it is unlikely that the quality of the placement will improve because the probability that hill-climbing moves are accepted is almost zero.

Wire-length Estimation

As was mentioned in Section 2.4.1, wire-length-driven placement tools try to minimize the total wire length needed to route all wires

in a placement. They thus use the total wire-length as their cost function. 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 for every move tried in the inner loop of the placer and in this way exactly calculating the total wire-length, leads to very long execution times for the placer. Therefore, the cost is not exactly calculated but estimated.

A common way of estimating the total wire-length is shown in Equation (2.8). The estimate is 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 weighted by a factor which depends on the number of terminals of the net. The factor $q(.)$ is taken from [23]. It is equal to 1 for nets with up to three terminals and slowly grows to 2.79 for nets with 50 terminals.

$$Cost = \sum_{\forall n \in NETs} q(terminals(n))bb(n) \tag{2.8}$$

2.5 Conventional Routing: Pathfinder

In this section I describe a Negotiated Congestion Router called Pathfinder [55], as it is used in VPR [6, 7]. This router is later used as a base to build our TROUTE router in Chapter 7.

2.5.1 The FPGA Routing Problem

Once the placement algorithm has placed each of the logic blocks of the input circuit on a physical block of the FPGA architecture, the router 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 input circuit.

In a routability-driven router the goal is to simply find a legal routing solution while a timing-driven router tries to maximize the circuit clock frequency by allowing critical paths to use shorter and faster routing resources. For the sake of simplicity, the routers used in this theses are routability-driven.

Routing-resource Graph

PATHFINDER [55] uses a directed graph, called the *Resource Graph*, as a model for the routing architecture of an FPGA. Because this graph can be constructed for any useful routing architecture, the algorithm is very flexible.

The resource graph is a directed graph $C = (N, E)$, where the nodes N represent the routing resources. A directed edge (t, h) represents the possibility of routing a signal from resource t (the tail of the edge) to resource h (the head of the edge), by setting a switch. There are five types of routing resources: wire segments, input pins, output pins, sinks and sources.

Unidirectional switches which, when closed, force the logic value of resource i on resource o, are modeled as a directed edge (i, o). Bidirectional switches that connect resource r to resource s, are modeled by two directed edges (r, s) and (s, r). This model can be extended for other types of switches like multiplexers [7], but that is beyond the scope of this thesis.

The distinction between input-pin nodes and sink nodes on the one hand and output-pin nodes and source nodes on the other hand is made in order to model logic equivalence of input and output pins [7]. The inputs of a 4-input LUT for example are logically equivalent. It does not matter in which order the nets are connected to the input pins of the LUT. A simple permutation of the LUT's truth table preserves the functionality of the input circuit. This extra degree of freedom is important because it improves the routability of the FPGA. Logic equivalence of input pins is modeled in the routing resource graph by connecting each group of logically equivalent input pins to one sink node. If a net needs to be routed to one of the pins, the target of the router is not the pin node but the sink node. In this way the router will automatically choose which of the pins is most efficient to use. A similar mechanism is implemented with output pins and sources.

Figure 2.10 (a) shows the resource graph of a simple 2×2 island style FPGA with only length 1 wires 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. The sink and source nodes are not shown in the figure. For the sake of clarity I have also not drawn the individual edges. The thin lines each represent two edges, one for each sense.

When the routing architecture of the FPGA is represented as a

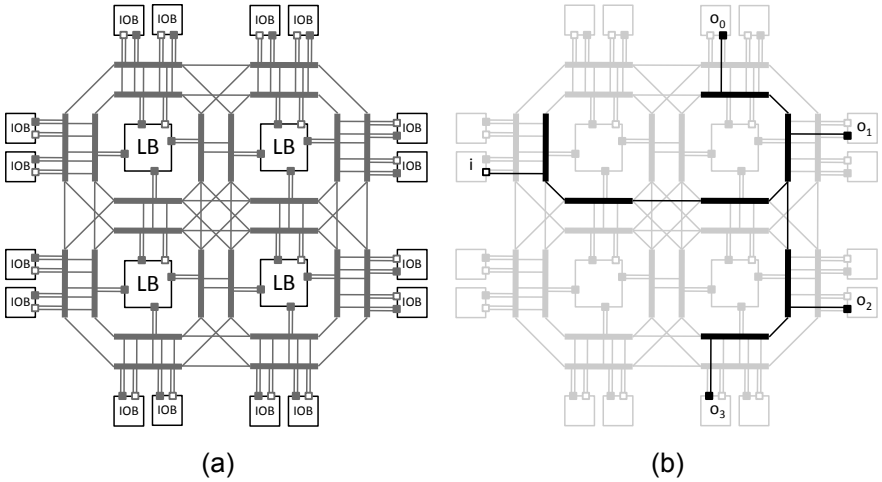


Figure 2.10: (a) Resource graph for a simple 2×2 island style FPGA. Wires are solid black lines; Edges are thin lines; Output pins are open boxes; And input pins are filled boxes. (b) Routing tree of a net $(i, \{o_0, o_1, o_2, o_3\})$.

routing-resource graph, the routing algorithm reduces to finding a subgraph of the routing-resource graph, called a routing tree, for each of the nets. These routing trees should be disjoint to avoid short circuits. Each routing tree should contain at least the source nodes and the sink nodes of its associated net and enough wire nodes so that a connection exists between the source node and each of the sink nodes. Figure 2.10 (b) shows the routing tree of a net $(i, \{o_0, o_1, o_2, o_3\})$.

2.5.2 The Pathfinder Algorithm

The main structure of the Pathfinder algorithm [55] is shown in Figure 2.11. In every routing iteration, the algorithm rips up and reroutes all the nets in the input circuit. These iterations are repeated until no shared resources exist [57] or, in other words, the routing trees of the nets are disjoint. This is achieved by gradually increasing the cost of sharing resources between nets. During the first iteration, nets can share resources at no extra cost and thus, each net is routed with a minimum number of wires. 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 shared heav-

```
while (sharedResourcesExist()):
  for each Net n do:
    n.ripUpRouting()
    route(n)
    n.resources().updateSharingCost()
  allResources().updateHistoryCost()
```

Figure 2.11: Pseudo code of the PATHFINDER algorihm (negotiated congestion).

ily in past routing iterations become more expensive. In this way a congestion map is built, which enables nets to avoid routing through heavily congested resources, if possible.

Negotiated Congestion

In this section I discuss the details of how the negotiated congestion mechanism updates the cost of the routing resources for a routability-driven router, as described in [7]. The cost of a node n is calculated as

$$cost(n) = b(n) \cdot p(n) \cdot h(n), \tag{2.9}$$

where $b(n)$ is the base cost, $p(n)$ is the present congestion penalty, $h(n)$ is the historical congestion penalty.

The base cost $b(n)$, as used in [7], is given in Table 2.2 for the five different node types. The quality of the router's result is not extremely sensitive to the exact values of the base cost, but by choosing the base cost of the input nodes and the sink nodes less than 1 the maze router (see next section) becomes faster.

The present congestion penalty, $p(n)$, is updated whenever a net is rerouted. The update is done as follows

$$p(n) = \begin{cases} 1 & \text{if } cap(n) > occ(n) \\ 1 + p_{fac}(occ(n) - cap(n) + 1) & \text{otherwise} \end{cases}, \tag{2.10}$$

where $cap(n)$ represents the capacity of the node and $occ(n)$ is the occupancy of the node. The capacity is the maximum number of nets that can legally use the routing resource. In our simple architecture all nodes have a capacity of 1, except for the sink nodes of the logic

Table 2.2: Base cost for different node types.

Node type	Base cost $b(n)$
wire segment	1
output pin	1
input pin	0.95
source	1
sink	0

blocks which have a capacity of 4. The occupancy of a node is the number of nets that are presently using it. The factor p_{fac} is used to increase the sharing cost as the algorithm progresses. This is explained below.

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

$$h^i(n) = \begin{cases} 1 & \text{if } i = 1 \\ h^{(i-1)}(n) & \text{if } cap(n) \geq occ(n) \\ h^{(i-1)}(n) + h_{fac}(occ(n) - cap(n)) & \text{otherwise} \end{cases} . \tag{2.11}$$

Again, the factor h_{fac} is used the control the impact of the historical congestion penalty on the total resource cost.

The way the factors p_{fac} and h_{fac} change as the algorithm progresses is called the routing schedule. Again I use the routing schedule proposed in [7]. In this schedule, h_{fac} is held equal to 1 independent of the iteration. On the other hand, p_{fac} is initially set to 0.5 and is doubled in every subsequent iteration.

Routing a Net

The task of the net router is to find a minimum cost routing tree for a given net in the resource graph. As mentioned before the routing tree should contain the source and the sinks of the net and a path from the source to each of the sinks. The cost of the resources is determined by the negotiated congestion mechanism.

The search space of all possible routing trees for a net is huge. Therefore a heuristic was developed that finds a low cost routing tree for a given net in a reasonable amount of time. The algorithm is shown in Figure 2.12. It is a variant of a maze router [46]. The

```
procedure routeNet(Net n):
  routingTree = {source}
  for each Sink s of n:
    path = dijkstra(routingTree, s)
    routingTree = routingTree ∪ path
```

Figure 2.12: Pseudo code for a maze router.

maze router loops over all the sinks of the net and extends the already found routing tree with the shortest path from this routing tree to the sink under consideration. The shortest path is found using Dijkstra's algorithm [33].

2.6 Conventional Dynamic Reconfiguration: Configuration Swapping

The tool flow that was described above can easily be adapted to support a form of dynamic reconfiguration which I will address as configuration swapping. In configuration swapping the functionality of an entire FPGA or of part of an FPGA is swapped with another pre-computed configuration that is loaded from a storage medium. FPGA manufacturers have adapted their tool flows for this type of reconfiguration. Xilinx has adapted its modular design flow [86] and a similar tool flow is available for Altera [3]. Unfortunately, it is not possible to use these configuration swapping tool flows to implement dynamic circuit specialization. The reason why is explained in Chapter 3.

The main difference for the designer is the fact that he will need 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 functionality present on the FPGA in exactly the same way so that the FPGA's configuration does not get corrupted. In the Xilinx flow this interface is defined using bus macros [86].

Next, the tool flow needs to generate the partial configurations for each of the functionalities and each of the areas. 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 that is described above is then run for each of the functionalities, with the exception that the placement and routing tools 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.

All these configurations are stored on a storage medium. When the system is online the system can fetch such a configuration and load it in the dynamic area when needed.

Chapter 3

Dynamic Circuit Specialization: What, Why and How?

As was explained in the introduction, the use of dynamic reconfiguration may for some applications lead to better area efficiency, higher performance and lower power consumption. Currently, the design tools of FPGA manufacturers only support configuration swapping, a form of dynamic reconfiguration where a limited number of functionalities are timeshared on the same piece of FPGA area. Dynamic circuit specialization is an other form of dynamic reconfiguration that is orthogonal to configurations swapping. Dynamic circuit specialization allows to specialize these functionalities for specific data. This may further reduce the area requirements, improve the performance and reduce the power consumption. In this chapter, I describe Dynamic Circuit Specialization (DCS) in detail. I explain the benefits and the costs involved in DCS and compare different existing implementations. I show that the current tools are inadequate to implement DCS systems and propose parameterized configurations as a first step towards a solution for this problem.

3.1 What is Dynamic Circuit Specialization?

Dynamic Circuit Specialization [34, 74] is a technique to dynamically specialize an FPGA configuration according to the values of a set of *parameters*. The general idea of DCS is that each time the parameter

values change, the device is reconfigured with a configuration that is specialized for the new parameter values. Since specialized configurations are smaller and faster than their generic counterpart, the hope is that the system implementation will be more cost efficient when using DCS.

At the system level, DCS can be seen as a technique to realize a communication channel between two subprocesses of a system, called the sender and the receiver, where the receiver is implemented on an FPGA. The parameters are the data that are sent over the DCS channel. This is illustrated in Figure 3.1. Process P is assigned to dedicated sets of FPGA resources and communication of the parameters is done by means of reconfiguration. This means that every time the sender process sends new parameters to the receiver a specialized version of the receiver is generated and loaded in the FPGA.

A good example of where DCS can be used is adaptive filtering. In adaptive filtering the filter coefficients are the parameters. Every time the filter characteristics need to change, new coefficients are sent over the DCS channel. More concrete, this means that every time the coefficients need to change a specialized filter configuration is generated and this configuration is loaded in the FPGA's configuration memory. The advantage here is that the specialized FIR filters are typically 3 times smaller than their generic counterparts (see Section 5.6.1). This is because constant multipliers can be used instead of bulky generic multipliers.

As was already mentioned, a DCS channel is an abstraction of functionality that will reconfigure a set of FPGA resources every time a new parameter value is sent over the DCS channel, so that the input/output relation of the set of FPGA resources is specialized for the new parameter. To implement these channels we will, on the one hand, need to select a set of FPGA resources that can implement the specialized functionality for each possible parameter value. On the other hand, we will need to implement a subsystem that, given a parameter value, calculates a configuration for these resources. These two communicating subsystems are shown in Figure 3.1. This can be seen as a partitioning of the functionality of process P into two subprocesses $P_{res.}$ and $P_{spec.}$. The subprocess $P_{spec.}$ is responsible for generating a configuration for the selected FPGA resources given a parameter value, while the subprocess $P_{res.}$ represents the functionality of these FPGA resources.

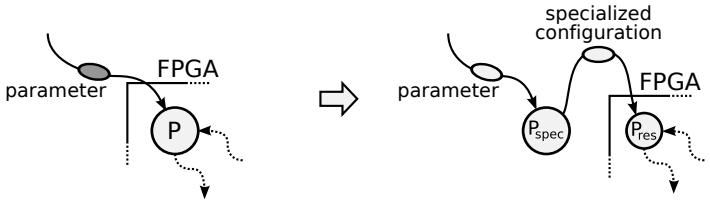


Figure 3.1: Refinement of communication by means of dynamic circuit specialization.

3.2 Why and When to use DCS?

The main advantage of using DCS to implement a process P, is the reduction of the number of expensive FPGA resources needed to implement P. This reduction is achieved because a specialized implementation, $P_{res.}$, of P will always require a smaller or equal amount of resources compared to the generic implementation $P_{gen.}$. From the partitioning point of view, the number of FPGA resources is reduced by splitting off part of the functionality of P_{gen} in $P_{spec.}$ and implementing this functionality in cheaper resources, e.g. cycles of a CPU. There are two secondary effects of reducing the number of FPGA resources. Firstly, reducing the number of FPGA resources will in some cases also reduce the number of resources in the longest path and thus the maximum clock frequency might decrease. Secondly, reducing the number of FPGA resources might also reduce the power consumption of the implementation.

The main disadvantage of DCS is the extra delay that is introduced between the moment a parameter changes and the moment this parameter change has effect on the FPGA processing. This delay is caused by the time needed for the specialization process, which includes calculation of the new configuration and reconfiguring the FPGA at run-time. The effect of the extra delay is illustrated in the timing diagram of Figure 3.2 (a). In some applications, the specialization process can run (partially) in parallel with the processing of the FPGA, e.g. when the new parameter value is known in advance or in the case a parameter change does not have to take effect immediately. I call the time that is available between the moment the new parameter is known and the moment processing with the old parameter stops the slack, T_{slack}. The situation with limited slack is illustrated in Figure 3.2 (b). If the available slack is larger than the

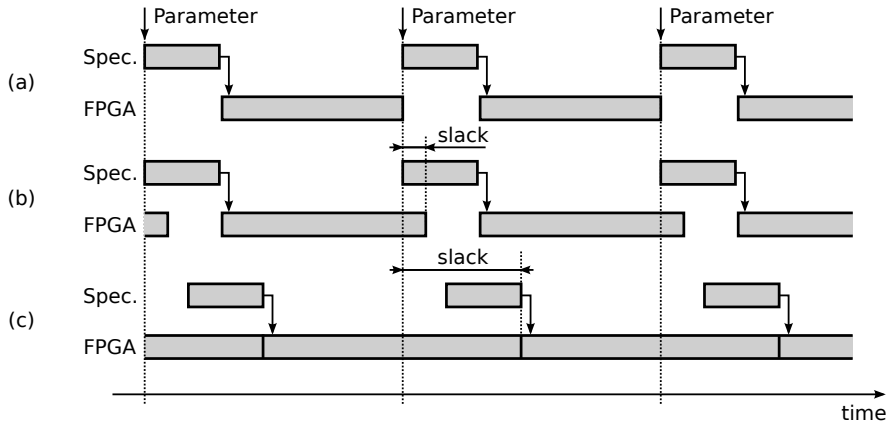


Figure 3.2: Timing diagram of a DCS system in different situations: (a) no slack (worst case), (b) limited slack and (c) abundant slack

time needed to specialize the FPGA, the idle time of the FPGA can be reduced to zero (Figure 3.2 (c)).

In commercially available FPGAs, reconfiguration is a sequential process. This means that not all configuration bits can be written in parallel. During the reconfiguration process some of the FPGA resources will already be configured for the new parameter value while others are still configured for the old parameter value. During this time, the behavior of the FPGA resources is unpredictable and they can therefore not be used for processing parallel to the reconfiguration process. This situation is represented in the timing diagram Figure 3.3 for two different cases. In the first case (Figure 3.3 (a)), a new configuration is first fully generated and only then is it written in the configuration memory. In this way the reconfiguration overhead is minimized. In the second case (Figure 3.3 (b)), once a reconfiguration word is generated it is written to the configuration memory. In this way memory is saved because the full configuration doesn't need to be stored. On the other hand the reconfiguration overhead is larger because FPGA resources cannot continue processing while a new configuration is generated.

The ratio between the total run-time of an application implemented using DCS, T_{DCS}^{total} , and the run-time of the same application without the use of DCS, $T_{gen.}^{total}$, is shown in Equation (3.1).

$$\frac{T_{DCS}^{total}}{T_{gen.}^{total}} = \frac{\hat{T}_{DCS}}{\hat{T}_{gen.}} = \frac{\widehat{max}(T_{spec.} - T_{slack}, 0)}{\hat{T}_{gen.}} + \frac{\hat{T}_{FPGA}}{\hat{T}_{gen.}} \tag{3.1}$$

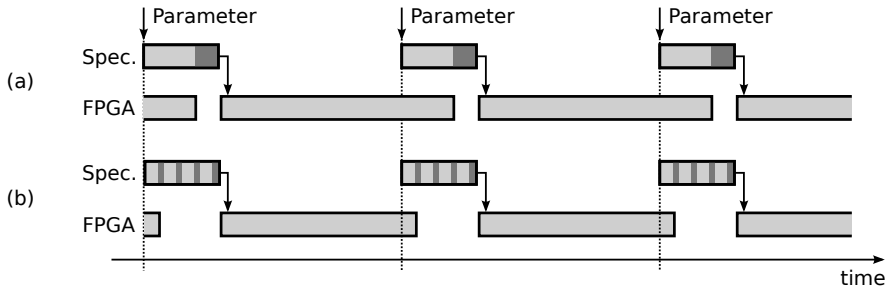


Figure 3.3: Timing diagram of a DCS system in the case of sequential reconfiguration. (a) postponed reconfiguration (b) immediate reconfiguration

This ratio equals the ratio of the average run time of the DCS implementation, $\hat{T}_{DCS}$, and the generic implementation, $\hat{T}_{gen.}$ of one parameter episode. The average run time of the DCS implementation can in its turn be expressed as a function of the specialization time $T_{spec.}$, the slack, T_{slack}, and the average FPGA run time, $\hat{T}_{FPGA}$. The first term of the right hand side of Equation (3.1) represents the relative run-time overhead of using DCS while the second term represents the gain in run-time because of the possible gain in clock frequency. In normal circumstances, the second term lies relatively close to 1. The first term, however, can become very large when the average time between parameter changes is small. Therefore, DCS is only useful for applications where the average time between parameter changes is a couple of orders of magnitude larger than the time needed to specialize the selected FPGA resources.

From Figure 3.2 one can see that when there is not enough slack available, the FPGA resources can become idle. This inefficiency in the use of FPGA resources is not taken into account when a DCS implementation is compared to a generic implementation in terms of area. A better measure for comparing DCS implementations in terms of usage of FPGA resources is the *functional density* [31], i.e. the number of computations per unit of area and per unit of time. It is defined as

$$D = \frac{N}{AT}, \tag{3.2}$$

where N is the total number of computations performed in time T using FPGA area A. The functional density of the DCS implementa-

tion can be written as

$$D_{DCS} = \frac{N}{A_{FPGA} T_{DCS}^{total}} \tag{3.3}$$

$$= \frac{1}{\frac{\widehat{max}(T_{spec.} - T_{slack}, 0)}{\hat{T}_{FPGA}} + 1} \quad \frac{N}{A_{FPGA} T_{FPGA}} \tag{3.4}$$

The second factor in Equation (3.4) represents the functional density in the case where enough slack is available. The first factor represents the relative degradation of the functional density because of the lack of slack. This factor goes to 0 if $\hat{T}_{FPGA}$ is small compared to $\widehat{max}(T_{spec.} - T_{slack}, 0)$ and grows to 1 if $\hat{T}_{FPGA}$ is large compared to $\widehat{max}(T_{spec.} - T_{slack}, 0)$. Again I must conclude that the average time between parameter changes should be a couple of orders of magnitude larger than the specialization time in order for DCS to benefit from the area reduction.

In order to compare different DCS implementations the functional density is expressed as a function of $\hat{N}$, the average number of computations performed in one parameter episode (Equation (3.5)).

$$D_{DCS} = \frac{1}{\frac{\widehat{max}(T_{spec.} - T_{slack}, 0)}{\hat{N} T_{FPGA}^{1}} + 1} \quad \frac{N}{A_{FPGA} T_{FPGA}} \tag{3.5}$$

T_{FPGA}^{1} represents the time needed to perform one computation.

3.3 How has Dynamic Circuit Specialization been Implemented Before?

Since mapping a hardware specification to FPGA resources is an NP-complete problem [64, 75] the specialization process will generate sub-optimal solutions. Therefore one can see that there will be a tradeoff between the resources spent on the specialization process and the quality of the specialized FPGA configuration. The more resources spent on generating the specialized functionality the fewer resources needed to implement the specialized functionality. This means that there is a range of Pareto-optimal implementations for the specialization process. Which of the implementations is the best choice, depends on the design specification.

As was mentioned in Section 3.1, implementing a DCS system requires the selection of a set of FPGA resources and the implemen-

tation of a specialization process that will specialize the selected resources given new parameter values. In most of the DCS methods the designer has to manually select a set of FPGA resources, but this thesis will significantly improve this process by automating it.

In order to automate the DCS implementation, I assume that the input to the design process is a RTL description of the generic functionality, where the parameters are inputs to the design. This means that the cycle by cycle scheduling of all specialized implementations is fixed to the scheduling in the original generic implementation. Actually, I don't know of any DCS implementation where the scheduling is specialized dynamically depending on the parameter values.

In what follows I briefly describe several ways of implementing DCS implementation. In order to make things more tangible I use the example of an adaptive FIR filter for each of the methods. In adaptive filtering the coefficients of the filter are the parameters. Changing the coefficients of the filter will thus be done by reconfiguring the FPGA. In section 3.3.6 I compare the methods in terms of their functional density. The RTL description of the generic functionality is shown in Figure 3.4 and the architecture is shown in Figure 3.5. As can be seen, the filter has 16 taps. The width of the input and the coefficients is 8 bit. The target FPGA architecture used for each of the implementations, is described in `4lut_sanitized.arch`. This is an FPGA architecture file which is included in the VPR distribution. It has logic blocks containing one 4-LUT and one flip-flop and the wire segments in the interconnection network only span one logic block. The specialization process was run on an Intel Core 2 processor running at 2.13 GHz with 2 GiB of memory.

3.3.1 The Generic Implementation as Comparison Base

The generic implementation is used as a base for evaluating DCS implementations.

The generic FPGA implementation is generated by running the RTL description of the generic functionality through a conventional FPGA tool flow. Therefore, the parameter inputs will be physical inputs of the implementation. Changing the parameter values thus only involves forcing the new parameter values on the parameter inputs of the generic hardware. This does not require any computation and thus the specialization time is zero, but the number of FPGA resources needed for the implementation is high compared to DCS

```
entity fir is
  port (
    clk          : in    std_logic;
    c            : in    coef_array; --PARAM
    i            : in    std_logic_vector (7 downto 0);
    o            : out   std_logic_vector (7 downto 0));
end fir;

architecture rtl of fir is
  type inter_array is array (0 to 15) of unsigned(31 downto 0);
  signal inter : inter_array;
begin
  process(clk)
  begin
    if clk'event and clk='1' then
      inter(0) <= unsigned(c(0))*unsigned(i);
    end if;
  end process;

  TAPS: for index in 1 to 15 generate
    process(clk)
    begin
      if clk'event and clk='1' then
        inter(index) <= inter(index-1) + unsigned(c(0))*unsigned(i);
      end if;
    end process;
  end generate TAPS;

  o <= intermediate(15);
end architecture rtl ;
```

Figure 3.4: The RTL description of the generic FIR filter written in VHDL.

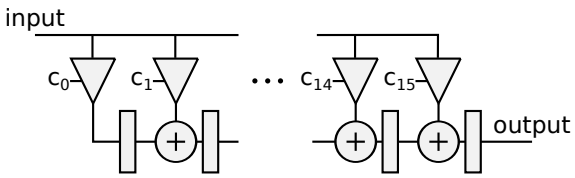


Figure 3.5: Architecture of the 16-tap FIR filter described in Figure 3.4.

implementations since it is not optimized for the parameter values at hand.

If the generic FIR filter is implemented using the conventional tool flow described in Chapter 2, 2999 4-LUTs are needed and the delay of the longest path is 118.4 ns.

3.3.2 Configuration Swapping

An obvious first approach to implementing DCS would be to use the tool flows for configuration swapping as provided by the FPGA vendors (see Section 2.6). In order to implement such a system, the designer needs to select a set of FPGA resources that are located in a rectangular area of the FPGA. These resources will be used to implement the specialized functionalities. The area must be large enough so that the worst case functionality in terms of resource usage fits in it. The designer also needs to design a fixed interface for the rectangular area through which the specialized functionalities can communicate with the outside world. Next, partial configurations need to be generated for each of the possible parameter values. These partial configurations need to be compatible with the selected area and its interface so that they can easily be loaded without corrupting the FPGA configuration. These partial configurations are generated by running a specialized RTL description through a constrained FPGA tool flow for each possible parameter value. One specialized RTL configuration is generated by merging a parameter value with the generic RTL description. The tool flow is constrained so that only the resources in the selected area are used and so that communication with the outside world is done through the predefined interface. The resulting configurations for all parameter values are stored in a database. Online, when a new parameter is sent over the DCS channel, the specialization process fetches the partial configuration that corresponds to the new parameter value and uses it to partially reconfigure the FPGA.

This approach works fine if the number of configurations is small. However, in DCS the number of parameter values grows exponentially with the number of bits needed to represent the parameter data. Hence, generating all configurations off-line and storing them in a database rapidly becomes infeasible for real-life applications. Therefore, the only option is to have the specialization process generate the partial configurations at run time.

In our FIR filter example, there are 16 taps which each have a 8-bit coefficient. In other words, there are 128 parameter bits. If there are no limitations on which coefficients are allowed, this will result in 2^{128} configurations. It is of course impossible to generate all these configurations beforehand and to store them.

3.3.3 On-line FPGA Tool Flow

If all configurations can not be generated beforehand and stored, they have to be generated at run-time. A first option to generate specialized configurations online is of course to have the specialization process merge the parameter value and the generic RTL description to form a specialized RTL description and then run it through the constrained tool flow.

On the one hand, implementing DCS in this way leads to a specialized configuration of very high quality, because all options for optimization are available in the tool flow. On the other hand, running a conventional FPGA tool flow is computationally very expensive. This makes the specialization time so high that only a few applications, which have very slowly changing parameters (see Section 3.2), can benefit from this type of DCS. This is for example the case in logic emulation [36].

The academic tool flow described in Chapter 2 takes on average 35634 ms to generate a specialized FIR filter configuration in our example. The time between parameter changes should be in the order of several minutes before this type of DCS becomes profitable. The generated configurations require on average 1005 LUTs and the average longest path equals 83.5 ns. The worst case filter required 1147 LUTs, so the selected area should at least contain this amount of LUTs and the worst case longest path had a delay of 90.4 ns.

As was explained in Chapter 2 the algorithms used in conventional FPGA tool flows are all heuristics. These algorithms search for an acceptable solution in vast solution spaces. A typical argument for a heuristic is the effort it should make in order to find a solution. If the effort is lowered the heuristic will find a solution in a shorter run time, but the quality of that solution will also be lower. By lowering the effort of the tool flow researchers have tried to trade some of the quality of the specialized configurations for a lower specialization time [63, 50]. However, the problem with this technique is that

the specialization time can not be lowered much before the quality of the specialized configuration drops unacceptably low.

3.3.4 Staged Compilation

The tool flows described in the previous section need a full RTL specification of the specialized functionality before the specialization process can start generating a specialized configuration. This RTL description is constructed from the generic RTL description, which is available at compile time, and new parameter values, which only become available at run time. As was discussed, generating a configuration from an RTL description is very compute intensive and therefore results in a large specialization time. However, since a large part of the specification (the generic RTL description) is available at compile time, one would expect that it should be possible to complete a large part of the mapping process at compile time, resulting in an intermediate result which can then be refined at run time when the parameter values become available. The hope is that this will result in a large reduction in specialization time since only the refinement step needs to be executed at run time. I call the technique *staged mapping* as in staged compilation, a similar technique used in software compilation. The tool flows described in Chapter 4 and 7 of this thesis use an advanced form of staged mapping, but they were not the first.

The idea of staged mapping was described in [32] where it was used to move part of the synthesis step from run time to compile time. Since most of the steps (synthesis, technology mapping, placement and routing) still needed to be done at run time, the reduction in specialization time was limited.

Constant Propagation, Compaction and Incremental Routing

I have used the staged mapping technique in my first attempt to build a DCS tool flow, which was described in [12]. In this tool flow (Figure 3.6), the intermediate result is the generic implementation, which is produced at compile time with a conventional FPGA tool flow. At run time, specialization is performed using the following steps: constant propagation, compaction and incremental routing (CPCIR). Each of these online steps corresponds to a compile-time step and builds on its results.

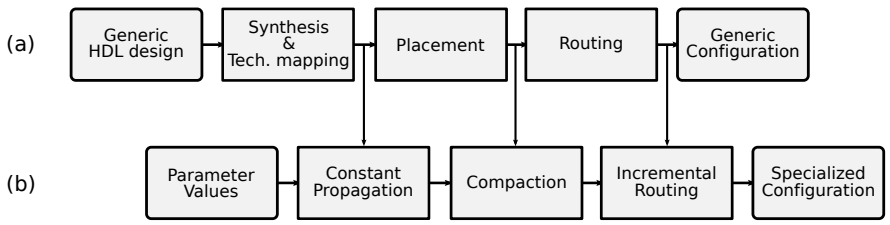


Figure 3.6: FPL2007 staged mapping tool flow. (a) Compile-time stage, (b) Run-time stage.

The constant propagation step uses the run-time information about parameter values to transform the generic netlist into a specialized netlist. This is done by propagating the constants into the circuit and simultaneously simplifying the circuit. Some LUTs are removed due to this simplification. The compaction step combines the generic placement and the specialized netlist and produces a compact specialized placement. This is done in two steps. First the generic placement is pruned only retaining the LUTs present in the specialized netlist. The emerging free space is fragmented and therefore cannot be used efficiently. Secondly this sparse placement is compacted to reduce the fragmentation, while trying to preserve the placement quality. The incremental routing step concludes the online hardware generation. Only those interconnections that were broken during constant propagation and compaction are rerouted. This last step was never completed in our first implementation. Instead it was replaced with a full routing step in the experiments.

The compaction step uses a fast online compacting heuristic while trying to preserve the placement quality. The generic placement is used as a condensed representation of the connectivity information of LUTs. This placement is produced by a placement algorithm that obtains good quality by placing strongly connected blocks closely together. The algorithm assumes that this property still holds for the sparse placement. Hence good placement quality can be achieved by keeping blocks that are closely together in the original placement closely together during compaction. The pseudo code for our compaction algorithm is shown in Figure 3.8. The algorithm first calculate the bounding box of the sparse placement. Then it iterates over all LUTs on the border of this bounding box and shifts them into the bounding box. This is done by finding the nearest free space in the bounding box and then sliding the logic blocks that are in be-

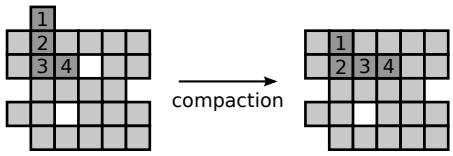


Figure 3.7: Principles of Compaction

```
procedure compact(Placement P):
  BoundingBox bb = P.findBoundingBox()
  LogicBlock lb = bb.firstLogicBlock()
  while (bb.freeSpace() != 0):
    Place free = bb.findNearestFreeSpace(lb)
    P.shift(lb, free)
    lb = bb.nextLogicBlock()
    if (lb == null):
      bb = findBoundingBox()
      lb = bb.firstLogicBlock()
```

Figure 3.8: Pseudo code of the compaction algorithm

tween at a right angle (Figure 3.7). By sliding a logic block over not more than one position every iteration, logic blocks that were closely together stay closely together during an iteration.

For the 16-tap FIR filter the total specialization time with this CP-CIR method amounts to 3932 ms. The constant propagation and the compaction steps only required 128 ms and 43 ms, respectively, while the online routing step required 3832 ms. The specialized FIR filters required on average 1491 4-LUTs and the longest path had a delay of on average 119.3 ns. The worst case filter required 1676 4-LUTs, so the selected area should at least contain this amount of LUTs. The worst case longest path had a delay of 140 ns.

3.3.5 Hand-crafted Approaches

The automatic tool flows described above all improved the specialization time, but as will be seen in this section, hand-crafted designs still result in far superior DCS implementations. Hand-crafted designs have been made for many applications: constant multiplica-

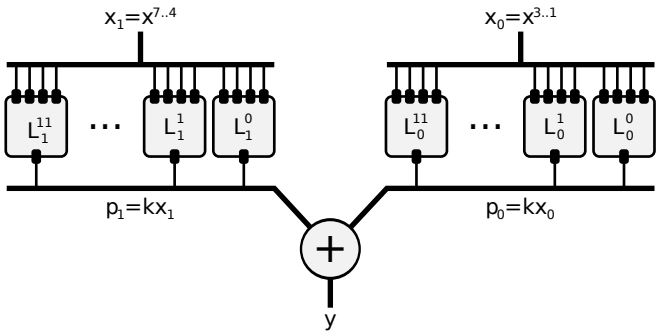


Figure 3.9: Architecture of an 8×8 KCM multiplier.

tion [20, 73], pattern matching [48, 54], Content Addressable Memory (CAM) [10],...

As an example, I will now explain how a reconfigurable constant $m \times n$ multiplier can be built using the KCM method [20, 28], afterwards this multiplier will be used to build a 16-tap adaptive filter. In the example the constant is denoted k, the variable input is denoted x and the output is denoted y. The main idea is to break up the variable input x into 4-bit nibbles x_i. In that case, the multiplication can now be written as a sum of partial products $p_i = kx_i$:

$$y = \sum_{i=0}^{\lceil \frac{n}{4} \rceil} kx_i 16^i. \tag{3.6}$$

Each of the $\lceil \frac{n}{4} \rceil$ partial products p_i can be implemented as a $16 \times m + 4$ look-up table, where x_i is the address. These look-up tables can be built using $m + 4$ 4-input LUTs L_i^j of the FPGA that each produce one bit p_i^j of the partial product. The final result is found by summing the partial products using an adder tree. Figure 3.9 shows the architecture of the multiplier.

The content of the LUTs depends on the constant k. From Equation (3.6) can be derived that: entry l of LUT L_i^j equals

$$L_i^j(l) = (jk)_l. \tag{3.7}$$

Using this equation and the location of each of the LUTs it is easy to build a specialization function that can specialize the architecture shown in Figure 3.9 for a given constant k. The specialization process

Table 3.1: Major results of five implementations of a 16-tap adaptive FIR filter with an 8-bit input and 8-bit coefficients.

Implementation	LUTs	Longest path [ns]	Reconf. time[ms]
Generic	2999	118.4	0
Online	1147	90.4	35634
CPCIR	1676	140.3	3932
Hand-crafted	1315	85.3	0.009
Param. Conf.	1301	86.3	0.166

will only involve executing $\lceil \frac{n}{4} \rceil$ multiplications and reordering the bits of the results in order to form the new truth tables for the LUTs. One can easily see that running this specialization procedure will be extremely fast compared to the techniques that were discussed above.

The handcrafted design of the 16-tap FIR filter requires an area of 1315 4-LUTs with a longest path of 85.3 ns. Calculating the new truth table contents given a set of coefficient values now takes only 9 μs.

3.3.6 Discussion

In the previous sections several DCS implementations were described. All of these DCS implementations were used to implement a 16-tap adaptive FIR filter with an 8-bit input and 8-bit coefficients. The results are summarized in Table 3.1. In this section I compare these methods in terms of their functional density (see Section 3.2). In Figure 3.10 the functional density for the generic implementation is plotted (Section 3.3.1), the method that uses the full tool flow online (Section 3.3.3), the CPCIR method (Section 3.3.4) and the handcrafted filter (section 3.3.5). Configuration swapping (Section 3.3.2)is not feasible due to the excessive memory requirements for this example. The functional density is plotted (using Equation (3.5)) as a function of the average number of samples, $\hat{N}$, that are processed between coefficient changes. Since 1 sample is processed in one clock cycle, T^1_{FPGA} equals the longest path of the design. I assumed that there is no slack available.

If we first consider only the automatic tool flows that can implement the DCS channel starting from an RTL description, we see that

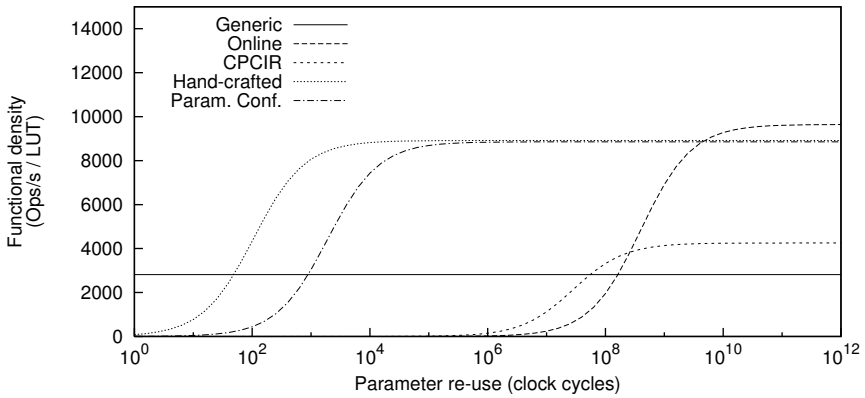


Figure 3.10: Illustration of the functional density (operations per second and per number of required LUTs) as a function of the rate of change of the parameter inputs, for a number of DCS implementation techniques.

each of them is the optimal solution in terms of functional density in an other range of $\hat{N}$. As can be seen from the figure, it is better not to use DCS if the filter coefficients change very often (left part of the figure). More specifically when the coefficients change in the range from every clock cycle to every 58 million clock cycles. However, if the coefficients change less often than every 58 million clock cycles, DCS starts outperforming the generic implementation. In the range between 58 million to 258 million cycles the best performing DCS implementation is the one that uses the CPCIR method. If the number of cycles between coefficient changes grows above 258 million running the full FPGA tool chain becomes profitable.

If we now also consider the hand-crafted DCS implementation of the adaptive FIR filter we see that it by far outperforms the CPCIR method. Using the hand-crafted design is profitable when the number of cycles between coefficient changes lies between 51 and 5 billion cycles. From this result we must conclude that there is a lot of room for improving the automatic tools.

3.4 DCS with Parameterized Configurations

As we have seen in the previous section, the performance of automatically implemented DCS systems are by far outperformed by hand-

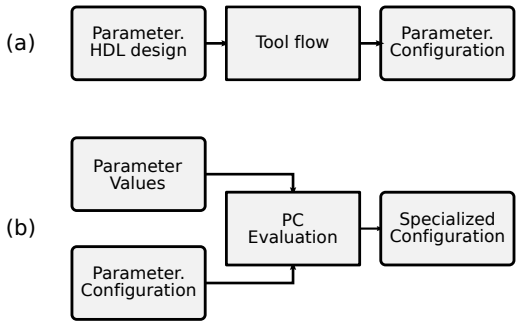


Figure 3.11: Staged compilation as used by the DCS techniques described in this thesis. (a) The generic stage of the tool flow (b) The specialization stage

crafted implementations in terms of specialization time. In this thesis, I will describe two methods (Chapter 4 and 7) that can automatically build DCS implementations with properties that are similar to those of a handcrafted implementation.

These methodss are built on the observation that the specialization process actually implements a multivalued Boolean function, which I call a *Parameterized Configuration* (PC). Indeed, both the input (a parameter value) and the output (a specialized configuration) of the specialization process are bit vectors.

The methods make use of staged compilation. During the generic stage (Figure 3.11(a)), a parameterized configuration is constructed and represented in a closed form starting from a parameterized HDL description. This stage is executed at compile time, when the parameter values are not known. During the specialization stage (Figure 3.11(b)), a specialized configuration is produced by evaluating the parameterized configuration given a parameter value. This stage is executed at run-time, after the parameter values have become available. After producing the specialized configuration, it is used to reconfigure the FPGA.

A parameterized HDL description is an HDL description in which distinction is made between *regular input ports* and *parameter input ports*.[1] The parameter inputs will not be inputs of the final specialized configurations. Instead, they will be bound to a constant value during the specialization stage.

[1]One should be careful not to confuse a parameter with a generic. A parameter is a special kind of input port.

It is important to note here that while the problem that needs to be solved by the generic stage of our staged compilation tool flow is computationally hard, the problem that needs to be solved by the specialization stage, evaluating the parameterized configuration, is not. This drastically reduces the specialization time compared to the other staged mapping tool flows.

When a 16-tap FIR filter is built with the method described in Chapter 4, it requires an area of 1301 4-LUTs and has a longest path of 86.8 ns. Evaluating the PC in order to calculate the new truth table contents given a set of coefficient values now takes only 166 μs. Figure 3.10 also shows the functional density for the parameterized configuration method. As can be seen, the new technique comes very close to the performance of the hand-crafted design and by far outperforms the CPCIR method. The only reason why the calculation of new truth tables is faster in the hand-crafted design is because the calculation makes use of the multiplication instruction of the CPU while this same multiplication is performed using logic operations when evaluating the PC.

Chapter 4

The TLUT method: Dynamic Reconfiguration of LUTs

In this chapter, I introduce a first method to automatically implement dynamic circuit specialization, called the TLUT method. It is able to generate a parameterized configuration starting from a parameterized HDL description. In the parameterized configuration produced by the TLUT method, only the truth table configuration bits are expressed as a Boolean function of the parameter inputs. All other configuration bits are static and will thus be part of the TC (template configuration). In Chapter 7, the TLUT method will be extended so that the routing bits can also be expressed as a function of the parameters. This new method will be called the TCON method. Although the TCON method can in some cases lead to more compact implementations, the TLUT method has the advantage that it can lead to very fast reconfiguration (see Section 2.1.2). This is because in the TCON method the bits that need to be reconfigured are scattered all over the configuration memory space, while in the TLUT method they are nicely clustered.

The problem faced by the TLUT method is to produce a parameterized configuration given a parameterized HDL description while minimizing some cost function. Similar to conventional FPGA map-

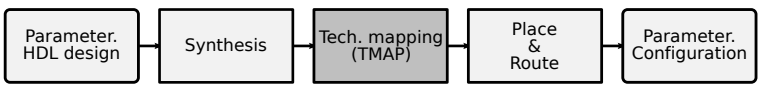


Figure 4.1: Overview of the TLUT method.

ping, we divide the generic offline mapping problem into four subproblems: synthesis, technology mapping, placement and routing. This can be seen in Figure 4.1. The algorithms used to solve the subproblems are adapted versions of the conventional algorithms (see Chapter 2). Since both the input and output of the method are parameterized, the internal data structures need to be adapted so that they are able to express the parameterizability. Also, the algorithms that transform the data structures need to be adapted so that they preserve the parameterizability. In the remainder of this chapter we give a conceptual description of how this is done for each of the data structures and algorithms. In Chapter 5, a detailed description is given of the adapted technology mapper, called TMAP, which is the most drastically changed algorithm.

4.1 Parameterized HDL Description

A parameterized HDL description is a regular HDL description in which a distinction is made between regular inputs and parameter inputs. In theory this requires an extension of the HDL syntax, but practically, we make the distinction by adding the comment line annotation `--PARAM` at the end of the input declaration in the entity declaration. This has the advantage that the parameterized HDL description can still be processed by conventional tools (resulting in a generic implementation as the parameters are treated as regular inputs).

Figure 4.2 shows the parameterized VHDL code of a 4-to-1 multiplexer where the select inputs are marked as parameters. We will use this multiplexer as an example throughout this chapter and the next.

4.2 Synthesis

Just like a conventional synthesis tool, the synthesis step in the TMAP method converts the parameterized HDL description into a gate-level circuit. The only difference is that the new synthesis tool should preserve the distinction between regular and parameter inputs. This can easily be done by allowing an extra type of input, a parameter input, in the data structure of the gate-level circuit. We call such a circuit a parameterized gate-level circuit. The only thing

```
entity multiplexer is
port (
  I   : in    std_logic_vector (3 downto 0);
  S   : in    std_logic_vector (1 downto 0); --PARAM
  O   : out   std_logic
);
end multiplexer;

architecture behavior of multiplexer is
begin
  O <= I(conv_integer(S));
end behavior;
```

Figure 4.2: The parameterized VHDL code of a 4-to-1 multiplexer where the select inputs are marked as parameters.

that needs to change in the synthesis tool, is that it should pass the information about the parameter inputs to the next step.

When the HDL description of Figure 4.2 is processed by the adapted synthesis tool we get the parameterized gate-level circuit that is represented as an AIG (And-Inverter-Graph), as is shown in Figure 4.3. In this figure, regular inputs are represented by black circles (I_0, I_1, I_2 and I_3) while the parameter inputs (S_0 and S_1) are grey.

Conceptually, changing the synthesis tool is not very challenging. However, it is practically impossible because the source code of commercial tools is not available. That is why we took another, more practical approach. This is shown in Figure 4.4. On the one hand we process the parameterized HDL description with a conventional synthesis tool which ignores the comment line annotations and produces a regular gate-level circuit. This circuit does not make any distinction between parameter inputs and regular inputs. On the other hand, the parameterized HDL description is analyzed with a program that looks for the `--PARAM` annotation and thus produces a list of all parameter input names. Together, the regular gate-level circuit and the list of parameter inputs contain the same information as a parameterized gate-level circuit. In practice, our synthesis setup passes both the circuit and the list, but conceptually this corresponds to a parameterized gate-level circuit that is passed.

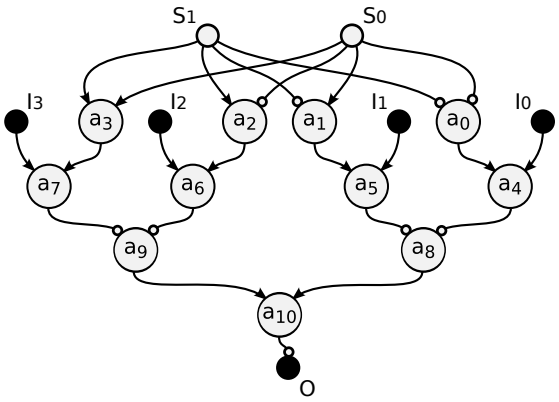


Figure 4.3: AIG of a 4-to-1 multiplexer. Large circles represent internal nodes. Smaller circles represent combinational inputs and output. Inverted edges have a circular arrowhead while non-inverted edges have a standard arrowhead.

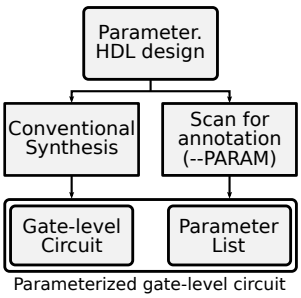


Figure 4.4: Practical setup for generating a parameterized gate-level circuit from a parameterized HDL description.

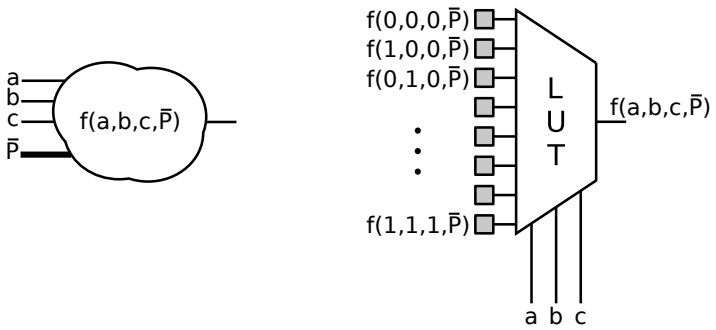


Figure 4.5: A K-input TLUT can implement any Boolean function with up to K regular inputs and any number of parameter inputs.

4.3 Technology Mapping

During conventional technology mapping the gate-level circuit produced by the synthesis step is mapped onto the resources available in the target FPGA architecture while minimizing a cost function. Basically, these resources are K-input LUTs and FFs. Mapping the FFs is straightforward, since every FF in the gate-level circuit is mapped on a FF in the mapped circuit. However, mapping the gates to LUTs is a complex problem. As explained in Section 2.3, a K-input LUT with a static truth table can implement any Boolean function with up to K arguments or, in other words, it can implement the functionality of a subcircuit of the gate-level circuit with one output and up to K inputs. These subcircuits are called K-feasible cones. Therefore, the conventional structural mapping problem reduces to selecting a set of K-feasible cones that cover the gate-level circuit.

The goal of the TLUT method is to produce a parameterized configuration where the truth table bits are function of the parameter bits. On the level of technology mapping, this means that we don't want to map to LUTs that have a static truth table, but to LUTs with a truth table expressed as a Boolean function of the parameter inputs. We call such LUTs tunable LUTs or TLUTs.

It is easy to see that a K-input TLUT can implement any Boolean function with up to K regular signals and any number of parameter signals as its arguments. Every entry of the truth table can be constructed by partially evaluating the given function for the input combination associated with this entry. This is illustrated in Figure 4.5.

In the context of structural mapping this means that a TLUT can

implement the functionality represented by a subcircuit with any number of parameter input nodes and up to K non-parameter input nodes. We call such a subcircuit a *K^*-feasible* cone. The TLUT mapping problem thus reduces to finding a minimum cost covering of the input cones with K^*-feasible cones. In Chapter 5, we explain in detail how to extend the conventional structural mapping algorithm described in Section 2.3, so that it maps to TLUTs. The result is an algorithm called TMAP.

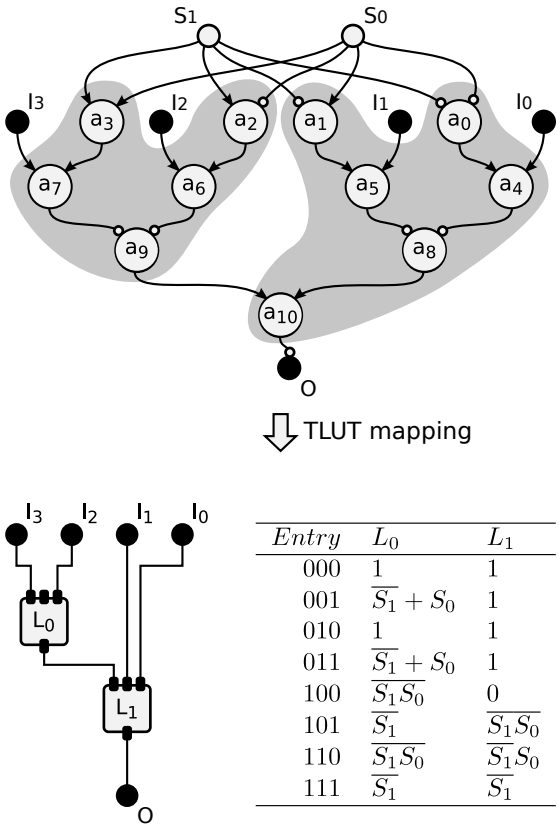
Figure 4.6 illustrates the TLUT mapping process for the multiplexer example. On the top, the parameterized circuit is shown together with the two selected K^*-feasible cones. On the bottom, the reader can see the resulting TLUT circuit. It shows the LUT structure and the truth tables of the LUTs.

4.4 Placement and Routing

A placement tool associates every abstract LUT in the input LUT circuit to a physical LUT on the FPGA while minimizing the routing cost. The routing tool calculates which of the switches in the configurable interconnect should be closed and which of them should be opened in order to realize the connections represented by the nets in the input LUT circuit. Since placement and routing do not depend on the truth tables of the LUTs but only on the topology of the input circuit, the conventional tools can be adopted without change to place and route TLUT circuits (where the parameter inputs are not taken as inputs to the placement and routing tool).

4.5 Parameterized Configuration

The final result of the TLUT method is a prarameterizable configuration. Since only the truth tables of the LUTs will be expressed as a function of the parameters, the majority of the configuration bits will have a static Boolean value. This property can be used to reduce the reconfiguration time. Therefore, the parameterized configuration is split up into a *template configuration* and a *Partial Parameterized Configuration* (PPC). The template configuration contains all static bits and is used to configure the FPGA once when the system is started. Just like the PC, the PPC is a multivalued Boolean function. The PPC will be used by the reconfiguration procedure to generate a new partial



Entry	L_0	L_1
000	1	1
001	$\overline{S_1} + S_0$	1
010	1	1
011	$\overline{S_1} + S_0$	1
100	$\overline{S_1 S_0}$	0
101	$\overline{S_1}$	$\overline{S_1 S_0}$
110	$\overline{S_1 S_0}$	$\overline{S_1} S_0$
111	$\overline{S_1}$	$\overline{S_1}$

Figure 4.6: Illustration of the TLUT mapping process. (above) The parameterized gate-level circuit and the two selected K^*-feasible cones. (below) The resulting TLUT circuit together with the truth tables of the LUTs.

configuration for the FPGA. In this way only a fraction of the configuration bits need to be rewritten upon a parameter change, thus reducing the reconfiguration time.

Chapter 5

Tunable LUT Mapping

In this chapter I propose TMAP, a structural mapping algorithm that is capable of mapping the combinational part of a Boolean circuit, in which distinction is made between regular inputs (RI) and parameter inputs (PI), to Tunable LUTs or TLUTs. Again, we use the 4-to-1 multiplexer (Figure 5.1) as an example, with the select inputs of the multiplexer (S_0 and S_1) as parameter inputs. In the figure, regular inputs are black circles while parameter inputs are grey.

The problem is formulated is Section 5.1. In the next section (Section 5.2), I describe an obvious, but naive, implementation, which is optimized in Section 5.3. In Section 5.3.3, I will show that the final optimized algorithm has the same computational complexity as a conventional LUT mapper.

5.1 Problem Definition

As discussed in Section 4.3, a TLUT is a LUT of which the truth table is expressed as a Boolean function of the parameter inputs. It is easy to see that such a TLUT can implement any Boolean function with up to K regular signals and any number of parameter signals as its arguments (since the parameter signals are only defining the truth table entries). In the context of structural mapping this means that a TLUT can implement the functionality represented by a cone with any number of parameter input nodes and up to K non-parameter input nodes as its cut. I call such cones K^**-feasible*. The TLUT mapping problem thus reduces to finding a minimum cost covering of the input AIG with K^*-feasible cones. As in Section 2.3 the mapping

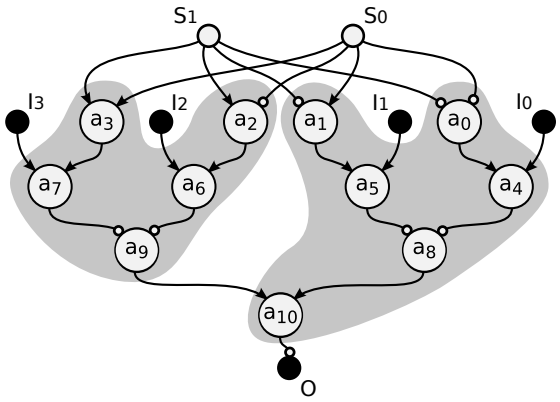


Figure 5.1: AIG of a 4-to-1 multiplexer. Large circles represent internal nodes. Smaller circles represent combinational inputs and output. Inverted edges have a circular arrowhead while non-inverted edges have a standard arrowhead.

criterion will be to minimize the depth of the LUT structure that will be implemented on the FPGA.

5.2 The Naive Implementation

The structural mapping algorithm described in Section 2.3 can easily be transformed to meet the requirements described in the previous section. The classical structural mapping algorithm consist of the following steps:

Cone enumeration Calculates all K-feasible cones of the input circuit.

Cone ranking Selects the lowest-cost cone for each node.

Cone selection Selects a subset of the best cones to form the final covering.

Mapping solution generation Generates the LUT structure and the truth tables.

For our new mapping algorithm, only the cone enumeration step needs to be adapted so that all K^*-feasible cones are enumerated instead of all K-feasible cones. This can easily be done by substituting

Equation (2.2) by Equation (5.2). The new cone enumeration process is formally represented as

$$\Phi(n) = \begin{cases} \{\{n\}\} & \text{if } n \in \text{CI} \\ \{\{n\}\} \cup M(n) & \text{otherwise} \end{cases} \tag{5.1}$$

$$M(n) = \{c_l \cup c_r | c_l \in \Phi(n_l), c_r \in \Phi(n_r), |(c_l \cup c_r) \setminus PI| \leq K\}. \tag{5.2}$$

The only difference is that instead of retaining the K-feasible cones now the K^*-feasible cones are retained after merging the input cone sets of the predecessor nodes of n. This naive implementation was first described in [14].

The intermediate results of applying the new mapping process on the multiplexer example are shown in Table 5.1. Again, I assume that the LUTs in the target FPGA fabric are three-input LUTs. The second column shows the input node sets of the 3^*-feasible cones for each node. The reader should notice that, compared to Table 2.1, Table 5.1 contains more (and larger) cones in the set of enumerated cones. This will allow the mapper to select a solution with fewer cones in the subsequent steps of the mapping process.

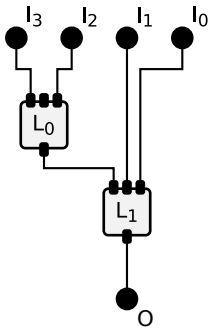
The cone ranking step and cone selection step can be adopted without change from Section 2.3.3. The equations for the calculation of the depth and the area flow of a cone are repeated in Equations (5.3) and (5.4), respectively. The results of the cone ranking step are shown in Table 5.1. The best cone, its depth and its area flow are shown in columns three, four and five, respectively. The cones $(a_{10}, \{a_9 I_0 I_1 S_0 S_1\})$ and $(a_9, \{I_2 I_3 S_0 S_1\})$ are selected as the covering by the Cone Selection step. The LUT structure for the multiplexer example can be found in Figure 5.2 together with the truth tables for each of the LUTs, expressed as a Boolean function of the parameter inputs. The way these Boolean functions are derived is explained in detail in Section 5.5. Note that the result of Figure 5.2 contains only two LUTs, which is a significantly less than the six LUTs needed in the generic implementation of Figure 2.8.

$$depth(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ 1 + \underset{u \in inode(C_n)}{max}(depth(bc(u))) & \text{otherwise} \end{cases} \tag{5.3}$$

$$af(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ 1 + \sum\limits_{u \in inode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{otherwise} \end{cases} \tag{5.4}$$

Table 5.1: Intermediate results of mapping the AIG represented in Figure 2.7 to an FPGA architecture containing 3-input LUTs with a naive implementation of a TLUT mapper.

Node n	Cut set $\Phi(n)$	Best cut $bc(n)$	Depth $depth(bc(n))$	Area flow $af(bc(n))$
a_0	$\{\{a_0\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_1	$\{\{a_1\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_2	$\{\{a_2\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_3	$\{\{a_3\},\{S_0S_1\}\}$	$\{S_0S_1\}$	1	1
a_4	$\{\{a_4\},\{a_0I_0\},\{I_0S_0S_1\}\}$	$\{I_0S_0S_1\}$	1	1
a_5	$\{\{a_5\},\{a_1I_1\},\{I_1S_0S_1\}\}$	$\{I_1S_0S_1\}$	1	1
a_6	$\{\{a_6\},\{a_2I_2\},\{I_2S_0S_1\}\}$	$\{I_2S_0S_1\}$	1	1
a_7	$\{\{a_7\},\{a_3I_3\},\{I_3S_0S_1\}\}$	$\{I_3S_0S_1\}$	1	1
a_8	$\{\{a_8\},\{a_4a_5\},\{a_0a_5I_0\},\{a_1a_4I_1\},$ $\{a_4I_1S_0S_1\},\{a_5I_0S_0S_1\},\{a_0I_0I_1S_0S_1\},$ $\{a_1I_0I_1S_0S_1\},\{I_0I_1S_0S_1\}\}$	$\{I_0I_1S_0S_1\}$	1	1
a_9	$\{\{a_9\},\{a_6a_7\},\{a_2a_7I_2\},\{a_3a_6I_3\},$ $\{a_6I_3S_0S_1\},\{a_7I_2S_0S_1\},\{a_2I_2I_3S_0S_1\},$ $\{a_3I_2I_3S_0S_1\},\{I_2I_3S_0S_1\}\}$	$\{I_2I_3S_0S_1\}$	1	1
a_{10}	$\{\{a_{10}\},\{a_8a_9\},\{a_4a_5a_9\},\{a_6a_7a_8\},$ $\{a_4a_9I_1S_0S_1\},\{a_6a_8I_3S_0S_1\},$ $\{a_5a_9I_0S_0S_1\},\{a_7a_8I_2S_0S_1\},$ $\{a_9I_0I_1S_0S_1\},\{a_8I_2I_3S_0S_1\}\}$	$\{a_9I_0I_1S_0S_1\}$	2	2



$Entry$	L_0	L_1
000	1	1
001	$\overline{S_1} + S_0$	1
010	1	1
011	$\overline{S_1} + S_0$	1
100	$\overline{S_1\ S_0}$	0
101	$\overline{S_1}$	$\overline{S_1}\ \overline{S_0}$
110	$\overline{S_1\ S_0}$	$\overline{S_1}\ S_0$
111	$\overline{S_1}$	$\overline{S_1}$

Figure 5.2: The final result of mapping the AIG represented in Figure 4 with a TLUT mapper (K = 3). The LUT structure (left) and associated truth tables (right).

To summarize, a naive TLUT mapper can easily be derived from the conventional LUT mapping algorithm by slightly changing the cone enumeration process. The main problem with the naive algorithm is its poor scalability. This problem will be analyzed and solved in the following section.

5.3 Optimized Algorithm

The main problem with the naive implementation of Section 5.2 is that it considers many uninteresting or redundant cones, which can lead to very long run times. In Section 5.3.1 I identify these redundant cones, called *Incomplete Cones*, and describe how to optimize the algorithm so that it avoids enumerating these cones. An interesting side effect of not considering incomplete cones is that all remaining cones, called the *Complete Cones*, can be uniquely specified by a subset of their input node set. I call this subset the *Reduced Cut* of the cone (Section 5.3.2). As will be explained, using reduced cuts decreases the complexity of operations on cones and reduces the memory needed to store a cone. In Section 5.3.3, I show that the final optimized algorithm has the same computational complexity as a conventional LUT mapper.

In the description which follows, I distinguish two types of internal nodes of an AIG: regular nodes and parameter nodes. A *Parameter Node* is an internal node of the AIG of which the fan-in cone is driven only by parameter inputs. The set of parameter nodes of the input AIG is denoted as PN. In our example, a_0, a_1, a_2 and a_3 are parameter nodes. All other internal nodes are regular nodes. They are denoted as RN.

5.3.1 Incomplete Cones

As parameter nodes are only dependent on parameter inputs, their cones should never be implemented by a LUT since they can always be combined with subsequent cones to form a larger cone. Therefore, cones that have such parameter nodes in their input set are called incomplete cones.

An *Incomplete Cone* C_n has one or more parameter nodes in its cut. In our example, the cone $C_{a_8} = (a_8, \{a_0 a_5 I_0\})$ ($depth(C_{a_8}) = 2$; $af(C_{a_8}) = 3$) is an incomplete cone. The cut of this cone contains the parameter node a_0. Incomplete cones are redundant because there

is always another K^*-feasible cone C'_n with a better rank. The cone C'_n can be constructed by simply adding the parameter nodes and all their predecessor nodes to cone C_n. If this is done for cone C_{a_8} cone $C'_{a_8} = (a_8, \{a_5 I_0 S_0 S_1\})$ ($depth(C'_{a_8}) = 2$; $af(C'_{a_8}) = 2$) is found by adding node a_0 to C_{a_8}.

More formally, if $p \in inode(C_n)$ is a parameter node that depends on parameter inputs P, the cone C'_n with $inode(C'_n) = (inode(C_n) \setminus p) \cup P$ has a higher rank than C'_n. The depth (Equation (2.3)) of C'_n is less than or equal to the depth of C_n because the depth of the parameter inputs, P, is zero and the maximum over a subset, $inode(C_n) \setminus p$, is always less than or equal to the maximum over the complete set. The area flow (Equation (2.4)) of C'_n is less than the area flow of C_n because $af(C'_n) = af(C_n) - af(bc(p))$ and since p is an internal node $af(bc(p)) \geq 1$.

The enumeration of incomplete cones can be avoided by not adding the trivial cut $\{p\}$ for parameter nodes p during the cone enumeration process, therefore p can never become part of an enumerated cut. The naive algorithm described in Section 5.2 enumerates 47 cones for the multiplexer example. By not adding the trivial cut for the parameter nodes a_0, a_1, a_2 and a_3, the enumeration of 16 incomplete cones is avoided. It is easy to think of real world circuits where the number of incomplete cones by far exceeds the number of complete cones.

5.3.2 Reduced Cuts

An interesting thing to note is that every complete cone C_n is uniquely specified by the ordered pair $(n, rinode(C_n))$, where n is the root of the cone and $rinode(C_n)$ is the set of regular (non-parameter) input nodes of the cone. This is true since the full cut C_n can easily be found by backtracking starting from the root node n and stopping at nodes that are in the set $rinode(C_n) \cup PI$. I call $rinode(C_n)$ the *Reduced Cut* of cone C_n. Because a complete cone is fully specified by its root and its reduced cut, it is not necessary to store the full cut of a K^*-feasible cone. Representing a cone using its reduced cut therefore makes the algorithm more memory efficient.

During cone enumeration the cut of a cone C_n rooted at a node n, with predecessors n_l and n_r, is calculated by taking the union of the cut of a cone C_{n_l} rooted at n_l and the cut of a cone C_{n_r} rooted at n_r (Equation (5.2)). One can easily see that the reduced cut of cone

C_n can be calculated by taking the union of the reduced cuts of C_{n_l} and C_{n_r}. Therefore the cone enumeration process can be executed without ever needing to use costly backtracking in order to calculate the full cut of a cone. Taking the union of two reduced cuts is less expensive than taking the union of the full cuts since the size of a reduced cut is never larger than the size of the corresponding full cut. Also note that the size of the reduced cut is limited by K while full cuts are limited by $K + |PI|$, which depends on the input circuit.

Since the fan-in cone of a parameter node only contains other parameter nodes, there exists only one complete cone for each parameter node. The reduced cut of each of these cones is the empty set. This means that the reduced cut set of a parameter node is known beforehand and thus it is not necessary to visit the parameter nodes during cone enumeration. Since the fan-in cone of a non-parameter node is driven by at least one regular input, non-parameter nodes can be visited in topological order by first labeling the parameter nodes (PN) as visited, visiting the regular inputs (RI) and then repeatedly visiting internal nodes of which both predecessors are already visited.

Given the above information, a new dynamic programming algorithm that enumerates all complete cones of the input AIG can be constructed. The algorithm traverses all non-parameter nodes of the input circuit in topological order and generates the reduced cut sets $\Phi^r(n)$ of the complete K^*-feasible cones $\zeta^*(n)$ by combining the reduced cut sets of n's predecessors n_l and n_r. The new cone enumeration process is represented by Equations (5.5) and (5.6).

$$\Phi^r(n) = \begin{cases} \{\{\}\} & \text{if } n \in \text{PN} \\ \{\{n\}\} & \text{if } n \in \text{RI} \\ \{\{n\}\} \cup M^r(n) & \text{otherwise} \end{cases} \tag{5.5}$$

$$M^r(n) = \{c_l \cup c_r | c_l \in \Phi^r(n_l), c_r \in \Phi^r(n_r), |c_l \cup c_r| \leq K\} \tag{5.6}$$

The result of applying this cone enumeration step on the example of Figure 5.1 can be found in column 2 of Table 5.2.

In Section 5.2, Equations (5.3) and (5.4) are used to calculate the rank of a cone. The problem with these equations is that they need the full cut of a cone in order to calculate its depth and area flow. This requires the cone ranking step to use backtracking to generate the full cut for every enumerated cone, which would have a negative impact on the run-time of TMAP. However, a closer look shows that

Table 5.2: Intermediate results of mapping the AIG represented in Figure 5.1 to an FPGA architecture containing 3-input LUTs with a optimized implementation of a TLUT mapper that makes use of reduced cuts.

Node n	Reduced cut set $\Phi^r(n)$	Best cut $bc(n)$	Depth $depth(bc(n))$	Area flow $af(bc(n))$
$a_0 \dots a_3$	$\{\{\}\}$			
a_4	$\{\{a_4\},\{I_0\}\}$	$\{I_0\}$	1	1
a_5	$\{\{a_5\},\{I_1\}\}$	$\{I_1\}$	1	1
a_6	$\{\{a_6\},\{I_2\}\}$	$\{I_2\}$	1	1
a_7	$\{\{a_7\},\{I_3\}\}$	$\{I_3\}$	1	1
a_8	$\{\{a_8\},\{a_4a_5\},\{a_4I_1\},\{a_5I_0\},\{I_0I_1\}\}$	$\{I_0I_1\}$	1	1
a_9	$\{\{a_9\},\{a_6a_7\},\{a_6I_3\},\{a_7I_2\},\{I_2I_3\}\}$	$\{I_2I_3\}$	1	1
a_{10}	$\{\{a_{10}\},\{a_8a_9\},\{a_4a_5a_9\},\{a_6a_7a_8\},$ $\{a_4a_9I_1\},\{a_6a_8I_3\},\{a_5a_9I_0\},$ $\{a_7a_8I_2\},\{a_9I_0I_1\},\{a_8I_2I_3\}\}$	$\{a_9I_0I_1\}$	2	2

there is a much better solution. The backtracking process will only find parameter inputs in addition to the nodes in the reduced cut. As can be seen from Equations (5.3) and (2.4), the depth and the area flow of a parameter input are both zero. Since the maximum and the sum over a set of positive numbers does not change by adding zeros to that set, the depth and the area flow of a cone are not affected by its parameter inputs. Equations (5.3) and (5.4) can thus be substituted by Equations (5.7) and (5.8), respectively. From these equations it is clear that the reduced cuts contain sufficient information to determine their rank, making backtracking unnecessary. The best cone, its depth and its area flow for each of the nodes in the example circuit can be found in columns 3, 4 and 5 of Table 5.2, respectively.

$$depth(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \cup \text{PI} \\ 1 + \underset{u \in rinode(C_n)}{max}(depth(bc(u))) & \text{otherwise} \end{cases} \tag{5.7}$$

$$af(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \cup \text{PI} \\ 1 + \sum\limits_{u \in rinode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{otherwise} \end{cases} \tag{5.8}$$

It's easy to see that the cone selection step can also be completed without the need of the full cuts of the enumerated cones. The need

for backtracking to generate the full cut of a cone is therefore postponed until after the final cover is selected. The number of selected cones is only a fraction of the total number of enumerated cones. Therefore, the impact of backtracking on the total run-time is limited.

The result of the cone selection step for our example contains the following reduced cuts: $(a_{10}, \{a_9 I_0 I_1\})$ and $(a_9, \{I_2 I_3\})$. After backtracking this results in cones $(a_{10}, \{a_9 I_0 I_1 S_0 S_1\})$ and $(a_9, \{I_2 I_3 S_0 S_1\})$. This result is exactly the same as for the naive implementation. The TLUT circuit thus is the same as in Figure 5.2.

5.3.3 Scalability

Just like in conventional mappers [52], most of the execution time of TMAP is spent in the cut enumeration step. It is easy to see that this time is proportional to the total number of enumerated cones. The enumerated cones of a node n are not just the K-feasible cones, but all cones that are considered as K-feasible cone for node n during the cone enumeration process. The enumerated cones of a node n are denoted $\mathscr{E}(n)$ for the conventional mapper and $\mathscr{E}^r(n)$ for the reduced cut mapper. Their formal definitions are given in Equation (5.9) and Equation (5.10), respectively.

$$\mathscr{E}(n) = \begin{cases} \{\{n\}\} & \text{if } n \in \text{CI} \\ \{\{n\}\} \cup \{c_l \cup c_r | c_l \in \Phi(n_l), c_r \in \Phi(n_r)\} & \text{otherwise} \end{cases} \tag{5.9}$$

$$\mathscr{E}^r(n) = \begin{cases} \{\{\}\} & \text{if } n \in \text{PN} \\ \{\{n\}\} & \text{if } n \in \text{RI} \\ \{\{n\}\} \cup \{c_l \cup c_r | c_l \in \Phi^r(n_l), c_r \in \Phi^r(n_r)\} & \text{otherwise} \end{cases} \tag{5.10}$$

Since the number of enumerated cones is highly dependent on the topology of the input circuit, the complexity analysis is very difficult. However, I will show that for the worst case circuits, the number of cones enumerated by TMAP is of the same order as the number of cones enumerated by a conventional mapper for a reduced version of that same input circuit. The computational complexity of TMAP is therefore of the same order as the computational complexity of a conventional technology mapper.

The reduced circuit is derived from the original circuit by simply removing the parameter nodes. The reduced circuit of our multiplexer example input circuit is depicted in Figure 5.3. In Theorem 5.3.1 I show that when the conventional cone enumeration al-

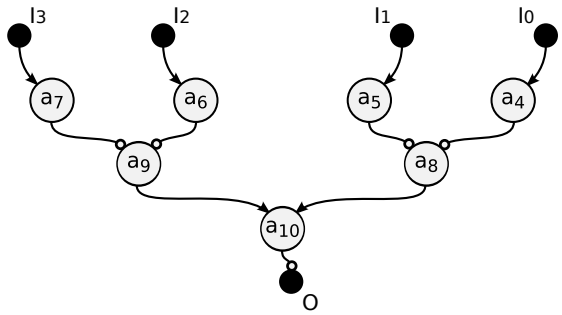


Figure 5.3: The reduced circuit of the 4-to-1 multiplexer AIG represented in Figure 5.1.

gorithm described in Section 2.3 is applied to the reduced input circuit, the K-feasible cuts found for a node n^* of the reduced circuit are exactly the same as the reduced K^*-feasible cuts found for the corresponding node n when the optimized cone enumeration algorithm, described in Section 5.3.2, is applied to the original input circuit. Given Theorem 5.3.1, Theorem 5.3.2 can easily be proved. Theorem 5.3.2 which is the same property but for the enumerated cones instead of the K-feasible cones.

Theorem 5.3.1. *If $A^* = (N^*, E^*)$ is the DAG that represents the reduced circuit, $A = (N, E)$ is the DAG of the original circuit and node $n^* \in N^*$ corresponds to node $n \in N$, then*

$$\Phi(n^*) = (\Phi^r(n))^* \quad \forall n^* \in N^*, \tag{5.11}$$

where $(\cdot)^*$ replaces the nodes n in the cut set by there corresponding node n^*.

Proof. By induction on n^*.

The *base case*, $n^* \in CI^*$, where CI^* is the set of all combinational inputs of N^* can be proven as follows.

$$\Phi(n^*) = \{\{n^*\}\} = (\{\{n\}\})^* = (\Phi^r(n))^* \tag{5.12}$$

The first step directly follows from Equation (5.1) for combinational inputs. The second step is valid because of the definition of operator $(\cdot)^*$. Since every combinational input in A^* corresponds to a regular input in A, the last step follows directly from Equation (5.5).

For the *inductive step* there are two cases: for nodes n^* with one predecessor (n_r^*) and nodes n^* with two predecessors (n_l^* and n_r^*). In

both cases, Equation (5.11) is assumed to hold for the predecessor(s) of n^*. For the proof of the single predecessor case, I start from the right hand side of Equation (5.11).

$$\begin{aligned}(\Phi^r(n))^* &= (\{\{n\}\} \cup \{c_p \cup c_r | c_p \in \Phi^r(n_p), c_r \in \Phi^r(n_r), |c_p \cup c_r| \leq K\})^* \\ &= (\{\{n\}\} \cup \{c_p \cup c_r | c_p = \{\}, c_r \in \Phi^r(n_r), |\{\} \cup c_r| \leq K\})^* \\ &= (\{\{n\}\} \cup \Phi^r(n_r))^* \\ &= \{\{n^*\}\} \cup (\Phi^r(n_r))^* \\ &= \{\{n^*\}\} \cup \Phi(n_r^*) = \Phi(n^*)\end{aligned}$$

The first step follows from Equations (5.5) and (5.6) and from the fact that in the case of a single predecessor for node n^*, its corresponding node n has two predecessors: a parameter node n_p and a regular node n_r. The second step uses Equation (5.5) to substitute the cut set of the parameter node n_p with $\{\{\}\}$. The third step is justified because the union of a set c_r and the empty set is equal to the set c_r itself and that a cardinality of a reduced K^*-feasible cut is at most K by definition. The fourth step makes use of the definition of the $(\cdot)^*$ operator, the fifth step uses the induction hypotheses and the sixth step uses Equations (5.1) and (5.2) for a single predecessor node.

For the proof of the two predecessors case I start from the left-hand side of Equation (5.11).

$$\begin{aligned}\Phi(n^*) &= \{\{n^*\}\} \cup \{c_l \cup c_r | c_l \in \Phi(n_l^*), c_r \in \Phi(n_r^*), |c_l \cup c_r| \leq K\} \\ &= \{\{n^*\}\} \cup \{c_l \cup c_r | c_l \in (\Phi^r(n_l))^*, c_r \in (\Phi^r(n_r))^*, |c_l \cup c_r| \leq K\} \\ &= (\{\{n\}\} \cup \{c_l \cup c_r | c_l \in \Phi^r(n_l), c_r \in \Phi^r(n_r), |c_l \cup c_r| \leq K\})^* \\ &= (\Phi^r(n))^*.\end{aligned}$$

The first step directly follows from Equations (2.1) and (5.2). In the second step the induction hypothesis is used. The third step makes use of the definition of operator $(\cdot)^*$. In the final step, Equations (5.5) and (5.6) are used. □

Theorem 5.3.2. *If $A^* = (N^*, E^*)$ is the DAG that represents the reduced circuit, $A = (N, E)$ is the DAG of the original circuit and node $n^* \in N^*$ corresponds to node $n \in N$, then*

$$\mathscr{E}(n^*) = (\mathscr{E}^r(n))^* \quad \forall n^* \in N^*, \tag{5.13}$$

where $(\cdot)^*$ replaces the nodes n in the cut set by their corresponding node n^*.

Proof. The proof follows directly from the definitions of $\mathscr{E}(n)$ (Equation (5.9)), $\mathscr{E}^r(n)$ (Equation (5.10)) and Theorem 5.3.1. □

Using Theorem 5.3.2, it can be shown that the number of cones enumerated when the reduced cut TMAP algorithm (Section 5.3.2) is applied to the original circuit is equal to the sum of the number of cones enumerated when the conventional mapping algorithm is applied to the reduced circuit and the number of parameter nodes $|PN|$.

$$\sum_{\forall n \in N} |\mathscr{E}^r(n)| = \sum_{\forall n \in (N \setminus PN)} |\mathscr{E}^r(n)| + \sum_{\forall n \in (PN \subset N)} |\mathscr{E}^r(n)| \tag{5.14}$$

$$= \sum_{\forall n^* \in N^*} |\mathscr{E}(n^*)| + |PN| \tag{5.15}$$

Which of the two terms in Equation (5.15) gets the upper hand depends on the type of circuit. In the extreme case when all nodes of the input circuit are parameter nodes, the second term of Equation (5.15) is equal to $|N|$ and the first term equals zero. This means that in the worst case, the second term is linear in the size of the input circuit. In the other extreme when none of the nodes of the input circuit are parameter nodes, the TLUT mapping problem reduces to a conventional mapping problem and thus the second term becomes zero. It is known that the number of cones enumerated by a conventional mapper will grow faster than linear for a large group of circuits. For the worst case circuits the second term can thus be ignored when the circuit size grows. Formally:

$$O(\sum_{\forall n \in N} |\mathscr{E}^r(n)|) = O(\sum_{\forall n^* \in N^*} |\mathscr{E}(n^*)|). \tag{5.16}$$

The computational complexity of TMAP is therefore of the same order as the computational complexity of a conventional technology mapper.

Theorem 5.3.1 also implies that TLUT mapping can be done using a conventional mapper with a front end to produce the reduced circuit. This method was used in our first TLUT mapper [13]. The reduced circuit can be derived from the original input circuit in linear time. The front end simply visits all nodes of the original circuit in topological order and removes the parameter nodes during this process. Since the front end runs in linear time, the mapper as described in [13] scales equally well as the mapper described in this

thesis. However, the mapper described in this section has the advantage that the cost function can be easily optimized for TLUT mapping. In [13] I focused on the minimization of the depth and the area of the resulting LUT structure. However, in TLUT mapping it might be interesting to also incorporate the reconfiguration effort in the cost function. During the mapping process, the number of configuration bits that need to be reconfigured or the complexity of the reconfiguration procedure, could for example be minimized (see Section 5.4 for more information). This is not possible with the approach described in [13].

5.3.4 Experimental Example: Multipliers

In this section, I use TMAP to map multipliers for which one of the inputs is a parameter. The result of the mapping is a hardware constant multiplier of which the constant can be changed by means of reconfiguration. This type of multipliers has been hand designed by several authors [20, 73]. The main advantage of our technique is of course that TMAP can generate such a constant multiplier automatically from an architecture independent HDL description of a multiplier.[1] In the experiment multipliers of different sizes are mapped, starting from a 4×4 multiplier up to a 64×64 multiplier.

The experiment in this section makes use of three different technology mappers: a conventional mapper as described in Section 2.3.3, the optimized implementation of TMAP as described in Section 5.3.2 and the naive implementation of TMAP as described in Section 5.2. All these algorithms are implemented in Java running on the Java HotSpotTM 64-Bit Server VM. The computer used in the experiments uses an Intel Core 2 processor running at 2.13 GHz and has 2 GiB of memory.

The conventional mapping algorithm takes a .aig file [8] as input and produces a .blif file [70] that represents the mapped circuit. Both TMAP implementations take a .aig file and a list of the parameter inputs as input and output a LUT structure and a .aig file that represents the PPC (Section 5.5). The LUT structure can be either represented as a VPR .net file [6] or a VHDL file that directly instantiates Virtex-II Pro LUTs [91]. The .net file can be further placed and routed using VPR. The VHDL file can be used as input to the ISE tool flow. This is explained in detail in [11].

[1] The multiplier in our experiment is implemented as a tree of adders.

Table 5.3: Comparison of the mapping result ($K = 4$) of conventional mapping and TMAP of a set of different-sized multipliers where one of the inputs is a parameter.

	Conventional		TMAP		
size	LUTs	Depth	LUTs	TLUTs	Depth
4×4	33	6	8	8	1
8×8	170	14	52	38	10
16×16	876	28	271	152	23
32×32	3650	55	1104	501	50
64×64	15142	111	4614	1972	105

Conventional Mapping vs. TMAP

In a first experiment, I compare the mapping result of a conventional mapper with the mapping result of TMAP. The mapping results ($K = 4$) are shown in Table 5.3. The first column shows the size of the multipliers. The second and the third column show the mapping results for the conventional LUT mapper. The fourth, fifth and sixth column show the mapping results for TMAP.

The number of LUTs needed by the conventional mapper found in column two are compared with the total number of LUTs needed by TMAP found in column four (includes both static LUTs and TLUTs). As can be seen, using TMAP results in a decrease of at least 69% in the number of LUTs. This gain in FPGA resources is due to the fact that part of the functionality of the multiplier has moved from expensive FPGA resources to the reconfiguration procedure which is executed in software.

I also compare the depth, i.e. the number of LUTs in the longest path, of the circuits. The depth of a circuit is a measure for the maximum delay in a circuit, or in other words the maximum operating frequency of the circuit. The depth decreases with at least 4 LUTs. This decrease is independent of the size of the multiplier and thus is more prominent for multipliers with a small data width.

Finally, column five shows that only part of all LUTs in the design (column four) are TLUTs. Many LUTs do not depend on the parameter inputs and thus have a static truth table. For FPGAs that are partially reconfigurable this will decrease the reconfiguration time.

Scalability

In a second experiment, I compare the run times of the naive implementation of TMAP described in Section 5.2 and the optimized TMAP algorithm described in Section 5.3.2. The results are shown in Table 5.4. The table shows the run times and the number of enumerated cones for both algorithms. The naive algorithm for the 128×128 multiplier ran out of memory after more than 4 days of run time. The missing data is indicated in the table with dashes.

The speedup (column 7) of the optimized algorithm compared to the naive algorithm clearly increases as the input multiplier grows larger. The complexity of the optimized algorithm is thus of a lower order than the complexity of the naive algorithm. There are two reasons for this. First, the number of cones enumerated by the optimized algorithm is of a lower order than the number of cones enumerated by the naive algorithm. This can be seen in column 4, where the ratio of the enumerated cones of both algorithms is given. The number of enumerated cones is lower for the optimized algorithm because it avoids enumerating incomplete cones (Section 5.3.1). Second, since the size of a reduced cut (Section 5.3.2) is limited by K, the complexity of operations on reduced cuts is independent of the input circuit. On the other hand, the complexity of operations on full cuts is limited by $K + |PI|$, where PI is the set of parameter inputs. Thus, the complexity of operations on full cuts grows with the number of parameter inputs. This effect can be observed in column 8 and column 9, which shows the average time needed to enumerate one cone for the naive algorithm and the optimized algorithm, respectively. Column 9 shows that for the optimized algorithm, the time needed to enumerate one cone is independent of the size of the input circuit for large multiplier sizes. For small multipliers, the overhead of starting the JVM distorts this trend. In other words, this means that the execution time of the optimized algorithm is, at least for multipliers, proportional to the number of enumerated cones, as was claimed in Section 5.3.3. Column 8 on the other hand, show that this is not the case for the naive algorithm. There, the time needed to enumerate one full cone grows as the multiplier size increases.

Table 5.4: Comparison of the execution time of the naive and the optimized implementation of TMAP ($K = 4$) for a set of different-sized multipliers where one of the inputs is a parameter.

	Enumerated Cones			Execution time [s]			Time per cone [μs]	
size	naive	optimized	ratio	naive	optimized	speedup	naive	optimized
4×4	61216	20848	2.94	0.53	0.64	1.21	8.69	30.75
8×8	388521	118966	3.27	1.66	0.93	1.68	4.27	7.85
16×16	3737049	608560	6.14	13.79	2.38	5.47	3.68	3.92
32×32	46070046	2513673	18.33	465.08	6.97	61.45	10.10	2.77
64×64	602449833	10571999	56.99	87705.32	24.96	3514.26	145.58	2.36
128×128	-	44096346	-	-	114.03	-	-	2.59

5.4 Minimizing the Number of TLUTs

Up till now, I focused on the minimization of the depth and the area of the resulting LUT structure. However, in TLUT mapping it might also be interesting to incorporate the reconfiguration effort in the cost function. During the mapping process, it should be possible to minimize the number of configuration bits that need to be reconfigured or the complexity of the reconfiguration procedure.

In order to do this, I need to introduce tunable cones and tunable area flow as a cost function metric. A *Tunable Cone* is a cone that depends on parameter inputs. When a tunable cone is implemented by a LUT, this LUT will need to be reconfigured every time the parameters change. A non-tunable cone can be implement by a LUT with a static truth table and therefore does not require any reconfiguration upon a parameter change. The *Tunable Area Flow*, $taf(C_n)$, of a cone C_n gives an estimate of how many tunable cones are needed to implement all the logic in the fan-in cone of n. The tunable area flow of a cone is calculated as is shown in Equation (5.17). As can be seen from the equation the calculation is very similar to the calculation of the area flow of a cone.

$$taf(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \cup \text{PI} \\ \sum\limits_{u \in rinode(C_n)} \frac{taf(bc(u))}{|oedge(u)|} + 1 & \text{if } tunable(C_n) \\ \sum\limits_{u \in rinode(C_n)} \frac{taf(bc(u))}{|oedge(u)|} & \text{otherwise} \end{cases} \tag{5.17}$$

To calculate the tunable area flow of a cone C_n it is necessary to know whether C_n is tunable. When the full cut of a cone $inode(C_n)$ is

given, it is easy to detect wether the cone is tunable or not by simply checking whether the cut contains parameter inputs or not. However, since in our case only the reduced cuts of the cones $rinode(C_n)$ are known detecting whether a cone is tunable becomes harder. This can be solved by keeping track of the tunability of a cone during the cone enumeration process. This can be done by making use of the following property. When a cone C_n is formed by merging two cones C_l and C_r of its predecessors n_l and n_r, it is tunable if C_l or C_r is tunable. This recursive definition is formally represented in Equation (5.18).

$$tunable(C_n) = \begin{cases} true & \text{if } (n \in \text{PN}) \vee tunable(C_l) \vee tunable(C_r) \\ false & \text{otherwise} \end{cases} \quad (5.18)$$

The tunable area flow can be used as a component of the cost function used to select the best cone during cone ranking. A depth optimal mapping can be achieved by selecting the cone with the lowest depth. If several cones have the same depth the one with the lowest area flow is selected. If there are still equivalent cones the tie can be broken by using the tunable area flow.

5.5 The Partial Parameterized Configuration

As I explained in Chapter 3, a DCS system solves a given problem in two steps. In the first step a configuration specialized for the problem at hand is generated and loaded into the FPGA. In the second step this specialized configuration is executed by the FPGA. The cost of solving the overall problem is the sum of the costs of both steps. Up untill now I have mainly focused on minimizing the cost of the second step by optimizing the area and the length of the longest path of the LUT structure. In this section I discuss the cost of generating a specialized configuration.

In Section 3.4 I explained that the partial parameterized configuration (PPC) has to be evaluated in order to generate a partial configuration that may be used to specialize the current FPGA configuration. This PPC is a multivalued Boolean function that maps the parameter inputs to the truth table entries of the LUTs. I assume that the cost of generating a specialized configuration is proportional to the number of Boolean operations that need to be performed in order to evaluate the PPC.

5.5.1 Representing the PPC

In [13, 14], I treated every truth table entry of the TLUTs as an independent single-output Boolean function (see Figure 5.2). However, representing a multivalued Boolean function as a set of independent single-output Boolean functions prevents the use of combined logic optimization, which could further minimize the total number of Boolean operations needed to generate a specialized configuration. Therefore, I will represent the PPC as a Boolean network, more specifically an AIG. This allows us to use well-established Boolean minimization techniques, like those implemented in ABC [5], to minimize the number of Boolean operations needed to evaluate the PPC.

5.5.2 Constructing the PPC

A Boolean network that generates one entry of the truth table of a TLUT can be constructed by copying the cone of the TLUT and connecting the regular inputs of that cone to constants associated to the entry. In that case, the root of the cone produces the truth table entry as a function of the parameter inputs and thus needs to be connected to an output of the network. If this is repeated for every entry of every cone in the covering, the result is a Boolean network with as inputs the parameter inputs and as outputs the entries of the truth tables of the TLUTs. The pseudo code for constructing the PPC is given in Figure 5.4. At each moment during the construction, the PPC is structurally hashed [42]. This means that constants are propagated and there is at most one node with a given pair of input edges. The result of applying this process to the multiplexer example is the PPC shown in Figure 5.5.

5.5.3 Complexity Analysis

As can be derived from the description above, each node in the input AIG is copied 2^K times for each cone in the cover it is part of. Without considering any optimization, the maximum number of nodes in the PPC can thus be expressed as shown in Equation (5.19), where $dup(n)$ is equal to the number cones in the cover it is part of and $\widehat{dup}$ is the average of $dup(n)$ over N, the number of nodes in the input AIG.

$$\sum_{n \in N} 2^K dup(n) = 2^K \widehat{dup} |N| \tag{5.19}$$

```
function constructPPC(ConeSet cover):
  AIG PPC = new AIG()
  for cone in cover:
    for entry in 0 to 2^K−1:
      Cone copy = cone.copy()
      copy.connectRegularInputs(entry.bits())
      copy.add(new Output(copy.root()))
      PPC.add(copy)
  return PPC
```

Figure 5.4: Pseudo code for constructing the PPC as an AIG from the input AIG and the cover found by the mapper.

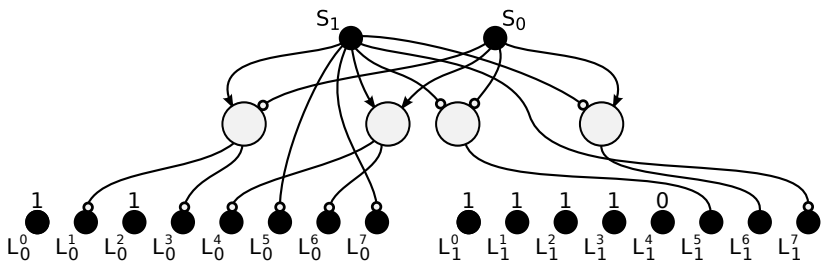


Figure 5.5: The optimized PPC of the 4-to-1 multiplexer example represented as an AIG.

For duplication free mapping, $dup(n)$ equals one. In that case, the equation can be simplified to $2^K|N|$. In other words, in duplication free mapping, the complexity of generating a specialized configuration given a set of parameter values is linear in the number of nodes in the input AIG.

In general, Equation (5.19) cannot be written in a closed form, but there is a closed form upper bound. Note that $dup(n)$ is limited by the number of cones in the cover and that this number is limited by the number of nodes in the subject graph. Therefore the number of nodes in the PPC is $O(|N|^2)$.

This upper bound is very pessimistic because it does not consider any of the many optimization opportunities (constant propagation, structural hashing,...) for the PPC. To support this claim, I measured the number of operations in the PPC for the multiplier example in Section 5.3.4. The results can be found in Table 5.5. Column 4 shows the average duplication of a node. As can be seen, duplication of nodes is high for the multipliers, up to 27 for the 64×64 multiplier, and as expected it rises with $|N|$ (column 8), but less than linear. Column 6 predicts the number of nodes in the PPC according to Equation (5.19) while column 7 shows the actual number of AND-gates in the PPC after optimization with the *resyn3* command of ABC [5]. When comparing these numbers, it is clear that Boolean optimization can greatly reduce the number of AND-gates in the PPC. In the last column shows the ratio of AND-gates in the PPC and AND-gates in the input AIG. This number first drops if the multiplier size rises and then stabilizes for large multipliers at about 0.76. This means that the size of the PPC grows linear with $|N|$ for these multipliers and not quadratic as the upper bound found earlier predicted. From Equation (5.19) the constant factor of $2^K\widehat{dup}$ is expected to be larger than 16 because $K = 4$, but the experiment results in only 0.76. This means that for real-life multipliers the complexity of calculating new truth tables is less than the complexity of evaluating one multiplication.

5.6 Experimental Results

In this section, I apply tunable LUT mapping to more complex circuits. In Section 5.6.1 I create adaptive FIR filters that are adapted by means of reconfiguration and in Section 5.6.2 I create TCAMs of which the content is written by means of reconfiguration. Both de-

Table 5.5: Study of the scalability of the PPC for a set of different-sized multipliers where one of the inputs is a parameter.

size	LUTs	TLUTs	$\widehat{dup}$	$\|N\|$	Eq. (5.19)	$\|PPC\|$	$\frac{\|PPC\|}{\|N\|}$
4×4	8	1	2.8	109	4,883	156	1.43
8×8	52	38	4.4	545	38,368	470	0.86
16×16	271	152	8.3	2608	346,343	1918	0.74
32×32	1104	501	14.3	10839	2,479,963	8503	0.78
64×64	4614	1972	27.0	45111	19,487,952	33783	0.75

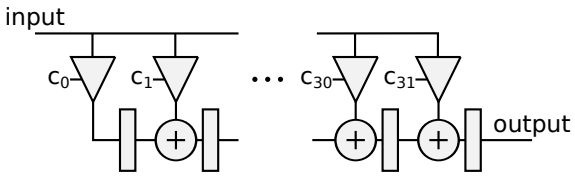


Figure 5.6: 32-tap fully pipelined adaptive FIR filter.

signs are implemented on a Virtex-II Pro XC2VP30 using Xilinx ISE 9.2.

5.6.1 Adaptive FIR Filter

Adaptive FIR filters that are adapted by means of reconfiguration can be created with TMAP by simply choosing the filter coefficients as the parameters of the design. In what follows this sort of adaptive filter is created for different numbers of taps and input widths. Afterwards, these filters are compared with conventional adaptive filters. The filters used are fully pipelined FIR filters as shown in Figure 5.6 for a 32-tap filter.

First, the filters are implemented as generic non-reconfigurable filters using the conventional ISE 9.2 tool flow starting from RTL descriptions of the filters. Synthesis is done using Xilinx XST 9.2 with the default settings except for the multiplier style which is set to LUT. This way the multipliers are implemented using LUTs rather than the hardwired multipliers available in the Virtex-II Pro. This is necessary to allow a fair comparison between the conventional implementation and the TMAP implementation. Technology mapping (Xilinx MAP

Table 5.6: Hardware properties for a set of differend-sized adaptive FIR filters implemented without using dynamic reconfiguration (Conventional) and with using dynamic reconfiguration (TMAP). The numbers between brackets are relative compared to the conventional implementation.

Size		Conventional		TMAP			
Width	Taps	LUTs	f_{max} [MHz]	LUTs		f_{max} [MHz]	
8 bit	32	2641	80.84	1520	(0.58)	123.82	(1.53)
8 bit	64	5298	72.41	3056	(0.58)	89.06	(1.23)
8 bit	96	7954	65.41	4592	(0.58)	87.96	(1.34)
8 bit	128	10611	53.81	6128	(0.61)	74.23	(1.38)

9.2) and Place and Route (Xilinx PAR 9.2) are done using default settings. The number of LUTs and the maximum clock frequency for the filters implemented using ISE can be found in columns 3 and 4 of Table 5.6.

A second implementation of the filters uses TMAP, again starting from RTL descriptions of the filters. This RTL description is first synthesized using Quartus II 7.2. Quartus is set to dump a .blif file after synthesis. This is possible due to the Quartus II University Interface Program (QUIP). Next, the .blif file is converted to an .aig file using ABC [5]. Together with a list of parameter inputs (the filter coeficients in this case) this .aig file is used as input for a Java implementation of TMAP which produces both a PPC represented as a .aig file and a LUT structure. In this experiment, the LUT structure is represented as VHDL file that directly instantiates Virtex-II Pro LUTs and FFs [91]. Finally the LUT structure is implemented on the Virtex-II Pro using the Xilinx ISE 9.2 tool flow (XST 9.2, MAP 9.2 and PAR 9.2) with default settings. For more information on how to integrate TMAP with the ISE tool flow see Section 6.1.

The number of LUTs and the maximum clock frequency for the dynamically reconfigurable filters can be found in columns 5 and 6 of Table 5.6. If the number of LUTs in both implementations are compared a reduction of at least 39% is seen for the TMAP implementations compared to the conventional implementation. Moreover, the TMAP implementation can be clocked at least 23% and up to 53% faster than the conventional implementation.

Of course, the gain in area and speed of the FPGA hardware of our dynamically reconfigurable adaptive filters comes at the cost of a larger adaptation time. While the filter coefficients of the conventional adaptive filter can be changed by simply rewriting the registers that store the coefficients, the TMAP implementation requires us to both generate a specialized configuration for the FPGA by evaluating the PPC and write this configuration in the configuration memory of the FPGA. The total time needed to change the coefficients of the filter is called the specialization overhead, $t_{special}$. It contains both the time needed to evaluate the PPC, t_{eval}, and the time to reconfigure the FPGA, t_{reconf}. In what follows I discuss the specialization overhead in detail.

The evaluation of the PPC is in this case done on an Instruction Set Processor (ISP). In our case, the PowerPC which is hardwired on the Virtex-II Pro FPGA is used. Efficiently evaluating a Boolean network on an ISP is an area of research by itself, and is beyond the scope of this thesis. However, in order to give an estimate of the evaluation time I have implemented a simple compiled evaluation technique. In compiled evaluation, a dedicated function is created that takes the input values of the network as its arguments and returns the output values of the network. In our case, the network is the PPC created by TMAP, the input values are the parameter values (the coefficients of the filter) and the output values are the truth tables for the TLUTs.

Starting from the PPC, an evaluation function is created by first generating a C function and then compiling it for the PowerPC. The C code of the evaluation function is generated by traversing the PPC in topological order from the inputs towards the outputs. For every node a statement is added to the C code. The statement calculates the output value of the node from the output value of its predecessors and stores the output in a local array (`node`). The size of the local array is minimized by freeing its elements when they are no longer needed and by always storing a node output value at the smallest available index. E.g. if the left predecessor is inverted, the smallest available index is 3 and the output values of the left and right predecessors are stored at indices 9 and 6, respectively, the expression would be `node[3] = !node[9] && node[6];`.

Unfortunately, when an evaluation function is generated as described above for the complete flattened FIR filter, this leads to a very large evaluation function and poor evaluation time. However,

Table 5.7: Evaluation of the PPC of different-sized adaptive FIR filters on the hardwired PowerPC of the Virtex-II Pro.

Size		Evaluation Time			Program Size
Width	Taps	$\|PPC\|$	t_{eval} [μs]	$\frac{t_{eval}}{\|PPC\|}$ [$\frac{ns}{AND}$]	S_{eval} [B]
8 bit	32	28672	317	11.06	14150
8 bit	64	57344	634	11.06	14918
8 bit	96	86016	951	11.06	15686
8 bit	128	114688	1268	11.06	16454

many designs, including our FIR filters, contain hierarchy and have a repetitive nature that can be used to build a more compact evaluation function. In our filters, one multiplier is instantiated for every tap. Instead of generating one flat evaluation function for the complete FIR filter, an evaluation function for one multiplier was generated, as described above, and the FIR evaluation function was generated by calling this function for each instantiation of the multiplier. This could be further optimized by calculating up to 32 of the multiplier evaluation functions at a time by using bitwise logic operations and packing 32 Boolean values in each 32-bit word. Although the FIR evaluation function for this experiment was built manually, it could easily be synthesized automatically from the hierarchy found in the HDL design. This is a subject for further research.

In the experiment the evaluation function for each of the FIR filters was created as described above and executed on the PowerPC. The PowerPC was clocked at 300 MHz and both the instruction and data caches were enabled. The evaluation time, t_{eval}, and the size of the compiled evaluation function S_{eval} are shown in Table 5.7. The evaluation time for our adaptive filters takes in the order of several hundreds of μs depending on the size of the filter. The ratio of the evaluation time and the number of AND nodes in the PPC shows that the evaluation time of the filters is proportional to the size of the PPC. The size of the evaluation functions is about 15 kB and slowly grows as the number of taps increases. The program size is almost independent of the size of the PPC because one multiplier evaluation function is created, that is reused to generate the truth tables for each of the multipliers in the design.

After evaluating the PPC, the PowerPC needs to write the cal-

Table 5.8: Reconfiguration of different-sized adaptive FIR filters through the ICAP of the Virtex-II Pro.

size		Reconfiguration Time			
Width	Taps	TLUTs	frames	S_{bit} [B]	t_{reconf} [μs]
8 bit	32	768	52	66573	1009
8 bit	64	1536	88	111357	1687
8 bit	96	2304	92	115493	1750
8 bit	128	3072	91	114669	1737

culated truth tables in the configuration memory of the FPGA. The PowerPC can access the configuration memory of the Virtex-II Pro from within the FPGA fabric through the ICAP (Internal Configuration Access Port) which is connected to the bus of the PowerPC. To reconfigure the FPGA, the PowerPC needs to send a partial bitstream to the ICAP which can be done at a maximum rate of 66 MB/s. The size of the bitstreams, S_{bit}, and the reconfiguration time, t_{reconf}, are shown in column 5 and 6 of Table 5.8. The reconfiguration time ranges from 1 ms to a maximum of 1.75 ms depending on the size of the filter. As can be seen, the reconfiguration time is not linear in the number of TLUTs as one could expect, but linear in the number of frames that need to be reconfigured. This is because the atom of reconfiguration of the Virtex-II Pro is not a LUT truth table but a frame [87]. All the truth tables of a column of CLBs (Configurable Logic Blocks) are stored in only two frames. If only one LUT in a CLB column changes half of the LUTs in that column need to be reconfigured. Because it's frames are smaller, the importance of this overhead is reduced for the Virtex-5 [93].

Finally, the total specialization overhead which is the sum of the evaluation time and the reconfiguration time, is shown in Table 5.9. As can be seen, the specialization time is of the order of a few ms, depending on the size of the filter. The area and clock frequency benefits of our adaptive filters (Table 5.6) are thus useful as long as the time in between coefficient changes is a few orders of magnitude higher than the specialization overhead.

Table 5.9: Specialization overhead of different-sized adaptive FIR filters implemented on the Virtex-II Pro.

size		Specialization Overhead		
Width	Taps	t_{eval} [μs]	t_{reconf} [μs]	$t_{special}$ [μs]
8 bit	32	317	1009	1326
8 bit	64	634	1687	2321
8 bit	96	951	1750	2701
8 bit	128	1268	1737	3005

5.6.2 Ternary Content Addressable Memory

In conventional memories, the read operation returns the data associated with a given address. The read operation of a Content Addressable Memory (CAM) does the opposite: it finds the address associated to a given data value. In both cases, the write operation stores a given data value at a given address. CAMs have many applications [80]. The most important commercial application is packet forwarding in network routers [59].

A TCAM (Ternary CAM) is a special kind of CAM that stores ternary patterns instead of a data value. Each digit in a ternary pattern can either be zero, one or don't care. The digits are represented by two bits: the data bit and the mask bit. A full pattern entry in the TCAM is represented by two bit vectors (the data and the mask) and one bit which indicates whether the entry contains a pattern or not. When new input data is provided to the TCAM, it simultaneously compares this data to all stored patterns. The incoming data matches a pattern if all bits of the incoming data for which the corresponding mask bit of the pattern is zero are equal to the corresponding value bit of the pattern.

The VHDL code of Figure 5.7 shows a behavioral description of the read operation of a TCAM. It takes *datain* as input and outputs the matching address, if any, on *addrout*. The output *match* indicates whether a match is found or not. The READ process in Figure 5.7 describes the hardware that compares the input data to all stored patterns. The code takes the array of pattern entries as input. In a conventional TCAM implementation, the entries will be provided by flip-flops (FFs) arranged as a memory. Each memory element uses a FF in the FPGA because all data needs to be accessed in every clock

cycle. In our reconfigurable implementation these inputs are the parameters of the design and will thus be provided by means of reconfiguration. In important applications such as Internet core routers, this approach is feasible, since the update rate is usually rather limited (at the very most a few hundred updates per second [44]), while the read rate is orders of magnitude higher (up to several millions of packets per second).

The problem with TCAMs is that their implementation requires many FPGA resources, even for small TCAMs. If the code in Figure 5.7 is extended by making `entry` an internal signal and adding a write port and a write process the result is a VHDL description of a full TCAM (256 entries of 32-bit). When this description is synthesized using ISE for a Virtex-II Pro, the implementation requires 16,874 FFs and 10,441 4-input LUTs and can be maximally clocked at 69 MHz. This is true for different sizes of the TCAM (see Table 5.10).

These resource requirements can be drastically reduced with the use of TMAP. In the TMAP design (Figure 5.7), the entry array of the TCAM is chosen as the parameter input of the design, by adding the `--PARAM` annotation. This means that the patterns stored in the TCAM will be changed by means of reconfiguration. When this design is mapped using TMAP it only requires 3,497 LUTs (a reduction by 67%), the maximum clock frequency rises from 69 MHz to 90 MHz (a gain of 30%) and the number of FFs is reduced dramatically from 16,874 to only 226 (see Table 5.10). This reduction in FFs is possible because the pattern information is no longer stored in FFs that are part of the FPGA fabric, but in the memory elements of the configuration memory that stores the truth tables of the LUTs. The only FFs left are the output registers for `addrout` and `match`. The `datain` port was also buffered in order to measure the maximum clock frequency. These last FFs are duplicated several times by ISE for speed optimization.

Because of the importance of TCAMs and the high resource usage of architecture independent HDL implementations, FPGA vendors offer TCAM constructor software which constructs TCAM structures that are highly optimized for a specific architecture [80, 89]. The designer can generate such a structure using the software and then instantiate it in his design. A good example of such a generator is the SRL16 TCAM generator [10] embedded in Xilinx Coregen, which generates TCAM structures that are very similar to the TCAMs that TMAP synthesizes from the code in Figure 5.7. The results for the

```
entity tcam is
  generic (
    DATA_W   : integer := 32;
    ADDR_W   : integer := 8 );
  port (
    clk      : in    std_logic;
    entry    : in    entry_array; --PARAM
    datain   : in    std_logic_vector (DATA_W-1 downto 0);
    addrout  : out   std_logic_vector (ADDR_W-1 downto 0);
    match    : out   std_logic  );
end tcam;

architecture rtl of tcam is
begin
READ: process(clk) is
  variable local    : std_logic;
begin
  if rising_edge(clk) then
    match  <= '0';
    addrout <= (others => '0');
    for i in 2**ADDR_W-1 downto 0 loop

      local := entry(i). used;
      for j in 0 to DATA_W-1 loop
        local := local and ((entry(i). data(j) xnor datain(j)) or
                 entry(i). mask(j));
      end loop;

      if (local = '1') then
        match  <= '1';
        addrout <= std_logic_vector(to_unsigned(i, ADDR_W));
      end if;

    end loop;
  end if;
end process READ;
end architecture rtl ;
```

Figure 5.7: Parameterized HDL code for a TCAM where the 256 entries of width 32 bits are the parameters. Changing the content of the TCAM is done by means of reconfiguration.

Table 5.10: Comparison of different implementations of a TCAM on a Virtex-II Pro. (ISE) Synthesis from behavioral VHDL using ISE 9.2. (SRL16) Generated with Xilinx Coregen. (TMAP) Synthesis from behavioral VHDL (Figure 5.7) using TMAP.

Design		ISE			SRL16			TMAP		
Width	Entries	LUT	FF	f_{max} [MHz]	LUT	FF	f_{max} [MHz]	LUT	FF	f_{max} [MHz]
16	128	2516	4302	86.72	1504	127	79.26	1095	56	88.72
16	256	5100	8577	74.36	2886	130	68.24	2217	85	83.51
32	128	4569	8419	79.97	2664	223	68.13	1735	237	95.57
32	256	10441	16874	69.01	5070	226	59.69	3497	259	90.00

SRL16 TCAM are also shown in Table 5.10. As can be seen the SRL16 TCAM (256 entries of 32-bit) is 45% larger and clocks 34% slower than the TMAP design. This is mainly due to the infrastructure needed to write new entries in the TCAM.

Again, the gain in area and speed of the FPGA hardware of our dynamically reconfigurable TCAM comes at the cost of a larger time to write an entry. While in the ISE implementation an entry can be rewritten in one clock cycle and in the SRL16 implementation in 16 clock cycles, the TMAP implementation requires us to both generate a specialized configuration for the FPGA by evaluating the PPC and write this configuration in the configuration memory of the FPGA. In the next experiment, I measured the specialization overhead for the TMAP implementation. As is explained in Section 5.6.1, the evaluation of the PPC is done on the embedded PowerPC and the reconfiguration is done using the ICAP of the Virtex-II Pro. I did the measurement both in the case when only one entry needs to be rewritten and in the case when all entries are rewritten. The results are shown in Table 5.11. If only one entry is written, the reconfiguration time depends on the way the TLUTs of the entry are placed on the FPGA, because the placement determines the number of frames that need to be reconfigured. Table 5.11 shows the reconfiguration time for the worst case entry. For the largest TCAM (256 entries of 32 bit), reconfiguring one entry takes 245 μs in worst case and reconfiguring the full TCAM takes 1716 μs. More details can be found in the table.

The disadvantage of using generator software is that it results in architecture dependent designs, because the TCAM structures inter-

Table 5.11: Specialization overhead of different-sized TCAMs implemented on the Virtex-II Pro. (One Entry) Only one of the TCAM entries is written. (All Entries) All the entries of TCAM are written.

Design		One Entry (Worst Case)					All Entries				
Width	Entries	S_{eval} [B]	t_{eval} [μs]	frames	t_{reconf} [μs]	$t_{special}$ [μs]	S_{eval} [B]	t_{eval} [μs]	frames	t_{reconf} [μs]	$t_{special}$ [μs]
16	128	7362	1.50	4	104	106	7446	190	41	795	985
16	256	7362	1.50	4	104	106	7446	380	52	1034	1414
32	128	9750	2.84	9	205	208	9834	360	49	946	1306
32	256	9750	2.84	9	242	245	9834	720	52	996	1716

Table 5.12: Several TCAMs mapped to different-sized LUTs: $K = 3$, $K = 4$ and $K = 5$.

Design		$K = 3$			$K = 4$			$K = 5$		
Width	Entries	Conv.	TMAP		Conv.	TMAP		Conv.	TMAP	
16	128	3637	1604	(0.44)	2516	1095	(0.44)	2471	867	(0.35)
16	256	7278	3197	(0.44)	5100	2217	(0.44)	4863	1724	(0.36)
32	128	6958	3022	(0.43)	4569	1735	(0.38)	4780	1527	(0.32)
32	256	13919	6013	(0.43)	10441	3497	(0.34)	9337	3018	(0.32)

nally instantiate architecture specific resources. Our TMAP design does not have that problem. The VHDL code of Figure 5.7 is architecture independent. The same design can be mapped to several FPGA architectures by simply changing the mapper. Of course these mappers must have TLUT capability in order to benefit form the resource reduction. To strengthen this point I have also mapped the code of Figure 5.7 to architectures with different LUT sizes. The LUT usage for $K = 3$, $K = 4$ and $K = 5$ can be found in Table 5.12. In the table one can clearly see that for the TCAMs the relative area gain improves when the LUT size increases.

5.7 Conclusions and Future Work

In this chapter I have proposed a novel technology mapper called TMAP. The mapper maps Boolean circuits to Tunable LUTs (TLUTs). These are LUTs of which the truth table is expressed as a Boolean function of the parameter inputs.

I have shown that the complexity of TMAP scales as well as that

of a conventional LUT mapper, which results in an acceptable runtime for the offline technology mapping step. I have also shown that the complexity of evaluating the Boolean functions in order to generate new truth tables, and thus the time required to generate a specialized configuration, scales favorably with the size of the original design.

The approach is validated by implementing adaptive FIR filters and Ternary Content-Addressable Memories (TCAMs) on a Virtex-II Pro. I showed large reductions in the number of LUTs (39% for the FIR filters and 66% for the TCAMs) and significant improvements of the maximum clock frequency (38% for the FIR filters and 30% for the TCAMs). The specialization of both designs was done using the embedded PowerPC and the ICAP of the Virtex-II Pro. The total time needed to change the coefficients of the FIR filter is 1.74 ms and the content of the TCAM can be rewritten in 1.72 ms.

As was mentioned in 5.4, the focus in this chapter has been on the minimization of the depth and the area of the LUT structure. However, in TLUT mapping it might also be interesting to incorporate the reconfiguration effort in the cost function. During the mapping process, it should, for example, be possible to minimize the number of configuration bits that need to be reconfigured or the complexity of the reconfiguration procedure. This is the subject of further research.

Chapter 6

TLUT-based Reconfiguration on Commercial FPGAs

In this chapter, I discuss how a DCS system that is produced by the TLUT method can be implemented using a commercial FPGA. I have chosen more specifically the Xilinx Virtex-II Pro because at the time this research was performed it was a state-of-the-art partially reconfigurable component. However, the same principles can be used to build a DCS system on other FPGAs as long as they support dynamic partial reconfiguration.

I describe three different forms of reconfiguration. In Section 6.1 I discuss how a DCS system can be built that makes use of the ICAP to reconfigure the FPGA. The ICAP is a reconfiguration interface of the Virtex-II Pro that can be accessed from within the FPGA fabric. I make use of the ICAP driver provided by Xilinx and show that the use of this driver introduces a huge overhead in the reconfiguration process. To overcome this overhead, I introduce a new concept of parameterized bitstreams in Section 6.2, which not only allows us to speed up the reconfiguration process, but also enables us to evaluate the PPC using customized hardware. Finally, I abandon ICAP reconfiguration and switch to SRL reconfiguration (Section 6.3) and I show that it allows faster reconfiguration with limited impact on the area and clock frequency of the design.

The three forms of reconfiguration are used to implement different sized adaptive FIR filters. I compare these filter implementations in terms of area, clock frequency and reconfiguration overhead.

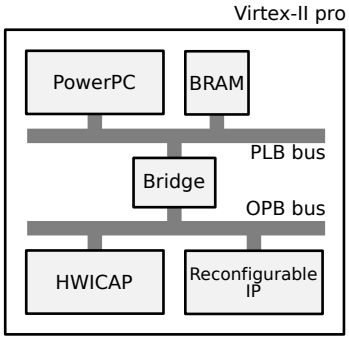


Figure 6.1: Self-reconfiguring platform implemented on a Xilinx Virtex-II Pro FPGA.

6.1 ICAP Reconfiguration

In this section, I show how a DCS system can be built on a Virtex-II Pro FPGA. During this process, I try to reuse the design tools provided by Xilinx as much as possible. This was described first in [12].

6.1.1 Typical DCS System on the Virtex-II Pro

Figure 6.1 shows the architecture of a DCS system that can be implemented on a Xilinx Virtex-II Pro FPGA. The specialization process is implemented in software on the embedded PowerPC of the Xilinx Virtex-II Pro FPGA, which ensures a tight connection to the FPGA fabric [9]. The interface between the specialization process and the configuration memory is realized using the PLB bus, the OPB bus and the Xilinx HWICAP module, which provides the interface between the OPB bus and the FPGA's ICAP (Internal Configuration Access Port). To configure the LUT truth tables after a parameter value has changed, the PowerPC generates a new configuration by evaluating the PPC and sends this new configuration to the FPGA configuration memory through the ICAP port of the FPGA via the HWICAP module. The DCS system shown in Figure 6.1 can easily be implemented using Xilinx XPS [90].

6.1.2 Embedding TMAP in the Xilinx Tool Flow

The tool flow used to implement a DCS system like the one shown in Figure 6.1, on the Virtex-II Pro FPGA is represented in Figure 6.2. The

tool flow tries to reuse as many of the Xilinx tools as possible and is therefore more complex than the tool flow represented in Chapter 4. Building a good quality tool flow from scratch is as good as impossible, because the detailed architecture of commercial FPGAs is not public and there is a very large effort involved in building good quality tools. Two concepts are used to make the use of the Xilinx tools possible: partial mapping and dummy truth tables, which will be explained later. All this could of course be avoided if Xilinx would incorporate the TMAP capabilities into their tool flow, but it shows that our DCS methodology already can be applied with the existing devices and existing tool flow.

Instead of producing a parameterized configuration, the tool flow directly creates a template configuration and one specialization procedure for each parameterized module. The template configuration contains all the configuration bits that remain static during the system's run-time. It is used to configure the FPGA at start-up. The specialization procedures are C functions that have the parameters as arguments. These C functions generate the new truth tables for the TLUTs using the PPC and reconfigure the LUTs.

In order to change one specific TLUT's function according to new parameter values, it is necessary to know the way its truth table is related to the parameter values, i.e. the tuning functions, and the location of the physical LUT that implements the TLUT under consideration. A tuning function is the Boolean function that expresses the content of a configuration bit as a function of the parameter inputs. The combination of this information for all TLUTs is used to generate a C procedure that reconfigures the FPGA. This is shown in the right-hand side branch of the tool flow of Figure 6.2. The arguments of the procedure are the parameter values.

Generating the Template Configuration

I assume that the input of the tool flow, the parameterized HDL design, contains a number of parameterized modules and a number of non-parameterized modules. I make this distinction because the Virtex-II Pro architecture is a very heterogeneous architecture compared to the homogeneous LUT architecture that TMAP targets. Therefore, using TMAP to map the full design would result in an inefficient use of the Virtex-II Pro architecture. The use of TMAP is thus limited to the parameterized modules, as is shown in Figure 6.2.

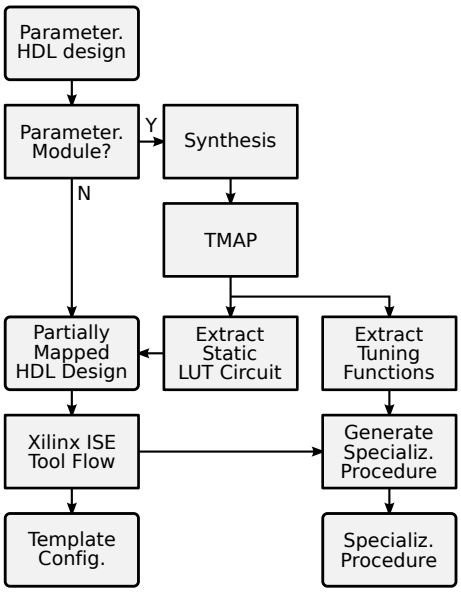


Figure 6.2: Practical tool flow for mapping a parameterized HDL design to a self-reconfiguring platform.

As was explained in Chapter 4, a parameterized VHDL module is nothing more than a regular VHDL description with annotations indicating which of the inputs are the parameter inputs. Figure 6.3 shows the parameterized module of the 6:1 multiplexer which will be used as an example. The annotation `--PARAM` indicates that the select inputs are parameters. As the annotations are in a comment line, any conventional synthesis tool can be used to synthesize the circuit. I used Altera Quartus II because it can dump a .blif file. The .blif file can easily be transformed to an .aig file that can then be used as input for TMAP, which was described in Chapter 5. The result produced by TMAP is a TLUT circuit.

The static LUT structure of the TLUT circuit is then expressed in VHDL by directly instantiating LUTs in the VHDL module. The Xilinx guidelines [91] for instantiating a LUT require the truth table of the LUT. For those LUTs in the TLUT circuit that have a static truth table this is not a problem, but for the tunable LUTs the truth tables are not known. Therefore, a dummy truth table is used. To prevent the Xilinx tools from changing the LUT structure, the truth tables must be chosen so that the Xilinx tools don't change the LUT structure. Apparently, this problem is solved by choosing a dummy

```
entity multiplexer is
port (
  i   : in    std_logic_vector (5 downto 0);
  s   : in    std_logic_vector (2 downto 0); --PARAM
  o   : out   std_logic
);
end multiplexer;

architecture behavior of multiplexer is
begin
  o <= i(conv_integer(s));
end behavior;
```

Figure 6.3: Parameterized VHDL module of the 6:1 multiplexer example.

truth table which contains only a single 1.

Combining the VHDL LUT structures with the non-parameterizable VHDL modules of the original design results in the partially mapped HDL design. This VHDL design can now be efficiently mapped to the Virtex-II Pro architecture with the Xilinx tools without corrupting the LUT structure of the parameterized modules found by TMAP. The result of this last mapping is the template configuration. Because some of the LUTs in the template configuration contain dummy truth tables, it is not functional until one specialization is performed by the specialization procedure.

It is important that every LUT instantiated in the VHDL LUT structure gets a unique name. This enables our tools to find the LUT's location after place and route (Section 6.1.2). Although it is not strictly necessary, I also locked the pins of the LUTs with the `lock_pins` attribute so that the router does not interchange the pins during routing. This greatly simplifies generating the reconfiguration procedure.

Building the Specialization Procedure

For each parameterized module a specialization procedure is created. This specialization procedure reconfigures all the TLUTs instantiated

in the module according to the parameter values that are passed as its arguments.

As mentioned before, in order to specialize a parameterized module upon a parameter change both the tuning functions of each of its TLUT and their location are needed. The tuning functions for each TLUT are provided by TMAP. The LUT locations are harder to come by. On the Virtex-II Pro a LUT location is specified by a slice row, a slice column and whether it is the F or the G LUT of the slice [87]. Finding these locations for each instantiated LUT is done in the following way. The Xilinx tool flow generates a .NCD file that contains all the information on the mapped circuit including the location of the LUTs. This .NCD file is first converted to a .XDL file, a clear-text representation of the .NCD file, using the Xilinx XDL program [58]. The LUT locations are found in this .XDL file by searching the unique names given to the LUTs when they were instantiated (section 6.1.2).

A reconfiguration procedure is then generated as follows. For each of the TLUTs in a parameterized module a TLUT specialization procedure is generated that takes the module parameter values as inputs, evaluates the tuning functions generated by TMAP and reconfigures the LUT. The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example is shown in Figure 6.4. The code that evaluates the tuning functions of a TLUT is generated by simply writing the Boolean expressions of the tuning functions using the C syntax. Note that, since the Virtex-II Pro family LUT configurations are stored in an inverted way [71], the configuration data must be inverted before configuring the LUTs. When executed, these expressions result in a new truth table for the LUT. The reconfiguration of the LUT is done by calling the procedure `XHwIcap_SetClbBits`, which is provided by Xilinx in the HWICAP module driver. This procedure takes the LUT location and the new truth table to reconfigure the LUT. In the example it is assumed that LUT L_1 is located in the G LUT of the slice at row 31 and column 45. The reconfiguration procedure for a module simply calls the TLUT reconfiguration procedure for each of the TLUTs of a module. The reconfiguration procedure for our 6:1 multiplexer example is shown in Figure 6.5.

Finally, the reader should be warned that, in the Virtex-II Pro family, reconfiguring a LUT will cause corrupted data in the SRL16s and LUT RAMs that are located in the same column. Therefore, placing TLUTs in the same columns as SRL16s or LUT RAMs must be avoided. This can be done using `AREA_GROUP` constraints. This is no

```
void L1( XHwIcap *hwIcap, Xuint8 S0, Xuint8 S1, Xuint8 S2) {
  Xuint8 truthTable[LUT_SIZE];

  truthTable [0]  = !(0) ;
  truthTable [1]  = !( S0 && S1);
  truthTable [2]  = !(! S0 && S1);
  truthTable [3]  = !( S1);
  truthTable [4]  = !( S0 && !S1);
  truthTable [5]  = !( S0);
  truthTable [6]  = !((! S0 && S1) || (S0 && !S1));
  truthTable [7]  = !( S0 || S1);
  truthTable [8]  = !(! S0 && !S1);
  truthTable [9]  = !(( S0 && S1) || (! S0 && !S1));
  truthTable [10]= !(! S0);
  truthTable [11]= !(! S0 || S1);
  truthTable [12]= !(! S1);
  truthTable [13]= !( S0 || !S1);
  truthTable [14]= !(! S0 || !S1);
  truthTable [15]= !(1) ;

  XHwIcap_SetClbBits( hwIcap, 31, 45, G_LUT, truthTable, LUT_SIZE);
}
```

Figure 6.4: The TLUT reconfiguration procedure for LUT L_1 of our 6:1 multiplexer example. It is assumed that LUT L_1 is located in the G LUT of the slice at row 31 and column 45.

```
void mux2w1 ( XHwIcap *hwIcap, Xuint8 S0, Xuint8 S1, Xuint8 S2) {
  L0(hwIcap, S0, S1, S2);
  L1(hwIcap, S0, S1, S2);
}
```

Figure 6.5: The reconfiguration procedure for our 6:1 multiplexer example.

Table 6.1: Comparison of the conventional implementation and the reconfigurable implementation.

	Conventional	ICAP reconfiguration
FIR Area (LUTs)	4,259	1,985
System Area (LUTs)	1,218	1,298
Total Area (LUTs)	5,477	3,283
Reconf. time (ms)	N/A	151

longer an issue in the Virtex-5 family, see Section 2.1.2.

6.1.3 Experimental Results

In this section I build an adaptive FIR filter, where the coefficients of the filter are the parameters of the DCS system. The filter is a fully pipelined 32-tap FIR filter with 8-bit coefficients and an 8-bit input.

This system is implemented on a Xilinx XUP board in two different ways. The first way is the conventional way. The filter is implemented using generic multipliers and coefficient values are handled as regular inputs to the filter. The coefficients are stored in registers which are mapped in the PowerPC's memory through the PLB and OPB buses. The coefficient manager is implemented in software on the PowerPC. It changes the filter characteristics by writing registers.

The second implementation uses the DCS tool flow to generate a reconfigurable FIR filter. Again, the coefficient manager is implemented in software, but now it changes the filter characteristics by reconfiguring the FPGA through the ICAP, as explained above.

In both implementations the PowerPC is clocked at 200 MHz and the busses are clocked at 66 MHz.[1] The resource usage of both implementations can be found in Table 6.1. The table shows that the reconfigurable implementation requires in total 2,194 (40%) fewer LUTs to implement the adaptive filter. This is mainly because of the size reduction (by over 53%) of the run-time reconfigurable FIR filter versus the generic FIR filter. The coefficient controller is only slightly bigger in the reconfigurable implementation because of the additional HWICAP module that is needed to connect the PowerPC

[1]The maximal clock frequency of the ICAP port in the Virtex-II Pro family is 66 MHz.

to the ICAP, and the memory needed to store the reconfiguration procedure which occupies 30% of the total memory of the PowerPC.

Of course, this size reduction does not come for free. The time needed to change the coefficients in the reconfigurable implementation, 151 ms, is much larger than for the conventional implementation, which requires only a few clock cycles to change the coefficients. No less than 126 ms of the specialization overhead is spent in the Xilinx driver function to reconfigure a LUT (`XHwIcap_SetClbBits`) and only 25 ms is spent on evaluating the Boolean functions. The `XHwIcap_SetClbBits` function is very inefficient. Every time the function is called to reconfigure a LUT, it reads back the frame that contains the LUT's truth table, changes the truth table and writes back the frame. When several TLUTs reside in the same frame, this procedure is repeated for every TLUT. The reconfiguration overhead could thus be reduced by switching from LUT-based reconfiguration to a frame-based reconfiguration. This is one of the effects of using parameterized bitstreams, which are described in the following section.

6.2 Parameterized Bitstreams

In commercial FPGAs, like the Xilinx Virtex-II Pro, the configuration memory is not accessible through a simple memory interface with an address bus and a data bus. Instead, the FPGA contains a configuration controller that processes the bitstreams that are sent to it. These bitstreams are a series of commands and data that not only allow to write data to the configuration memory of the FPGA but also to control things like starting up the FPGA, CRC checks,... This means that every time the parameters in a DCS system change, not only is it necessary to generate new content for the configuration memory of the FPGA, but it is also necessary to write this configuration data in the form of a bitstream that is compatible with the target FPGA.

Whenever the parameters of a certain module change value the corresponding configuration bits (those that depend on the parameters) need to be reconfigured. Such a reconfiguration is performed by sending a bitstream to the FPGA. If the bitstreams for different parameter values are compared, it is easy to see that these bitstreams all have the same form. The commands are the same, only the data fields of the bitstreams differ. This is because the positions of the

configuration bits that need to be reconfigured are independent of the parameter value.

In a regular bitstream the stream of commands and data is represented as a stream of Boolean values. In the light of the parameterized configuration, Parameterized Bitstreams (PBS) [1] are defined as a stream of Boolean functions. Again a PBS can easily be converted to a regular bitstream given a set of parameter values, by simply evaluating the Boolean functions. The resulting regular bitstream can be directly used to reconfigure the FPGA, without the need of additional processing.

After the introduction of the PBS concept, the obvious next step is to extend the configuration controller so that it can directly process PBSs. Or, in other words, when a PBS and a set of parameter values is sent to the configuration controller, the FPGA is reconfigured accordingly. This means that the configuration controller hardware will need to be able to evaluate the Boolean functions that are present in the PBS. In this section such a configuration controller is built by adding a front-end to the configuration controller of the Xilinx Virtex-II Pro. The main design issue when building this system is how to represent the PBS in a compact way while at the same time keeping the complexity of the front-end hardware low.

6.2.1 The PBS Specializer

As was already mentioned, the Xilinx Virtex-II Pro configuration controller is extended by adding a front-end. This front-end is called the PBS specializer. It is responsable for producing a regular bitstream for the Virtex-II Pro when given a set of parameter values and a PBS. This regular bitstream is then sent to the configuration controller of the FPGA in order to complete the reconfiguration.

In principle, the most economic way to implement the PBS specializer would be to hardwire it on the FPGA chip. Since this was impossible the specializer is implemented in the FPGA fabric as is shown in Figure 6.6. The PBS and the parameter values are inputs to the specializer and the specialized bitstream is the output. The specialized output bitstream is sent to the configuration control logic of the FPGA through the Xilinx ICAP (Internal Configuration Access Port) interface, which results in the reconfiguration of the FPGA.

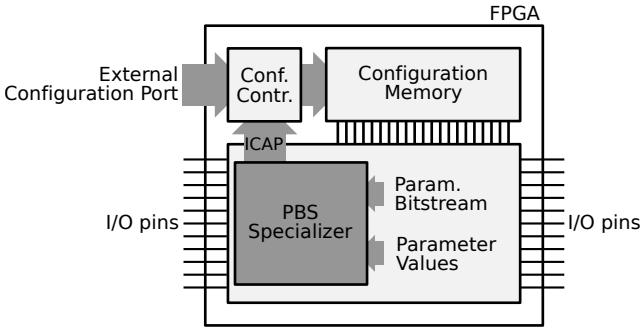


Figure 6.6: FPGA architecture (in a light gray color) with a front-end specializer

6.2.2 The Stack Machine Architecture

A stack machine architecture (Figure 6.7) is chosen for the PBS specializer. As a consequence the PBS is represented as an instruction stream for the stack machine. The stack machine architecture was chosen because it is able to evaluate Boolean functions using little hardware. Moreover, a stack machine can achieve higher code density [41, 37] compared to the accumulator machine and other machines, since the instructions executed by the stack machine are small and simple. This results in a compact representation of the PBS.

The detailed architecture of the stack machine is shown in Figure 6.7. Just like any stack machine it contains a stack to store intermediate results, an ALU that executes the instructions and an instruction decode unit that generates the internal control signals given the current instruction. Our architecture also includes a parameter memory which stores the current parameter values. New parameter values can be written through a write port of the memory.

The main task of the stack machine is to evaluate a PBS and produce a specialized bitstream for the FPGA. Each bit of the specialized bitstream is expressed as a Boolean expression using OR, AND, and NOT operators. Only 8 instructions are needed to evaluate such a PBS. They are shown in Table 6.2 together with their opcode and description. Only the PUSH instruction has an operand: the address of the parameter in the parameter memory. The PUSH instruction pushes a parameter value on the top of the stack. The logic operations (AND, OR and NOT) are executed by the ALU. Their operands are popped from the stack and the result is pushed on the top of

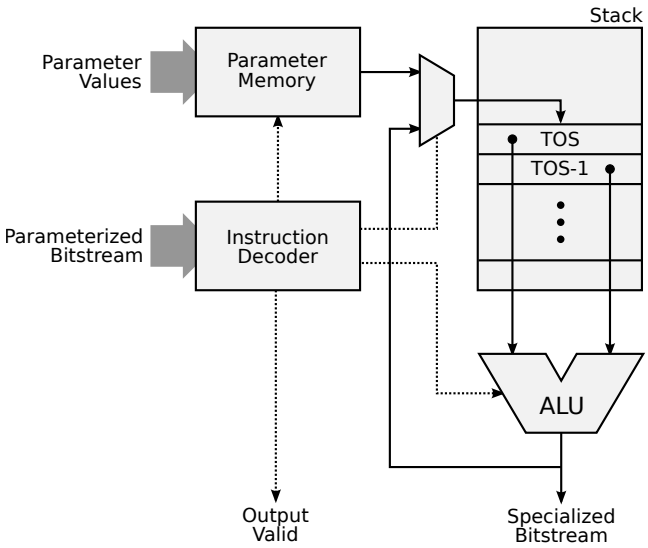


Figure 6.7: Stack machine architecture

the stack. The POP, OUT0 and OUT1 instructions are used to create the output stream. The POP instruction pops the TOS (Top Of Stack) and writes it to the output stream. This instruction always ends the group of instructions that evaluate a Boolean function. The OUT0 and OUT1 instructions write a constant 0 an a constant 1 to the output stream, respectively. These instructions are needed because many of the bits in the PBS are constant Boolean values.

6.2.3 The Compilation Process

In this section I explain how a compiled PBS is generated starting from a parameterized configuration. Therefore three different representations for the PBS are introduced: the first is the conceptual PBS which is a stream of Boolean expressions, the second is the instruction PBS, a stream of instructions and the last one is the encoded PBS, a stream of encoded instructions. The encoded PBS is the final result of the compilation process.

In the first stage, a conceptual PBS is generated from the parameterized configuration. First all the frames[2] that need to be reconfigured in case of a parameter change are identified. From the location

[2] A frame is the atom of reconfiguration in the Virtex-II Pro. See Section 2.1.2 for more information

Table 6.2: Stack machine instruction set

Instruction	Opcode	Description
NOP	000	No operation
AND	001	Logic AND
OR	010	Logic OR
NOT	011	Complement
PUSH x	100	Push parameter value to TOS
POP	101	Pop TOS
OUT0	110	Output constant 0
OUT1	111	Output constant 1

and content of the frames a conceptual PBS can be generated. The conceptual PBS contains both commands for the configuration controller of the Virtex-II Pro and the frame data. The commands are independent of the parameters and thus are a series of Boolean values, while the frame data can contain both Boolean values and Boolean expressions. Figure 6.8 shows the compilation process for a small PBS fragment. The first column shows the conceptual PBS.

The second stage converts the conceptual PBS to the instruction PBS. Each Boolean value is represented by either the OUT0 instruction or the OUT1 instruction. Boolean expressions are represented as a stream of instructions which can easily be found by writing the expression in a postfix notation. The second column in Figure 6.8 shows the instruction PBS for our example PBS fragment.

The last stage replaces each of the symbolic instructions in the instruction PBS to its binary encoding, as is shown in Table 6.2. The third column in Figure 6.8 shows the encoded PBS for our example PBS fragment. In the example, it is assumed that the parameter A is located at address 0 and parameter B is located at address 1. An address is represented by three bits.

6.2.4 Experimental Results

Again, the 32-tap adaptive FIR filter is implemented on the XUP board, but now the PBS specializer is responsible for generating the specialized configuration and reconfiguring the FPGA. The size of the filter does not change when switching from ICAP reconfiguration

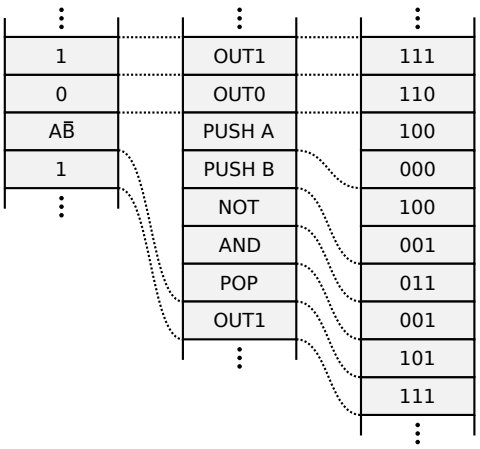


Figure 6.8: Example of the compilation process of a PBS fragment. From left to right: the conceptual PBS, the instruction PBS and the encoded PBS.

to PBS reconfiguration. Only the way the specialization is performed changes.

The PBS specializer was built in the Virtex-II Pro using only 217 LUTs and one BRAM and it was clocked at 100MHz. In these circumstances, the specializer was able to specialize (evaluate the PPC and reconfigure the FPGA) the filter in only 13.8 ms, which is a huge improvement compared to the ICAP reconfiguration described in Section 6.1 where it took 151 ms to reconfigure the same filter. This decrease in reconfiguration overhead is mainly due to the frame based reconfiguration which is inherent to PBS reconfiguration and also due to the fact that the stack machine is more effective in evaluating Boolean functions.

The disadvantage of PBS reconfiguration is the size of the PBS itself. In this experiment, the size of the PBS was 467 kB which is large compared to the available RAM on the FPGA. However, this problem could be solved by exploiting the repetitive nature of the FIR filter, and many other applications that are suited for DCS. Instead of representing the PBS as an instruction stream for the stack machine which has 32 copies of the instructions needed to reconfigure a multiplier, the PBS could be represented as a regular program with a loop that calls the multiplier reconfiguration 32 times. This could reduce the size of the PBS by a factor of about 32.

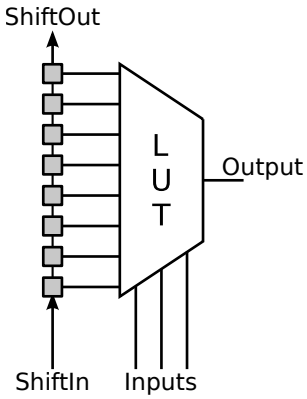


Figure 6.9: Schematic view of a Shift Register LUT (SRL).

6.3 SRL Reconfiguration

As mentioned in Section 2.1.2, the basic building block of the Virtex-II Pro FPGA is not a simple LUT but a Shift Register LUT (SRL). An SRL is a LUT in which the 16 configuration bits are arranged as a shift register of which the input and the output are accessible from the configurable routing (Figure 6.9). Therefore the truth table configuration bits can not only be changed through the configuration interface of the FPGA but also by shifting a new truth table in the SRL. Since a parameter change in the TMAP tool flow only requires reconfiguration of LUTs and not of routing bits, the SRL reconfiguration naturally fits the TMAP tool flow.

6.3.1 SRL DCS System

Since the reconfiguration of the FPGA will no longer be done through the ICAP interface the DCS system described in Section 6.1.1 needs to be adapted. In order to make the truth table bits of multiple TLUTs accessible from the PowerPC, they are grouped and each group is arranged as a larger shift register, called a *reconfiguration path*, by connecting the shift out of a TLUT to the shift in of the next TLUT. The shift in of the first TLUT of each reconfiguration path is connected to the HWSRL, which replaces the HWICAP and interfaces directly to the PLB bus (Figure 6.10). The HWSRL is basically a FIFO that buffers the reconfiguration data, with some logic added to start and stop the reconfiguration. On the side of the PLB the FIFO is 32 bit

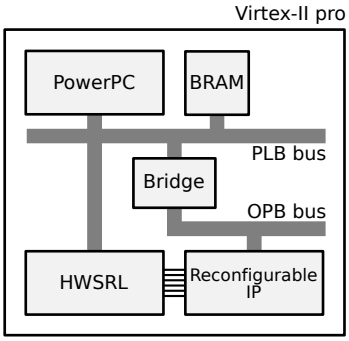


Figure 6.10: Self-reconfiguring platform implemented on a Xilinx Virtex-II Pro FPGA using SRL reconfiguration.

wide, while on the side of the reconfiguration paths the width depends on the number of reconfiguration paths. Figure 6.10 shows a self reconfiguring platform that makes use of SRL reconfiguration. In the example 6 reconfiguration paths are used.

There are three degrees of freedom when constructing the reconfiguration infrastructure. It is possible to vary (i) the number of reconfiguration paths, (ii) how TLUTs are partitioned into reconfiguration paths, and (iii) the order of the TLUTs in the reconfiguration paths. The number of paths can be adjusted to adapt the reconfiguration speed to the needs of the application, since configuration data can be shifted in the configuration paths in parallel. This is an advantage over ICAP reconfiguration where the reconfiguration speed is fixed by the bandwidth of the ICAP. The two remaining degrees of freedom can be used to minimize the impact of the reconfiguration infrastructure on the design. Indeed, the extra connections may make the design harder to place and route, and thus it might be harder to reach timing closure. Therefore, the reconfiguration paths need to be carefully constructed.

6.3.2 Tool Flow

The tool flow for SRL reconfiguration differs from the normal tool flow mainly in that extra routing needs to be inserted after the TMAP step in order to implement the reconfiguration paths. This extra routing can be inserted at different points in the tool flow. In my opinion there are three options: before placement, during placement and after placement.

If the SRL routing is inserted before placement there is no placement information available yet, which makes it hard to calculate the cost of one arrangement of the reconfiguration paths or, in other words, choosing a good arrangement is difficult at this point. However, the topology of the TLUT circuit can be used to estimate the cost. TLUTs that are closely connected are likely to be placed closely together.

If the routing is inserted after placement, the positions of the TLUTs are known. This makes it easier to estimate the cost of one specific arrangement and thus to choose a good arrangement. The problem of finding a good arrangement for the reconfiguration infrastructure can be reduced to a traveling salesman problem with multiple salesmen, where the number of salesmen is equal to the number of reconfiguration paths. The advantage of this option is that the routing infrastructure has no impact on the placement of the design. This can, however, also lead to long reconfiguration times, since the placement tool didn't take the reconfiguration paths into account during placement.

If both the reconfiguration speed and the performance of the design under need to be controlled, the reconfiguration paths need to be inserted during placement. The problem with this approach is that a simulated annealing placer needs to estimate the cost of the placement in every iteration. In the worst case this means that the traveling salesman problem needs to be solved in the inner loop of the placer, which could lead to long execution times.

At this point none of these three automatic methods have been implemented. In the current tool flow the arrangement of the reconfiguration infrastructure is done manually. Further research will point out which option gives the best quality/cost tradeoff.

Previously, the truth table reconfiguration bits were calculated and then written in the configuration memory by using the location of the LUT as an address (Section 6.1.2). Now, the order of the TLUTs in a reconfiguration path is important, not the exact location of the LUT on the FPGA. Since the order of the TLUTs is known at compile time, the reconfiguration procedure described in Section 6.1.2 can be adapted in such a way that the order in which the truth tables bits are generated and sent to the HWSRL is done according to the order in which TLUTs are arranged in the reconfiguration paths.

6.3.3 Experimental Results

When switching from ICAP reconfiguration (Section 6.1) to PBS reconfiguration (Section 6.2) there was a big reduction in reconfiguration overhead for our 32-tap adaptive FIR filter. Only 1009 μs of the PBS reconfiguration overhead of 13.8 ms is actually spent on reconfiguring the FPGA. So one could think that it is not really useful to further reduce the reconfiguration time. However, the previous experiments assumed that no slack was available, i.e. the time between the moment new parameter values get known and the moment the reconfiguration must be finished (Section 3.2). If the slack is large enough, new specialized bitstreams can be calculated beforehand and loaded when needed. In this case only the reconfiguration time is considered overhead. Therefore, it is useful to further reduce the reconfiguration time.

Again, the 32-tap adaptive FIR filter is implemented on the XUP board in almost the same way as was done in the previous experiments. The only difference now is that it will make use of SRL reconfiguration, which requires the implementation of the reconfiguration paths. The reconfiguration paths only require extra connections, the number of LUTs remains the same as in the previous experiments. As explained above, the reconfiguration paths could be instantiated automatically, but this is not yet implemented. Therefore, the reconfiguration paths are instantiated manually in this experiment. The paths basically follow the one dimensional structure of the pipelined FIR filter. In the future this task will be performed automatically. Since the reconfiguration time for SRL reconfiguration will depend on the number of reconfiguration paths, different filters have been implemented that use 1, 2, 4, 8, 16, or 32 paths of balanced length.

Besides the reconfiguration time, the effect the introduction of the reconfiguration paths has on the quality of the design was also measured. Indeed, since extra connections are introduced in the design it is expected that it will be harder to place and route the design and thus a reduction of the maximum clock frequency of the design is to be expected.

Table 6.3 shows the results of the experiment. The degradation in maximum frequency is limited to only 2%. This is because the extra connections are not critical. They directly connect a FF output to a FF input without any logic in between, which results in a lot of timing slack for the extra connections. Thus, they have limited influence on the placement. The speedup in reconfiguration time is almost linear

Table 6.3: Comparison between SRL reconfiguration and ICAP reconfiguration of a 32-tap adaptive FIR filter. Both the degradation of the maximum clock frequency and the speedup of the reconfiguration time are shown.

		SRL reconfiguration [paths]					
	ICAP	1	2	4	8	16	32
f_{max} [MHz]	151	150	151	151	150	148	150
Degradation [%]	0.0	0.7	0	0	0.7	2.04	0.7
t_{reconf} [μs]	1009	81.9	40.7	20.3	10.2	5.2	2.6
Speedup	1.0	12	25	50	99	194	388

in the number of paths and ranges from a factor 12 for one configuration path to a factor 388 in the case of 32 reconfiguration paths. In this last case, the filter can be reconfigured in only 2.6 μs. The reason for this speedup is threefold. First, only the truth tables of the TLUTs are present in the reconfiguration path while in ICAP reconfiguration all the LUTs in a full frame need to be reconfigured even if they have static truth tables. Second, a FPGA bitstream that reconfigures only LUTs is full of padding frames, which also need to be sent to the ICAP. Third, in SRL reconfiguration, the bandwidth of the reconfiguration interface can be increased by simply increasing the number of reconfiguration paths while the bandwidth of the ICAP is fixed.

6.4 Conclusions and Future Work

In this chapter, I have shown several ways to implement TLUT DCS systems on a commercial FPGA and validated each of them on a 32-tap adaptive FIR filter. In Section 6.2, I introduced the concept of parameterized bitstreams as an extension of the parameterized configuration concept. It enabled us to greatly reduce the specialization time compared to conventional ICAP reconfiguration. In Section 2.1.2, I discussed SRL reconfiguration and showed that using the SRL functionality of the LUTs can greatly reduce the reconfiguration time compared to ICAP reconfiguration. A nice feature of SRL reconfiguration is the fact that the designer can choose the number of reconfiguration paths and thus can adapt the reconfiguration bandwidth to his needs.

The techniques presented in this chapter can be further refined in the future. The introduction of reconfiguration paths for SRL reconfiguration is at this point done manually. As was mentioned above, this could easily be automated. The main problem for the PBS reconfiguration is the size of the PBS itself. This is mainly because the PBS is represented as an instruction stream and not as a regular program containing jumps and function calls. Many applications that are suitable for DCS however have a repetitive structure, which could be exploited to greatly reduce the program size if control instructions were available.

Functionally, many applications have a repetitive structure, for example when a module is instantiated several times. For each of these modules the same Boolean functions need to be evaluated in order to generate a specialized configuration. This could be done by calling the same function several times. However, the placement tool generally does not preserve the repetitive structure of the implementation. The TLUTs of the modules are placed in a somewhat random way. Therefore, the configuration bits of a module need to be generated in a differed order for each module. This requires us to reorder the configuration bits for each of the modules, which will increase the complexity of the reconfiguration procedure and the memory needed. However, this problem can be completely resolved by combining PBS reconfiguration and SRL reconfiguration. Indeed, if the order of the TLUTs of a module is the same for every module, its possible to simply generate the reconfiguration bits for one module and shift them in the reconfiguration path without reordering of the bits. Thus reducing the complexity of the reconfiguration procedure. One could see the combination of PBS reconfiguration and SRL reconfiguration as a way to decouple the placement problem and the problem of optimizing the reconfiguration procedure.

Chapter 7

Towards Reconfigurable Routing

The TLUT method described in Chapter 4 is an important step forward in the automatic implementation of DCS systems. Although good results were obtained in terms of area and clock frequency, only allowing reconfiguration of truth table bits can, for some applications, limit the possible gain of using DCS. In this chapter, I therefore extend the TLUT method so that it produces a PPC in which not only the truth table bits can be expressed as a function of the parameter inputs, but also the routing bits. This new method is called the TCON method. As I will show in Section 7.5, the TCON method can in certain cases lead to a further reduction of the number of LUTs.

The problem faced by the TCON method is again to produce a parameterized configuration given a parameterized HDL description while minimizing some cost function, but now the routing bits in the parameterized configuration can also be expressed as a function of the parameter inputs. Similar to conventional FPGA mapping, the mapping problem is divided into four subproblems: synthesis, technology mapping, placement and routing. This can be seen in Figure 7.1.

Since both the input and output of the method are parameter-

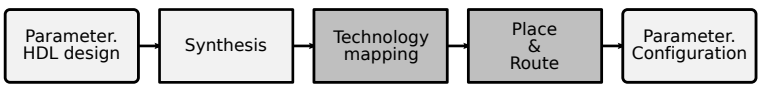


Figure 7.1: Overview of the TCON method.

ized, this parameterizability needs to be preserved by each of the algorithms in the TCON method. In the TLUT method the parameterizability was preserved by introducing the TLUT, an abstraction of a LUT that reflects the reconfigurability of that LUT by expressing its truth table bits as functions of the parameters. In order to also make the routing bits of the FPGA reconfigurable, a new structure is introduced which is called a *Tunable Connection* or TCON (see Section 7.1). Just like a net, a TCON is an abstraction of a subset of an FPGA's routing resources, but in contrary to a net, a TCON reflects the reconfigurability of those routing resources.

The algorithms used to solve the subproblems are adapted versions of the conventional algorithms (see Chapter 2). The parameterized gate-level circuit produced by the synthesis tool is mapped to a *Tunable Circuit* by an adapted technology mapper. This tunable circuit contains a mixture of TLUTs and TCONs. Next, an adapted placement tool places each of the TLUTs, whereafter an adapted router routes each of the TCONs. In this chapter I describe: the technology mapping algorithm (Section 7.2), the placement algorithm (Section 7.3) and the routing algorithm (Section 7.4). At this point, a simple adapted placer and an adapted router, called the pattern router (Section 7.4.2), are implemented. The adapted mapper is not. Nevertheless, I describe possible approaches to implement each of these algorithms. Further, I also describe an improved TCON routing algorithm, called the connection router (Section 7.4.3).

7.1 Tunable Connections

As was already mentioned, a TCON is an abstraction of a subset of an FPGA's routing resources that reflects the reconfigurability of those routing resources. In the TCON method, TCONs first appear after technology mapping and are then further refined by the placement and routing steps. Figure 7.2 shows the example of a TCON that implements a four-way switch. The left-hand side shows the TCON after technology mapping when it is no more than a functional block. The parameter inputs of the TCON control the way in which the regular inputs of the TCON are connected to the outputs of the TCON. The right-hand side shows that same TCON after it has been refined by place and route. At this level, the TCON consists of a set of routing resources (wires and switches), where each of the switches is con-

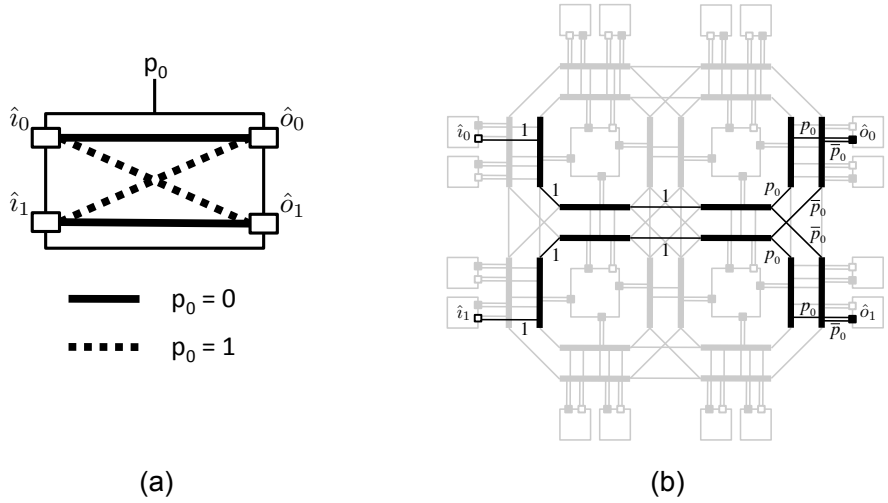


Figure 7.2: (a) Schematic representation of a TCON with the functionality of a four-way switch. (b) Implementation of that TCON (black) in a simple 2×2 island style FPGA resource graph (grey). Wires are solid lines; Edges are thin lines; Sources are open boxes; And sinks are filled boxes.

trolled by a Boolean function of the parameter inputs, called a tuning function.

7.1.1 Functional Level Description of TCONs

At the functional level, a TCON, $\tau = (\mathcal{I}, \mathcal{O}, \zeta_p)$, is a functional block with a number of regular inputs, parameter inputs and outputs. The set of inputs is defined as $\mathcal{I} = \{\hat{\imath}_0, \hat{\imath}_1, \ldots, \hat{\imath}_{L-1}\}$ and the set of outputs is defined as $\mathcal{O} = \{\hat{o}_0, \hat{o}_1, \ldots, \hat{o}_{M-1}\}$. Further, the set of possible parameter values is defined as $P = \{0, 1\}^N$ and $p = (p_0, p_1, \ldots, p_{N-1}) \in P$.

Because of the nature of the routing infrastructure of FPGAs, TCONs can only implement functions in which the parameter values control the way the inputs and outputs of the TCON are connected. This functionality is represented by the *Connection Function* of the TCON ζ_p. In what follows it is assumed that TCONs are directed. Thus, a signal always travels through the TCON from the inputs to outputs of the TCON.

The Connection Function

Before presenting the details of the TCON connection function, I first describe a *Connection Pattern*. A connection pattern, $\pi \in \Pi$, is one way of connecting the inputs of a TCON to its outputs, where Π is the set of all possible connection patterns. Mathematically a connection pattern is a function, $\pi(\hat{o})$, that maps every output of the TCON to one of its inputs.

$$\pi(\hat{o}) : \mathcal{O} \rightarrow \mathcal{I} : \hat{o} \mapsto \hat{i} \tag{7.1}$$

The fact that outputs are mapped to inputs and not visa versa is important since this eliminates short circuits and at the same time allows fanout of the inputs. Indeed, since π is a function every output is mapped to maximally one input (no short circuits), but multiple outputs can map to the same input (fanout).

As was mentioned above, the way the outputs of a TCON are connected to its inputs is not fixed, but depends on the parameter value. This functionality is represented by a connection function ζ_p that maps a parameter value to a connection pattern, or in other words an output is mapped to an input depending on the parameter value.

$$\zeta_p : P' \rightarrow \mathcal{O} \rightarrow \mathcal{I} = P' \rightarrow \Pi : p \mapsto \pi \tag{7.2}$$

In some cases, certain parameter values $p \in P$ are not allowed or don't have a meaning. Therefore the domain of $\zeta_p(\hat{o})$ is not P, but $P' \subseteq P$ which contains all valid parameter values.

The four-way-switch TCON shown in Figure 7.2 has two inputs $\{\hat{i}_0, \hat{i}_1\}$ and two outputs $\{\hat{o}_0, \hat{o}_1\}$, and the parameter p_0 controls how the inputs are connected to the outputs. As is shown in Figure 7.2(a), when $p_0 = 0$, $\hat{o}_0$ is connected to $\hat{i}_0$ and $\hat{o}_1$ is connected to $\hat{i}_1$; when $p_0 = 1$, $\hat{o}_0$ is connected to $\hat{i}_1$ and $\hat{o}_1$ is connected to $\hat{i}_0$. The connection function of the four-way switch can then be written as

$$\zeta_p^{4w}(\hat{o}) = \begin{cases} \hat{i}_0 & \text{if } (\hat{o} = \hat{o}_0 \wedge p_0 = 0) \vee (\hat{o} = \hat{o}_1 \wedge p_0 = 1) \\ \hat{i}_1 & \text{otherwise} \end{cases} . \tag{7.3}$$

Pattern Sets and Pattern Conditions

The *Pattern Set* of a TCON τ, $\Pi^\tau \subseteq \Pi$, is the set of all possible patterns of τ. It can be derived from the connection function of τ as

$$\Pi^\tau = \{\pi \in \Pi : \exists p \in P'(\pi(\hat{o}) \equiv \zeta_p(\hat{o})\}. \tag{7.4}$$

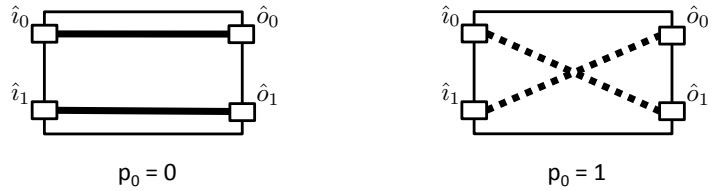


Figure 7.3: Graphic representation of the patterns associated to the four-way switch example.

The pattern set associated to the four-way switch contains two patterns, which are graphically represented in Figure 7.3.

For every connection pattern $\pi \in \Pi^\tau$, it is possible to define a Boolean function $\kappa_\pi(p)$, called the *Pattern Condition*, that evaluates to one for those parameter values where the connection function maps to π. Formally, the pattern condition is defined as

$$\begin{aligned} \kappa_\pi(p) \quad : \quad \Pi^\tau \rightarrow P' \quad &\rightarrow \quad \{0,1\} \\ \kappa_\pi(p) \quad &= \quad \begin{cases} 1 & \text{if } \zeta_p = \pi \\ 0 & \text{otherwise} \end{cases} . \end{aligned} \tag{7.5}$$

The four-way switch has two connection patterns (Figure 7.3). The pattern π_0 with pattern condition $\kappa_{\pi_0}(p) = \overline{p}$ and pattern π_1 with pattern condition $\kappa_{\pi_1}(p) = p$.

Connections and Nets

A connection pattern can also be represented as a set of connections. A *Connection* is an ordered pair $(so \in \mathcal{I}, si \in \mathcal{O})$, where so is called the source of the connection and si is called the sink of the connection.

In the four-way switch example, pattern π_0 contains two connections, $(\hat{\imath}_0, \hat{o}_0)$ and $(\hat{\imath}_1, \hat{o}_1)$, and pattern π_1 contains two connections, $(\hat{\imath}_0, \hat{o}_1)$ and $(\hat{\imath}_1, \hat{o}_0)$.

The connections in a pattern that have the same sources can be grouped to form nets. A *Net* can be represented as an ordered pair $(so \in \mathcal{I}, SI \subseteq \mathcal{O})$, where so is the source of the net and SI are the sinks of the net. If the connections of a pattern are grouped as nets, a pattern can be represented as a set of nets.

The nets in the four-way switch example only have one sink. Pattern π_0 contains two nets, $(\hat{\imath}_0, \{\hat{o}_0\})$ and $(\hat{\imath}_1, \{\hat{o}_1\})$, and pattern π_1 contains two nets, $(\hat{\imath}_0, \{\hat{o}_1\})$ and $(\hat{\imath}_1, \{\hat{o}_0\})$.

Boolean Function of a TCON

The functionality of a TCON τ can also be expressed as a multi-valued Boolean function, $o = \beta^\tau(i,p)$, where $o = (o_0, o_1, \ldots, o_{M-1})$ (with $o \in O = \{0,1\}^M$) represents the Boolean values of the TCON's outputs, $i = (i_0, i_1, \ldots, i_{L-1})$ (with $i \in I = \{0,1\}^L$) represents the Boolean values of the TCON's inputs and, as already defined, $p \in P'$ represents the Boolean values of the parameter inputs.

This Boolean function can be constructed from the connection function, ζ^τ as follows. First, I define the Boolean connection function $\widetilde{\zeta}^\tau_{\hat{o},\hat{i}}(p)$ that evaluates to one when $\hat{i}$ is connected to $\hat{o}$ given p. Formally,

$$\begin{array}{rcl} \widetilde{\zeta}^\tau \;:\; \mathcal{O} \to \mathcal{I} \to P' & \to & \{0,1\} \\ \widetilde{\zeta}^\tau_{\hat{o},\hat{i}}(p) & = & \begin{cases} 1 & \text{if } \zeta^\tau_p(\hat{o}) = \hat{i} \\ 0 & \text{otherwise} \end{cases} . \end{array} \tag{7.6}$$

The Boolean functionality of a TCON τ can now be written as

$$o = \beta^\tau(i,p) = \boldsymbol{C}(p)\; i = \left[\widetilde{\zeta}^\tau_{\hat{o},\hat{i}}(p)\right]\; i \tag{7.7}$$

where $\boldsymbol{C}$ is called the connection matrix.

From Equation (7.2) follows that for a given parameter value every output of a TCON is mapped on one input of the TCON. Therefore, no two elements of a row of the connection matrix can evaluate to one simultaniously, or formally

$$c_{ki}(p) c_{kj}(p) \equiv 0 \quad \forall i \neq j, \forall k. \tag{7.8}$$

The inverse is also true. A Boolean function that can be written in the form given in Equation (7.7) under condition (7.8), can be implemented in the routing infrastructure of an FPGA.

7.1.2 Merging TCONs

Two or more TCONs that are connected in a legal way (no short circuits and no loops) can be merged to form a new TCON. Figure 7.4 shows how two four-way switches can be combined to form a new TCON with three inputs $\{\hat{i}_0, \hat{i}_1, \hat{i}_2\}$ and three outputs $\{\hat{o}_0, \hat{o}_1, \hat{o}_2\}$. The new TCON has four different patterns while the two original TCONs each have two. As will be explained later, merging TCONs is necessary before the placement step.

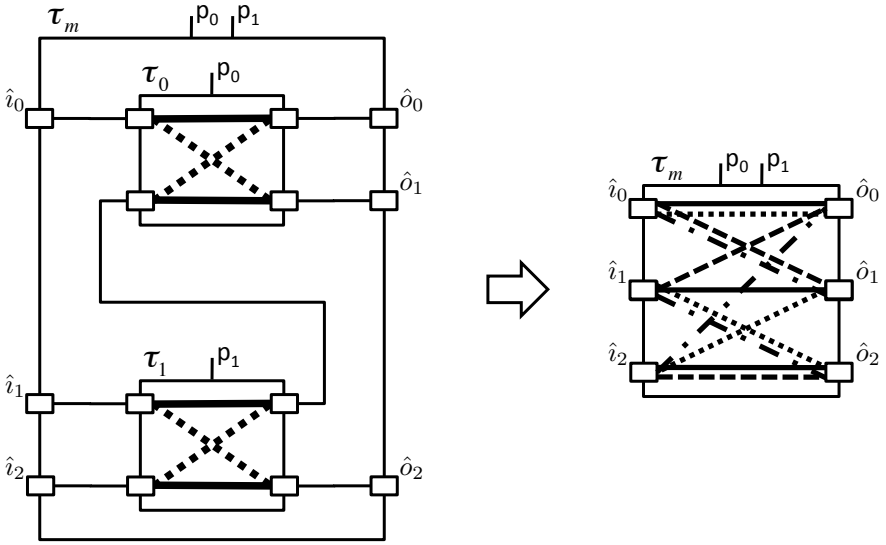


Figure 7.4: Combining two interconnected four-way switches (left) to form a new merged TCON (right).

The connection function of the merged TCON, $\tau_m = (\mathcal{I}, \mathcal{O}, \zeta^{\tau_m})$, can easily be calculated from the connection functions of the original TCONs, $\tau_0 = (\mathcal{I}_0, \mathcal{O}_0, \zeta^{\tau_0})$ and $\tau_1 = (\mathcal{I}_1, \mathcal{O}_1, \zeta^{\tau_1})$, and the way the TCONs are interconnected.

The way the TCONs are interconnected can be expressed as a function ι that maps the sinks that are available inside the merged TCON to the sources that are available in the merged TCON. The sinks set equals the union of the outputs of the merged TCON and the inputs of the original TCONs. The source set equals the union of the inputs of the merged TCON and the outputs of the original TCONs.

$$\iota \quad : \quad \mathcal{O} \cup \mathcal{I}_0 \cup \mathcal{I}_1 \quad \rightarrow \quad \mathcal{I} \cup \mathcal{O}_0 \cup \mathcal{O}_1 \tag{7.9}$$

If two TCONs are connected in this way, it is easy to see that there will never be a short circuit. A short circuit occurs if there would exist a parameter value for which an element of the sink set would be connected to two elements of the source set, but since both the interconnection function ι and the connection functions of the original TCONs, $\zeta_p^{\tau_0}$ and $\zeta_p^{\tau_1}$, are functions and thus map a sink to only one source this is impossible. On the the other hand loops are not

excluded in this way. The designer should make sure that no loops occur. If the TCONs are extracted from a gate-level circuit as is explained in Section 7.2, there will be no loops in the merged TCON by construction, as long as there are no combinational loops in the gate-level circuit.

$$\begin{array}{rcccc} \zeta_p^{\tau_m} & : & P \to \mathcal{O} & \to & \mathcal{I} \\ & & (p,o) & \mapsto & \widehat{\zeta}^{\tau_m}(p,o) \end{array} , \tag{7.10}$$

where $\widehat{\zeta}^{\tau_m}$ is recursively defined as

$$\widehat{\zeta}_p^{\tau_m}(a) = \begin{cases} a & \text{if } a \in \mathcal{I} \\ \widehat{\zeta}_p^{\tau_m}(\iota(a)) & \text{if } a \in \mathcal{O} \cup \mathcal{I}_0 \cup \mathcal{I}_1 \\ \widehat{\zeta}_p^{\tau_m}(\zeta_p^{\tau_0}(a)) & \text{if } a \in \mathcal{O}_0 \\ \widehat{\zeta}_p^{\tau_m}(\zeta_p^{\tau_1}(a)) & \text{if } a \in \mathcal{O}_1 \end{cases} . \tag{7.11}$$

More than two TCONs can be merged by repeatedly applying the method described above.

For the example shown in Figure 7.4, the number of patterns of the merged cone equals the product of the number of patterns of the original TCONs. This is the case because the original TCONs have independent parameter inputs. If the parameter inputs are not independent, the number of patterns in the merged TCON can become a lot smaller. For example, if the parameter inputs p_0 and p_1 are related, the reader can easily see that the number of patterns in the merged TCON becomes two. It is even possible that the number of merged patterns becomes lower than the minimum number of patterns of the original TCONs. In the example shown in Figure 7.5 the merged TCON has only one pattern while both the original TCONs have two patterns.

7.1.3 TCON Implementation

In this section, I describe how a given TCON, $\tau = (\mathcal{I}, \mathcal{O}, \zeta_p)$, may be implemented in the routing infrastructure of the FPGA. In Section 7.4 I describe algorithms that can automatically calculate such an implementation. The implementation of a TCON consists of:

- A subset of the routing resources of the FPGA (wires, pins and switches) represented as a subgraph of the routing-resource graph of the FPGA;

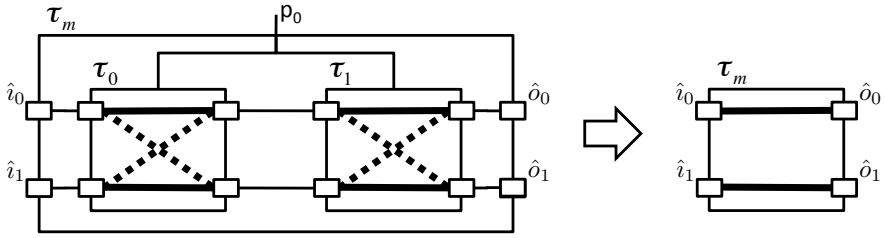


Figure 7.5: Combining two interconnected four-way switches (left) to form a new merged TCON (right). The merged TCON has fewer patterns than the original TCONs.

- For each of the configuration elements that control the selected switches, a Boolean function that expresses the state of that element as a function of the parameter inputs of the TCON.

Figure 7.2 (b) shows an implementation of the four-way switch TCON in a simple 2×2 island-style FPGA. The wires of the routing infrastructure are depicted by the thick lines in between the blocks and the switches are depicted by thin lines that connect the wires and the ports of the blocks. The resources that are used to implement the four-way switch are highlighted. Next to each of the switches used in the TCON implementation, its configuration bit is shown as a function of the parameter input p_0. A switch is assumed to be closed when its configuration bit is high.

Interface Assignment

Before the implementation of a TCON in the routing infrastructure of an FPGA can start, a routing resource (pin or wire) needs to be assigned to each of the inputs and outputs of the TCON. Indeed, it is necessary to know which routing nodes need to be interconnected by the TCON. This task is implicitly performed by the placement step of the tool flow. See Section 7.3 for more information.

In the four-way switch example of Figure 7.2 (b), the two inputs of the TCON are driven by the output pins of two IOBs on the left-hand side of the FPGA architecture, and the outputs drive the input pins of two IOBs on the right-hand side of the FPGA.

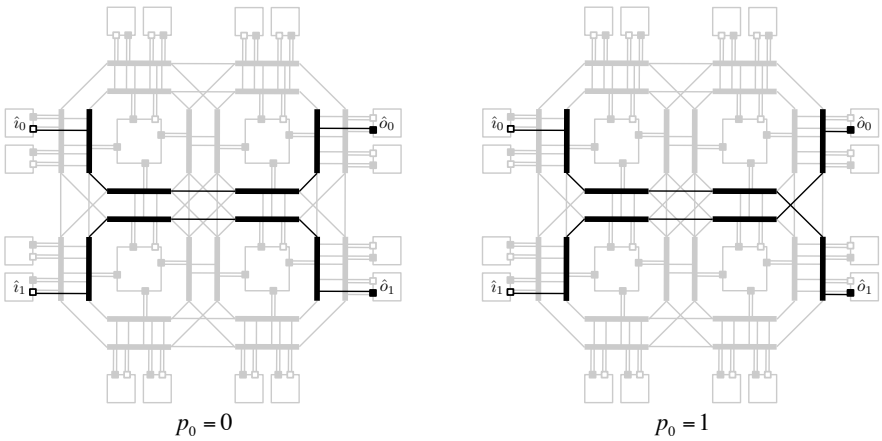


Figure 7.6: Reduced routing graphs for the four-way switch.

Reduced Routing Graph

The reduced routing graph of a TCON, given a parameter value, is a subgraph of the TCON's full routing graph where only the edges for which the tuning function evaluates to one given the parameter value are retained. Figure 7.6 shows the two reduced routing graphs for the four-way switch.

TCON Implementation Validity

A TCON implementation implements a given TCON function under a given interface assignment if and only if:

- for every valid parameter value, the reduced routing graph contains paths for every connection in the connection pattern associated to that same parameter value. A path of a connection is a path of the routing resource graph that connects the resource node assigned to the source of the connection to the resource node assigned to the sink of the connection;
- two paths that are associated to connections that are part of the same pattern and have a different source may never overlap.

The first property makes sure that every connection of every pattern of the TCON can be made with the resources represented by the routing graph. The second property makes sure that connections

that coincide don't use the same resources, thus avoiding short circuits between the input signals of the TCON.

The inverse of the second rule ("Paths that are associated to connections that are not part of the same pattern or have the same source may overlap") is very important. Indeed, this will allow the TCON router to minimize the number of resources needed to implement a TCON.

Figure 7.6 shows the reduced routing graph for each of the valid parameter values of the four-way switch. It is clear that these reduced routing graphs contain paths for each connection in their associated patterns. Note that that the paths that are associated to connections that are part of the same pattern don't overlap, e.g. the paths of connection $(\hat{\imath}_0, \hat{o}_0)$ and connection $(\hat{\imath}_1, \hat{o}_1)$ of pattern π_0 don't have any resources in common. Also, paths that are associated to connections that are part of different patterns do overlap, e.g. the paths of connection $(\hat{\imath}_0, \hat{o}_0)$ of pattern π_0 and connection $(\hat{\imath}_0, \hat{o}_1)$ of pattern π_1 do have resources in common.

7.2 Technology Mapping

During technology mapping the gate-level circuit produced by the synthesis step is mapped onto the resources available in the target FPGA architecture while minimizing a cost function. In conventional technology mapping (see section 2.3) the gate-level circuit is mapped to a LUT circuit where the LUTs have static truth tables. In the TLUT method (see Chapter 4) the truth-table bits of the FPGA were expressed as functions of the parameter inputs. To express this reconfigurability of the LUTs at the level of the mapped circuit, the TLUT was introduced. Now, the routing bits of the FPGA need to be expressed as a function of the parameter inputs. As explained above, this is done by introducing an extra structure at the level of the mapped circuit that expresses the reconfigurability of the interconnect, called a TCON. Just like a TLUT is a generalization of a regular LUT, a TCON is a generalization of a net.

In Section 2.3, I explained that the conventional LUT mapping problem reduces to selecting a set of K-feasible cones that cover the gate-level circuit. Later, in Chapter 5, I extended this to K^*-feasible cones in order to map to TLUTs. Now, I want to map the gate-level circuit to a tunable circuit which is a circuit containing TLUTs and TCONs. Again, it will be necessary to select a set of cones that

cover the gate-level circuit, but now it should be possible to implement these cones either as a TLUT (K^*-feasible cone) or as a TCON ($TCON$-feasible cone). At this point I have not implemented a mapping algorithm that can map to a combination of TLUTs and TCONs. However, in the next section I will give a rough sketch of how such an algorithm might work.

7.2.1 Algorithm Sketch

The TCON mapper again uses the same steps as the mappers described in Section 2.3 and Chapter 5: cone enumeration, cone ranking, cone selection and the generation of the mapping solution. In the following paragraphs I will discuss the changes that are needed in each of these steps.

Cone Enumeration

Just like in the mappers that were discussed above, the TCON mapper will need to enumerate all feasible cones of all nodes n of the input AIG. In contrast to the mapping algorithms discussed so far, a cone can now either be implemented by a K-input TLUT or a TCON. The cones that can be implemented by a TLUT are called K^*-feasible cones and the cones that can be implemented by a TCON are called $TCON$-feasible cones. A $TCON$-feasible cone is a cone of which the Boolean function can be written in the form of Equation (7.7) under the condition given in Equation (7.8). The set of all K^*-feasible cones of node n is denoted as $\Phi(n)$ and the set of all $TCON$-feasible cones of a node n is denoted as $\Theta(n)$. The set of all cones of a node n is denoted $\Upsilon(n) = \Phi(n) \cup \Theta(n)$.

The K^*-feasible cones can easily be enumerated using the method described in Section 5.3. At this point, I don't have a turnkey solution for enumerating all $TCON$-feasible cones, $\Theta(n)$. One possibility would be to use a Boolean matching algorithm [21].

Cone Ranking

The cone ranking step is similar to the one used by conventional mappers. The nodes are processed in topological order from the combinational inputs (CIs) to the combinational outputs (COs). For each visited node n the best non-trivial cone $bc(n)$ is selected from $\Upsilon(n)$ according to the mapping criterion. In depth-oriented mapping the

cone with the lowest depth is selected. If several cones have the same depth the one with the lowest area flow is selected. The difference with the algorithms described above, is that it is necessary to make distinction between K^*-feasible cones and $TCON$-cones. While K^*-feasible cones are implemented by a LUT, $TCON$-cones use only routing resources and thus do not contribute to the total area of the design, which is measured as the total number of LUTs. Moreover, since the depth of a circuit is defined as the number of LUTs in the critical path of the circuit, $TCON$-feasible cones also don't contribute to the depth of a circuit. The adapted equations for depth and area flow of a cone C_n are given by Equation (7.12) and Equation (7.13), respectively. In these equations the cost of a TCON is set to zero, which is of course an underestimate. In the future it might be necessary to develop a better balanced cost function.

$$depth(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ \underset{u \in inode(C_n)}{max}(depth(bc(u))) & \text{if } C_n \in \Theta(n) \\ 1 + \underset{u \in inode(C_n)}{max}(depth(bc(u))) & \text{if } C_n \in \Phi(n) \end{cases} \tag{7.12}$$

$$af(C_n) = \begin{cases} 0 & \text{if } n \in \text{CI} \\ \sum\limits_{u \in inode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{if } C_n \in \Theta^r(n) \\ 1 + \sum\limits_{u \in inode(C_n)} \frac{af(bc(u))}{|oedge(u)|} & \text{if } C_n \in \Phi^r(n) \end{cases} \tag{7.13}$$

Cone Selection

The cone selection step stays unchanged. A subset of the best cones is selected as the final covering of the graph. This is done by traversing the nodes of the graph in topological order, starting from the COs and moving towards the CIs. First the best cones of the nodes driving the COs are selected and then the best cones of the input nodes, $inode(.)$, of the selected cones are added until the CIs are reached.

Generating the Mapping Solution

During this last step, the tunable circuit containing both TLUTs and TCONs is generated together with the truth tables for the TLUTs and

the connection functions for the TCONs. This is done by instantiating a TLUT for each of the K^*-feasible cones in the covering, instantiating a TCON for each of the TCON-cones in the covering and connecting the TLUTs and TCONs in the same way their corresponding cones are connected. The truth tables and the connection functions are found by analyzing the functionality of a K^*-feasible cone or a TCON-feasible cone, respectively. All this information is passed to the placement step.

7.3 Placement

During the placement step each of the TLUTs and other functional blocks that are part of the tunable circuit are assigned to a physical logic block on the target FPGA. Conventionally, wirelength-driven placement makes use of simulated annealing in order to find an acceptable placement for a given static circuit. Section 2.4 describes the well known conventional placement algorithm called VPLACE. As will be explained in this section, this algorithm can almost completely be reused in order to place a tunable circuit. The only thing we need to change is the estimation of the total wirelength, which is used as the cost function in a wirelength-driven placement tool. The only difference between the algorithms lies in the fact that the structures that interconnect the TLUTs are no longer nets but are now TCONs. We thus need to estimate the number of wires segments the router will need to route a TCON given the positions of the TLUTs it connects to and do this in an efficient way.

The approach taken in this thesis is to estimate the wirelength of a TCON in exactly the same way as the wirelength of a net is estimated, i.e. the half perimeter of the bounding box weighted by a factor that depends on the number of terminals of the TCON (See Section 2.4.2). This approach is used in the experiments presented in Section 7.5.

The approach assumes that we know the coordinates of each of the TCON's terminals. Given a placement, only the terminals that connect directly to a logic block are known. This means that our placement approach will not work if TCONs are connected to other TCONs. In this work, we solve this problem by merging interconnected TCONs (Section 7.1.2) until no TCON/TCON connections are left. This could result in very large TCONs, which are difficult to route if logic blocks are interconnected by complex interconnect structures. Another way to solve this problem is assigning the inter-

TCON signals to a wire resource during placement, but this is beyond the scope of this thesis.

7.4 Routing TCONs

In this section, I describe two algorithms that are able to find a solution for the TCON routing problem. Both algorithms are based on the widely used PATHFINDER algorithm [7, 55], which is described in Section 2.5.2. The first algorithm (Section 7.4.2) will be refered to as the *Pattern Router* and was first described in [15]. The second algorithm (Section 7.4.3) will be named *Connection Router* and is only described but not implemented at this point. This algorithm is designed in order to reduce the complexity of the first router at the cost of a possible increase in routing resources.

7.4.1 The TCON Routing Problem

The TCON routing problem is to find a TCON implementation (Section 7.1.3) for each of the TCONs in the input tunable circuit. I assume that there is an interface assignment available for each of the TCONs. This interface assignment is produced by the placement step, which was explained in Section 7.3.

7.4.2 Pattern Router

In this section, I describe a first algorithm that solves the TCON routing problem. I begin by describing a heuristic subroutine that searches a minimum cost routing graph for a given TCON. Each node n in the resource graph has an associated cost c_n. The cost of a routing graph is the sum of the costs of its nodes. Afterwards, I explain how this subroutine may be combined with negotiated congestion in order to find a set of disjoint routing graphs for a given set of TCONs.

Routing one TCON

The pattern router routes a TCON by routing each individual pattern separately. The union of the routing graph of all the patterns is the routing graph of the TCON. Nets that are part of the same routing

```
procedure routeTCON(Tcon τ):
  for each pattern π in τ do:
    for each net η in π do:
      routeNet(η)
      η.resources.setCost(∞)
    τ.resources.setCost(0)
  τ.resources.resetCost()
```

Figure 7.7: Pseudo code for the TCON router.

pattern need to be realized at the same time and thus have to be disjoint (in order to avoid short circuits). However, two nets that are part of different patterns, never need to be realized at the same time and can thus share routing resources. This last property will be used to minimize the routing cost of a TCON by maximizing the overlap among different patterns.

The pseudo code of the proposed heuristic algorithm is shown in Figure 7.7. The algorithm contains two nested for-loops. The outer loop loops over all patterns of the TCON. The inner loop loops over all nets in the current pattern and routes them using a net router. The net router is a heuristic that searches a minimum cost routing tree for a given net. See Section 2.5.2 for more information.

In order to forbid resource sharing for nets within one pattern and allow resource sharing for nets in different patterns the costs of the nodes are manipulated within the TCON router. After routing a net, the cost of the resources used by that net are set to infinity. This way, the next net will avoid resources that are already used by previously routed nets of the current pattern. After routing a pattern, the cost of all the resources that are already used by the current TCON are set to zero. This way, overlap between patterns is stimulated. After routing the full TCON, the resource costs are reset to their original values.

Negotiated Congestion

The pattern router uses negotiated congestion to calculate a set of disjoint routing graphs for a given tunable circuit. It does this in the same way PATHFINDER calculates disjoint routing trees for nets. See

```
while (sharedResourcesExist()):
  for each TCON τ do:
    τ.ripUpRouting()
    routeTCON(τ)
    τ.resources().updateSharingCost()
  allResources().updateHistoryCost()
```

Figure 7.8: Pseudo code for the negotiated congestion loop of the pattern router.

Section 2.5.2 for more information. The pseudo code of the pattern router is shown in Figure 7.8. The algorithm iteratively rips up and reroutes (*routeTcon*) each of the TCONs until their routing graphs are disjoint. Or, in other words, there are no shared resources.

In negotiated congestion, the individual TCON routing problems are coupled by updating the node costs c_n during the routing process. Our algorithm calculates and updates the node cost in exactly the same way as the routability-driven router described in [7], see Section 2.5.2.

Tuning Functions

Given the routing tree of a TCON, the paths associated to each of the connections, and the condition for each of the connections it is easy to find the Boolean function for each of the switches in the routing tree of the TCON. In order to realize a connection, the switches that are part of the path associated to that connection need to be closed. Since connections only need to be realized when their associated pattern condition is true, these switches only need to be closed under that condition. Because paths can overlap, switches can be part of the paths associated to several connections. These switches need to be closed whenever one of the conditions associated to the connections that make use of the switch is true. Or, in other words, the condition under which a switch needs to be closed is the logical OR of all pattern conditions associated to the connections that make use of the switch. The Boolean function for the configuration bit that controls the switch is equal to this condition or its inverse if the control of the switch is active high or active low, respectively.

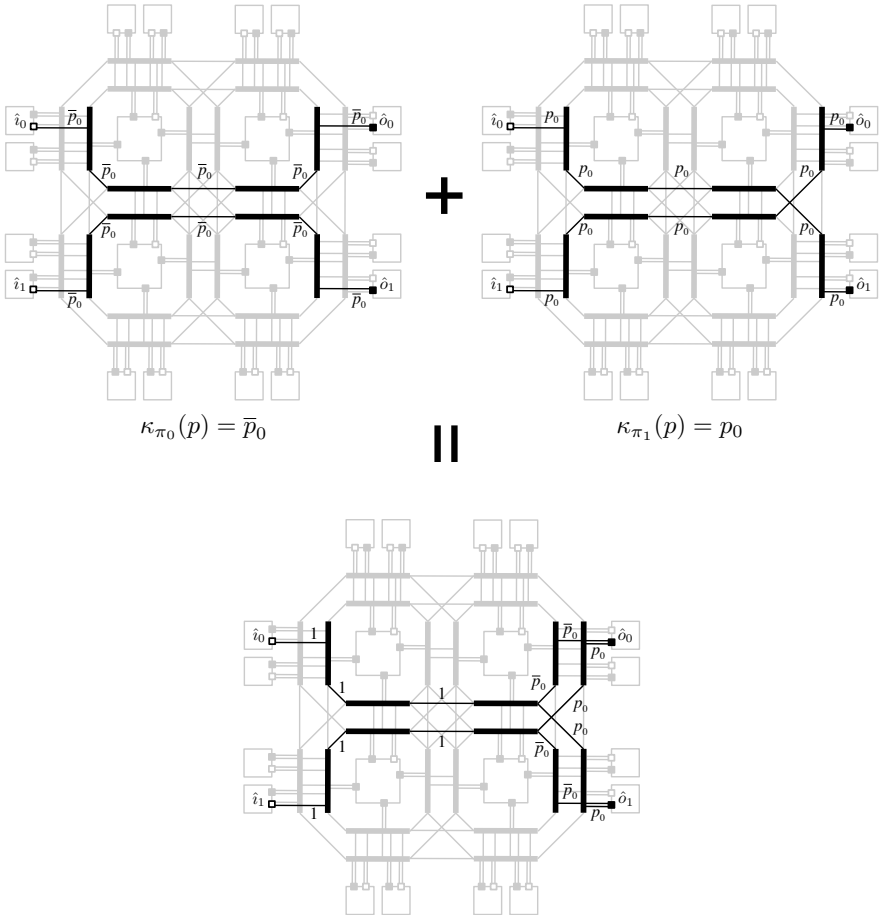


Figure 7.9: The tuning function of a switch is the logic OR of the pattern conditions associated to the paths that use the switch. At the bottom: the final implementation of the four-way switch. The figure shows the routing graph and the tuning function for the switches in that routing graph. At the top: the routing graph for each of the patterns of the four-way switch and their pattern conditions.

Figure 7.9 illustrates how the tuning functions are found for the four-way switch example. The figure on the top left shows the paths associated to the connections of pattern π_0 with condition $\kappa_{\pi_0}(p) = \overline{p_0}$ and the figure on the top right shows the paths associated to the connections of pattern π_1 with condition $\kappa_{\pi_1}(p) = p_0$. The figure at the bottom shows the final implementation (the routing graph and the tuning functions) of the four-way switch TCON assuming the control of the switches is active high.

Complexity

The main problem with the pattern router as described above is the high complexity of the TCON router. As a measure of that complexity I take the number of times the Dijkstra algorithm needs to be invoked in order to route one TCON. In worst case, this number is equal to

$$|\Pi| = |\mathcal{O}||\mathcal{I}|^{|\mathcal{O}|} \tag{7.14}$$

where $|\mathcal{I}|$ is the number of inputs of the TCON and $|\mathcal{O}|$ is the number of outputs of the TCON. Indeed, there are $|\mathcal{I}|^{|\mathcal{O}|}$ different patterns (or ways to choose an input for each of the outputs) and each of these patterns contain $|\mathcal{O}|$ connections that need to be routed using Dijkstra.

For large TCONs the number of Dijkstra invocations can become very high, which will result in large runtimes for the pattern router. The connection router, which is described next, tries to address this problem.

7.4.3 Connection Router

The connection router only differs from the pattern router in the way the TCON implementation is generated. The negotiated congestion loop is not changed.

TCON Representation

In this new algorithm a TCON, $\tau = (\mathcal{I}, \mathcal{O}, \zeta_p)$, is represented as a *Connections Set* where each connection is associated with a connection condition. The connection set of τ can be derived from its connection function as follows

$$\Gamma^\tau = \{\gamma = (so, si) \in \Gamma : \exists p \in P'(\zeta_p(si) = so)\}, \tag{7.15}$$

where $\Gamma = \mathcal{I} \times \mathcal{O}$ is the set of all possible connections in a TCON. The connection condition of a given connection $\gamma = (so, si)$ is a Boolean function of the parameters that is true for those parameter values that require the connection to be made. This function is equal to the Boolean connection function $\widetilde{\zeta}^{\tau}_{si,so}(p)$, as defined in Equation (7.6).

If the TCON is routed by invoking Dijkstra once for every connection, the worst case number of Dijkstra invocations will decrease to only

$$|\Gamma| = |\mathcal{O}||\mathcal{I}|, \tag{7.16}$$

because in worst case there are only that many possible ways to connect an input of a TCON to one of its outputs.

Overlap Mechanism

Since in the pattern router, the different patterns of the TCON were routed one by one, it was easy implement the overlap mechanism (Section 7.1.3) since this is naturally expressed in terms of patterns. When we represent a TCON as described in the previous section, connections can belong to several patterns at the same time. This makes it extremely difficult to preserve the full overlap mechanism during the routing process without sacrificing a large part of the speed gain made by switching from the pattern representation to the connection representation. Therefore I will now propose a simplified version of the overlap mechanism that naturally fits the connection router and thus leads to a fast algorithm. As will be explained, the disadvantage of the simplified algorithm is the fact that not all possible overlap will be found by the connection router, which can reduce the quality of the routing result.

The adapted overlap mechanism only allows overlap between a set of connections that have the same source or a set of connections that have the same sink. A set of connections that have the same source can overlap because they always carry the same signal and therefore there is never any danger of a short circuit. In a legal TCON, a set of connections that have the same sink will never be realized at the same time because this would mean a short circuit.

The Algorithm

As was already mentioned, the connection router only differs in the way the implementation of a TCON is generated. The negotiated

congestion loop remains unchanged. The pseudo code of the new TCON router is shown in Figure 7.10. As can be seen, the TCON router now loops over all connections in the TCON an determines the cheapest path for these connections using Dijkstra's algorithm [33].

Like in the pattern router, the overlap mechanism is implemented by manipulating the cost of the nodes that will be used by the Dijkstra algorithm. The cost function used by the Dijkstra algorithm depends on the paths of the already routed connections of the TCON. In order to clearly explain the cost calculation, I will first introduce pin equivalence. If a node is used by a set of connections that either have the same source or the same sink, that node is said to be equivalent to that common pin. Indeed, given the overlap mechanism described above, that node will either carry the same signal as the common pin or no signal at all. If a connection needs to be routed towards a certain sink pin, it can be routed up to an equivalent pin of that sink and from there on the already existing path to the sink can be used. In an analog way, connections that need to be routed from a certain source can take an already existing path from the source to one of its equivalent nodes and from there it can be further routed towards its sink. If a node is not used yet it is not equivalent to any pin and if a node is used by only one connection, it is equivalent to both the connection's source and the connection's sink. If the set of pins that node n is equivalent to is denoted $E(n)$ and the set of nodes used in the current TCON is denoted U, the cost of a node can be calculated as follows.

$$cost(n) = \begin{cases} c(n) & \text{if } n \notin U \\ 0 & \text{else if } so \in E(n) \vee si \in E(n) \\ \infty & \text{otherwise} \end{cases} . \tag{7.17}$$

As can be seen in the pseudo code, after the cheapest path for the connection is found, the equivalency set of each of the nodes in that path is changed. If a node was not used yet in the TCON, the node is set to be equivalent to both the source and the sink of the connection. If the node was used before, the equivalency set of the node is set to the intersection of the previous equivalency set and the pin set of the connection.

```
procedure routeTCON(Tcon τ):
  for each connection γ = (so, si) in Γ^τ do:
    path = dijkstra (so,si)
    for each node n in path do:
      if n.notUsed():
        n.setEquivalentSet({so, si})
        n.setUsed()
      else:
        n.setEquivalentSet(n.equivalentSet() ∩ {so, si})
```

Figure 7.10: Pseudo code for the TCON connection router.

Tuning Functions

The tuning functions can be found in almost the same way as for the pattern router. The only difference is that the connection router works with connections and connection conditions instead of patterns and pattern conditions. In short, the condition under which a switch needs to be closed is the logical or of all connection conditions associated to the connections that make use of the switch.

Quality/run-time Tradeoff

The simplification of the overlap mechanism will drastically reduce the run time of the routing algorithm, but it will of course in some cases lead to a reduction of the routing quality. By quality I mean the number of resources needed to route a TCON. Since the algorithm is not implemented at this point, it is not possible to quantify this reduction, but I believe that loss of quality will be limited.

For the four-way switch example, the connection router will find the same result as the pattern router would. The four-way switch TCON has four connections $(\hat{\imath}_0, \hat{o}_0)$, $(\hat{\imath}_1, \hat{o}_1)$, $(\hat{\imath}_0, \hat{o}_1)$ and $(\hat{\imath}_1, \hat{o}_0)$ with connection conditions p_0, p_0, $\overline{p}_0$ and $\overline{p}_0$, respectively. If the pattern router's implementation of the four-way switch is examined, no overlap is used that could not be found by the connection router. Indeed, the paths of $(\hat{\imath}_0, \hat{o}_0)$ and $(\hat{\imath}_0, \hat{o}_1)$ overlap, but this overlap could also be found by the connection router because these two connections have a common source. The same thing can be said about connection $(\hat{\imath}_1, \hat{o}_0)$ and $(\hat{\imath}_1, \hat{o}_1)$.

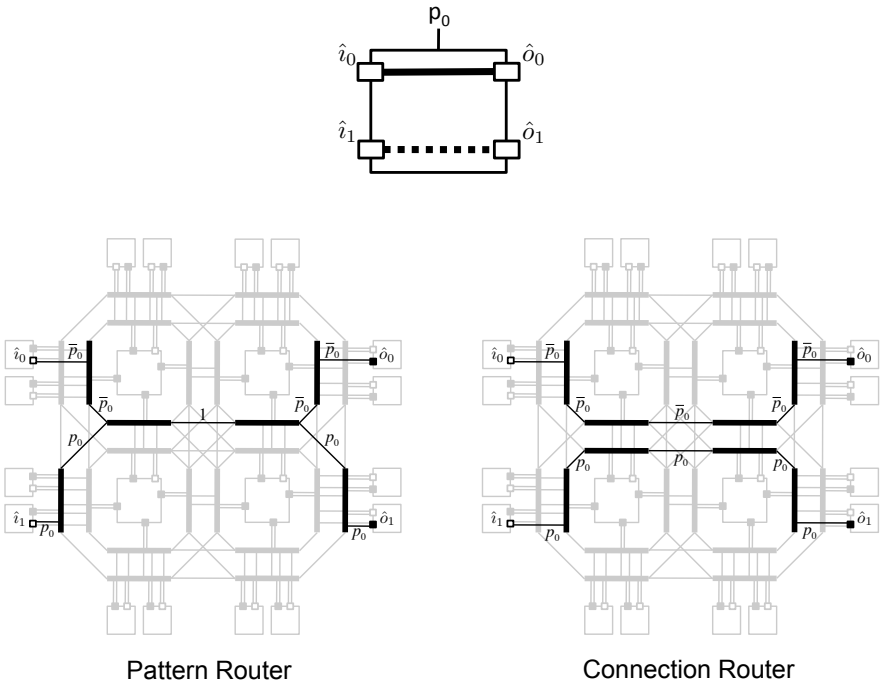


Figure 7.11: Illustration of the reduction in overlap when comparing the pattern router (left) and the connection router (right).

However, it is possible to find examples of TCONs that are routed more efficiently with the pattern router than with the connection router. An example is shown in Figure 7.11. The TCON shown has two connection $(\hat{\imath}_0, \hat{o}_0)$ and $(\hat{\imath}_1, \hat{o}_1)$ which are mutually exclusive. The pattern router will thus be able to overlap their paths, but the connection router will not because the two connections have different sinks and different sources.

7.5 Experiments and Results

The pattern router (Section 7.4.2), which is called TROUTE, was implemented based on my own Java version of the VPR (Versatile Place and Route) [7] routability-driven router. I use a simple FPGA archi-

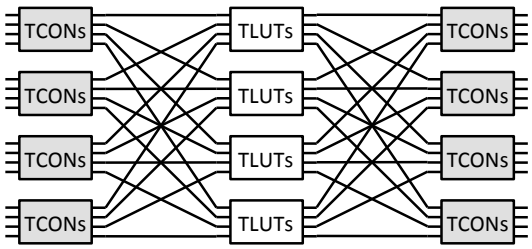


Figure 7.12: *Tcon* implementation of the 16×16 Clos network.

tecture[1] with logic blocks containing one 4-LUT and one flip-flop. The wire segments in the interconnection network only span one logic block. The architecture is specified by three parameters: the number of logic element columns ($cols$), the number of logic element rows ($rows$) and the number of wires in a routing channel (W).

I validate TROUTE on Multistage Interconnect Networks that are known as Clos Networks [25]. Our Clos network uses 4×4 crossbar switches as building blocks. I use 4×4 switches because these can be efficiently implemented using four 4-input TLUTs or four TCONs. I compare three network types, called *Conv*, *Tlut* and *Tcon*, each for three sizes: 16×16 (3 stages), 64×64 (5 stages) and 256×256 (7 stages). The network *Conv* uses signals to control the crossbar switches while *Tlut* and *Tcon* use reconfiguration. The network *Tlut* only uses reconfiguration of LUT truth tables while *Tcon* uses both reconfiguration of LUTs and reconfiguration of routing. In *Tlut* all the switches are implemented with 4 TLUTs while in *Tcon* the switches in the even stages are implemented using TLUTs and the switches in the odd stages are implemented using TCONs. This lasts implementation results in a good balance between TLUTs and TCONs. Also not that since the pattern router is used, the number of connections that need to be routed will grow very large if the complete Clos switch would be implemented with only one TCON. This problem will not occur when the connection router is used. Figure 7.12 shows the *Tcon* implementation of the 16×16 Clos network.

These implementations are comparable to the reconfigurable switches described in [76, 49] except that their design is done manually at the architectural level while ours is done automatically at

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

the more abstract level of a tunable circuit, thus greatly reducing the design effort.

I implemented the nine networks and measured (i) the number of LUTs, (ii) the number of wires, (iii) the logic depth, (iv) the routing time and (v) the minimum channel width (W_m). Table 7.1 shows the results. The table also shows the parameters of the FPGA architecture. As suggested in [7], low-stress place and route is ensured by choosing the number of LUTs in the FPGA architecture 20% larger than the number of LUTs in the circuit and the number of wires per channel 20% larger than W_m, the minimum channel width.

The wire utilization of the implementations will be influenced by the placement of the inputs of the network. If the inputs and outputs are placed far apart, more wires will be needed than when they are placed closely together. To normalize this influence each input and output is connected to a LUT that is connected to no other signals. This way the placer is free to place the inputs and outputs to minimize the number of wire resources. These extra LUTs are not accounted for in the LUT count of Table 7.1, because they are not part of the actual Clos network.

The routing of the *Conv* and *Tlut* implementations is done with the VPR routability-driven router. Their placement is done using the VPR routability-driven placer with default settings. The routing of the *Tcon* implementations is done using the pattern router as explained in Section 7.4.2. The placement is done using an adapted version of the VPR routability-driven placer, as explained in Section 7.3.

The *Tcon* networks save up to a factor 12.63 in the number of LUTs compared to the *Conv* networks and up to a factor of 3 compared to the *Tlut* networks. The number of LUTs in the longest path (logic depth) is used as a measure for the clock speed. When using the *Tcon* implementation, the logic depth can be reduce up to a factor of 5 compared to the *Conv* implementation and a factor of 3 compared to the *Tlut* implementation.

The table also shows that up to a factor 4.84 can be saved in the number of wires compared to the *Conv* networks and up to 20% of the wires can be saved compared to the *Tlut* networks. This last result might be counterintuitive since TCONs are more complex to route than nets. However, switching from *Conv* to *Tlut* to *Tcon* decreases the number of nets/TCONs and the number of LUTs. Fewer nets/TCONs connecting fewer LUTs that can be placed more closely together thus results in fewer wires used. On the other hand, because

the LUTs get placed more closely together, W_m goes up, but it stays far below the channel widths available in commercial FPGAs.

Table 7.1 also shows the routing time needed for each implementation. All these experiments are done using an Intel Core 2 processor running at 2.13 GHz with 2 GiB of memory running the Java HotSpot™ 64-Bit Server VM. Using the *Tcon* networks the routing time is reduced with a factor of 10.61 up to 13.24 compared to the *Conv* networks and 4% to 40% of the routing time can be saved compared to the *Tlut* networks. This gain in routing time is due to the decrease in routing complexity as is explained in the previous paragraph.

7.6 Conclusions and Future Work

In this chapter, I have made the first steps towards the extension of the TLUT method, described in Chapter 4, so that not only the LUT truth tables can be expressed as a function of the parameter inputs but also the configuration bits that control the routing of the FPGA. To this extent, the concept of a TCON was introduced. A TCON is an abstraction of a subset of an FPGA's routing resources, which reflects the reconfigurability of those routing resources.

The new TCON method requires changes in technology mapping, placement and routing. I described algorithms for each of these steps, but focused on the routing step, for which a first algorithm was implemented. This pattern router algorithm was used to automatically implement large reconfigurable switches, showing promising results.

This chapter leaves lots of open questions, implementation work and experimenting. In my opinion, the technology mapping step and the closely related synthesis step leave the biggest challenge. How can TCON cones be identified in an efficient way? Can changes to the synthesis step lead to more efficient mapping? Conventional synthesis tools are built to minimize the number of gates in the hope that this will also result in a low number of LUTs after technology mapping. However, it has been shown that this is not always the case [22]. This problem is known as structural bias. I believe that the problem of structural bias will be even more prominent when mapping to a mix of TLUTs and TCONs, because of the discrepancy in cost between a TLUT and a TCON. The challenge here would be to

adapt the cost mechanism of a synthesis tool in such a way that this discrepancy is taken into account.

Furthermore, the connection router and the placement tool needs to be implemented and thoroughly evaluated. At this point, all the described algorithms optimize for wire-length, while commercial tools optimize for timing. Further research should be done on how to extend the timing model currently used by commercial tools to support TCONs.

Table 7.1: Properties of nine multi stage Clos network implementations. The numbers between brackets are relative compared to the *Tcon* implementation of the same size.

Impl.		Area				Speed		Routing			Architecture		
size	type	LUTs		wires		logic depth		$t_{route}[s]$		W_m	$cols$	$rows$	W
	Conv	202	(12.63)	2131	(4.84)	5	(5.00)	7.96	(10.61)	6	20	20	7
16	*Tlut*	48	(3.00)	526	(1.20)	3	(3.00)	1.06	(1.41)	4	10	10	5
	Tcon	16	(1.00)	440	(1.00)	1	(1.00)	0.75	(1.00)	7	8	8	8
	Conv	1016	(7.94)	13613	(3.91)	9	(4.50)	294.73	(12.44)	6	47	47	7
64	*Tlut*	320	(2.50)	3511	(1.01)	5	(2.50)	24.71	(1.04)	8	23	23	10
	Tcon	128	(1.00)	3483	(1.00)	2	(1.00)	23.69	(1.00)	14	18	18	17
	Conv	6760	(8.80)	97994	(3.94)	12	(4.00)	15415.51	(13.24)	9	114	114	11
256	*Tlut*	1792	(2.33)	25353	(1.02)	7	(2.33)	1234.66	(1.06)	13	53	53	16
	Tcon	768	(1.00)	24851	(1.00)	3	(1.00)	1164.59	(1.00)	20	39	39	24

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Dynamic Circuit Specialization is an important technique in the field of Dynamic FPGA reconfiguration. The technique is orthogonal to the better known technique of configuration swapping. In this work, I showed how the development of Dynamic Circuit Specialization systems can be automated. Previously, DCS systems needed to be hand-crafted at the architectural level, which made the development of such systems very expensive.

In order to do this, I introduced the concept of a parameterized configuration and developed methods which can automatically generate such a configuration starting from a parameterized RT-level design. This is a synthesizable HDL design where some of the inputs are designated as parameters. This way, the effort for implementing DCS applications is reduced to the effort needed for implementing a generic implementation with the same functionality. However, our automatic DCS implementation method is able to find a solution that is significantly more resource efficient and that can run at higher clock speeds.

The key concept in this thesis is a parameterized configuration. It is a multivalued Boolean function that expresses the configuration memory of an FPGA as a function of the parameters. The DCS systems realized in this thesis use a parameterized configuration to dynamically specialize the configuration of an FPGA for changing parameter values.

TLUT method

The first method I described is called the TLUT method. This method generates parameterized configurations in which only the truth table bits of the FPGA are expressed as a function of the parameters. All other configuration bits are static. I explained that only reconfiguring the LUTs can lead to very fast reconfiguration, but on the other hand might not lead to the most compact implementation.

The TLUT method is implemented as an adapted version of a conventional FPGA tool flow, where mainly the technology mapping step is changed. The new technology mapping algorithm I developed is called TMAP. TMAP maps a gate-level circuit to Tunable LUTs (TLUTs). Instead of a static truth table, these LUTs have a truth table that is expressed as a Boolean function of the parameters. I showed that TMAP scales equally well as conventional LUT mappers and that the complexity of evaluating the truth tables of the TLUTs scales favorably with the size of the input gate-level circuit.

TCON method

The second method is called the TCON method. The TCON method extends the TLUT method so that the routing configuration bits are also expressed as a function of the parameters. For certain applications, also allowing reconfiguration of interconnect can lead to further reduction of the needed FPGA area, increased performance and reduced power consumption. The disadvantage is that the reconfiguration time may increase because more bits need to be reconfigured and, unlike the truth table bits, the routing bits are typically scattered over the configuration memory space.

The TCON method requires drastic changes to technology mapping, placement and routing. The fundamental difference in the TCON method is the replacement of the net, an interconnection with one source and multiple sinks, by the TCON. A TCON has a number of sources and a number of sinks and its parameter inputs control how these sinks and sources are interconnected. The technology mapper now needs to map to a mixture of TLUTs and TCONs and the router will now need to route TCONs instead of nets. Not all these changes are completed at this point. This work contains an algorithm sketch of a TCON technology mapper and a fully implemented TCON router, called the pattern router. Apart from the

pattern router there is also a description of an other better scalable TCON router, called the connection router.

DCS Systems

Besides describing how parameterized configurations can be generated I have also used them to implement DCS systems and showed that such systems can use FPGA resources more efficiently as long as the parameters don't change too often.

For the parameterized configurations generated by the TLUT method, I have built DCS systems (adaptive FIR filters, TCAMs) that run on commercial FPGAs, more specifically the Xilinx Virtex-II Pro. Therefore, I developed a way to incorporate TMAP in the Xilinx tool flow. I showed large reductions in the number of LUTs (39% for the FIR filters (128 taps with 8-bit coefficients) and 66% for the TCAMs (256 32-bit wide entries)) and significant improvements of the maximum clock frequency (38% for the FIR filters and 30% for the TCAMs). The specialization of both designs was done using the embedded PowerPC and the ICAP of the Virtex-II Pro. The total time needed to change the coefficients of the FIR filter is 1.74 ms and the content of the complete TCAM can be rewritten in 1.72 ms using ICAP reconfiguration. I also explored SRL reconfiguration which is a form of reconfiguration that naturally fits with the TLUT method and enables very fast reconfiguration. For SRL reconfigurations I showed speedups in reconfiguration time exceeding more than two orders of magnitude compared to ICAP reconfigurations.

Building a DCS system that makes use of the TCON method for a commercial FPGA is very difficult since FPGA vendors don't publish detailed descriptions of the routing architectures of their FPGAs. Nevertheless, my first experiments on large Clos networks showed promising results. For a 256×256 Clos network, I showed reductions with a factor of 8.8 and 3.9 in the number of LUTs and the number of wires, respectively.

8.2 Future Work

As the new concept of parameterized reconfiguration has brought an entirely new sight on FPGA reconfiguration, I believe that the work and research opportunities of parameterized configurations are far

from exhausted. Most of them I can probably not even imagine at this point. A few possibilities for further research are listed hereafter.

8.2.1 Generating Parameterized Configurations

Optimized Algorithms

The tools described in this work are all extensions of the very basic algorithms available for technology mapping, placement and routing. I started from these algorithms in order not to overcomplicate things and keep the focus on the new concepts. However, these algorithms have all been optimized: place and route tools nowadays are timing-driven instead of wire-length driven and technology mappers are optimized for larger LUT sizes (priority-cuts [56]) and to reduce structural bias (AIG with choices [21]). From my experience with the current algorithms, I see no reason why these optimizations could not be applied to the algorithms described in this thesis, but still they need to be applied and validated.

TCON method

As was mentioned several times, only the pattern router of the TCON method is fully implemented. In Chapter 7, I made the first steps towards the implementation of a new technology mapper and I described an optimized TCON router, called the connection router. Both these algorithms need to be implemented and thoroughly validated.

Representation of Parameterized Configurations.

At this point, a parameterized configuration is represented as a completely flattened Boolean network. This leads to a large size of the parameterized configuration and a poor evaluation time. As was already mentioned, the hierarchy and the repetitive nature that is present in many applications could be used to compact the representation of the parameterized configuration.

Besides the hierarchy, a parameterized configuration could be further compacted by not only allowing Boolean operations but also introducing RT-level operations in the representation. These operations are present in the HDL design made by the designer, so they

could well be used efficiently. Of course this will require that the synthesis tool does not output a completely flattened gate-level circuit, but preserves RT-level operations and that the technology mapper can handle these operations while mapping to TLUTs. Many synthesis tools already preserve RT-level operations, so the main problem lies with extending the technology mapper.

Allowing RT-level operations in a parameterized configuration will also reduce its evaluation time on a standard instruction set processor. Indeed, one RT-level operation can be executed in one instruction while the same operation replaces many logic level operations.

Synthesis

The synthesis tool used in this work was only slightly adapted so that it passes the information on which inputs are parameter inputs. Conventional synthesis tools are built to minimize the number of gates in the hope that this will also result in a low number of LUTs after technology mapping. However, it has been shown that this is not always the case [22]. This problem is known as structural bias. I believe that the problem of structural bias is even more prominent when mapping to TLUTs and certainly when mapping to a mixture of TLUTs and TCONs. A gate that is only dependent on parameter inputs will end up in the parameterized configuration and will be executed on for example an ISP while other gates are implemented by a LUT on the FPGA. However, in conventional synthesis tools these gates are given the same cost. A similar discrepancy exists for gates that are implemented by TLUTs and those implemented by TCONs. In the future, the cost mechanism of the synthesis tool could be adapted so that the discrepancy is better taken into account.

8.2.2 Using Parameterized Configurations

I believe that, although this work focused on DCS, the applicability of parameterized configurations is not limited to DCS. In this light, the techniques described in this thesis can be seen as a way to efficiently generate groups of FPGA configurations where each group member is specialized for a specific parameter value.

Customized Configurations

System manufacturers that make use of FPGAs can use parameterized configurations to customize their products before they are shipped. This could for example be used to embed a unique encryption or license key in the configuration. Previously, the manufacturers were required to build generic implementations that kept the key in memory resulting in larger and slower configurations or they could run a conventional FPGA tool flow for each individual product which is very computationally expensive. In a similar way, products could be calibrated using parameterisable configurations.

Logic Emulation

Parameterized configurations could also be used in the field of logic emulation. They could for example be used to introduce stuck-at faults in a design and thus speed up test pattern generation. This can be done by simply taking the gate-level circuit that needs to be emulated and introducing a multiplexer on each signal. The multiplexer either passes the signal or forces it to 0 or 1 depending on its control inputs. If we choose these control signals as parameters and pass the new circuit through one of my tool flows, the result is a parameterized configuration that makes it possible to rapidly generate configurations with stuck-at faults at any of the signals of the original gate-level circuit. Also not that adding the multiplexers in the gate-level circuit will not increase the size of the final configurations since they can be fully absorbed by the TLUTs. The result is similar to what is described in [16].

By adding extra flexibility to a design that needs to be debugged, parameterized configurations could shorten the design loop. When a design is built by interconnecting several IP blocks, a common mistake is the miss polarization of a reset signal. Although this looks like a minor mistake, it requires a full remapping of the design to the emulation system, which can take hours and even days for large designs. This problem could be solved if the reset input of every block would be accommodated with a multiplexer that allows to either connect the reset port to the inverted or the non-inverted reset signal depending on the control signal of the multiplexer. These control signals are again chosen as parameter inputs. Next, the design is mapped to the logic emulation system with one of the tool flows described in this thesis which results in a parameterized configuration.

If now a miss-polarization of a reset signal occurs, a configuration with the correct polarization can almost instantly be generated by evaluating the parameterized configuration.

Bibliography

[1] F. Abouelella, K. Bruneel, and D. Stroobandt. Efficiently generating FPGA configurations through a stack machine. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, Milano, 2010.

[2] E. Ahmed and J. Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(3):288 – 298, 2004.

[3] Altera. Altera unveils innovations for 28-nm fpgas. Press Releas, Feb. 2010.

[4] S. Bayar and A. Yurdakul. Self-reconfiguration on spartan-iii fpgas with compressed partial bitstreams via a parallel configuration access port (cpcap) core. In *Research in Microelectronics and Electronics, 2008. PRIME 2008. Ph.D.*, pages 137 –140, 222008-april25 2008.

[5] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*.

[6] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, London, UK, 1997. Springer-Verlag.

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

[8] A. Biere. *The AIGER And-Inverter Graph (AIG) Format*. Johannes Kepler University, 2007.

[9] B. Blodget, P. James-Roxby, E. Kelle, S. McMillan, and P. Sundararajan. A selfreconfiguring platform. *International Conference on Field-Programmable Logic and Applications*, pages 565– 574, 2003.

[10] J.-L. Brelet and B. New. *XAPP203: Designing Flexible, Fast CAMs with Virtex Family FPGAs*. Xilinx, 1999.

[11] K. Bruneel, F. Abouelella, and D. Stroobandt. Automatically mapping applications to a self-reconfiguring platform. In K. Preas, editor, *Proceedings of Design, Automation and Test in Europe*, pages 964–969, Nice, 4 2009.

[12] K. Bruneel, P. Bertels, and D. Stroobandt. A method for fast hardware specialization at run-time. In K. Bertels and W. Najjar, editors, *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications*, pages 35–40, Amsterdam, 8 2007.

[13] K. Bruneel and D. Stroobandt. Automatic generation of run-time parameterizable configurations. In U. Kebschull, M. Platzner, and T. J., editors, *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications*, pages 361–366, Heidelberg, 9 2008. Kirchhoff Institute for Physics.

[14] K. Bruneel and D. Stroobandt. Reconfigurability-aware structural mapping for LUT-based FPGAs. In *2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 223–8, Piscataway, NJ, USA, 2008 2008. IEEE. 2008 International Conference on Reconfigurable Computing and FPGAs (ReConFig), 3-5 December 2008, Cancun, Mexico.

[15] K. Bruneel and D. Stroobandt. TROUTE: a reconfigurability-aware FPGA router. In *Lecture Notes in Computer Science*, volume 5992, pages 207–218, Berlin, Germany, 2010. Springer Verlag Berlin.

[16] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, and O. Lepape. Serial fault emulation. In *Proceedings of the 33rd Design Automation Conference*, 1996.

[17] 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.

[18] D. Chang and M. Marek-Sadowska. Partitioning sequential circuits on dynamically reconfigurable fpgas. *Computers, IEEE Transactions on*, 48(6):565 –578, June 1999.

[19] Y.-W. Chang, D. F. Wong, and C. K. Wong. Universal switch modules for fpga design. *ACM Trans. Des. Autom. Electron. Syst.*, 1:80–101, January 1996.

[20] K. Chapman. Fast integer multipliers fit in FPGAs. *EDN*, 39(10):80, May 1993.

[21] S. Chatterjee. *On algorithms for technology mapping*. PhD thesis, Berkeley, CA, USA, 2007. AAI3306090.

[22] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. Reducing structural bias in technology mapping. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ICCAD '05, pages 519–526, Washington, DC, USA, 2005. IEEE Computer Society.

[23] C.-L. E. Cheng. Risa: accurate and efficient placement routability modeling. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 690–695, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[24] S. Churcher, T. Kean, and B. Wilkie. The xc6200 fastmap processor interface. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 36–43. Springer Berlin / Heidelberg, 1995.

[25] C. Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, XXXII:406–424, 1953.

[26] 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*, 13:1–12, 1994.

[27] M. Cummings and S. Haruyama. FPGA in the software radio. *IEEE COMMUNICATIONS MAGAZINE*, 37(2):108–112, FEB 1999.

[28] F. de Dinechein and V. Lefèvre. Constant multipliers for FPGAs. Technical report, École Normale Supśerieure de Lyon, 2000.

[29] A. DeHon. Dpga utilization and application. In *Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays*, FPGA '96, pages 115–121, New York, NY, USA, 1996. ACM.

[30] A. DeHon. DPGA utilization and application. In *International Symposium on Field Programmable Gate Arrays*, pages 115–121, 1996.

[31] A. M. Dehon. *Reconfigurable architectures for general-purpose computing*. PhD thesis, 1996. AAI0597715.

[32] A. Derbyshire, T. Becker, and W. Luk. Incremental elaboration for run-time reconfigurable hardware designs. In *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 93–102, New York, NY, USA, 2006. ACM.

[33] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[34] P. Foulk. Data-folding in SRAM configurable FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 163 –171, 5-7 1993.

[35] A. Gerstlauer, C. Haubelt, A. Pimentel, T. Stefanov, D. Gajski, and J. Teich. Electronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

[36] S. Hauck and A. Dehon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, November 2007.

[37] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[38] M. Huang, F. Romeo, and A. Sangiovanni-Vincentelli. An efficient general cooling schedule for simulated annealing. In *ICCAD*, pages 381–384, 1986.

[39] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

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

[41] P. J. Koopman, Jr. *Stack computers: the new wave*. Halsted Press, New York, NY, USA, 1989.

[42] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[43] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, February 2007.

[44] C. Labovitz, G. Malan, and F. Jahanian. Internet routing instability. *IEEE/ACM Transactions on Networking*, 6(5):515 –528, Oct. 1998.

[45] J. Lam and J.-M. Delosme. Performance of a new annealing schedule. In *DAC '88: Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 306–311, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[46] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, 10:346–365, 1961.

[47] G. G. F. Lemieux, S. D. Brown, and D. Vranesic. On two-step routing for fpgas. In *Proceedings of the 1997 international symposium on Physical design*, ISPD '97, pages 60–66, New York, NY, USA, 1997. ACM.

[48] E. Lemoine and D. Merceron. Run time reconfiguration of FPGA for scanning genomic databases. In *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[49] P. Lysaght and D. Levi. Of gates and wires. *Parallel and Distributed Processing Symposium, International*, 4:132a, 2004.

[50] R. Lysecky, G. Stitt, and F. Vahid. WARP processors. *Transactions on Design Automation of Electronic Systems*, 11(3):659–681, July 2006.

[51] 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.

[52] V. Manohararajah, S. Brown, and Z. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(11):2331 –2340, Nov. 2006.

[53] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for fpgas. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 203–213, New York, NY, USA, 2000. ACM.

[54] G. McGregor and P. Lysaght. Self controlling dynamic reconfiguration: A case study. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 144–154, London, UK, 1999. Springer-Verlag.

[55] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *FPGA*, pages 111–117, 1995.

[56] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. Combinational and sequential mapping with priority cuts. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, ICCAD '07, pages 354–361, Piscataway, NJ, USA, 2007. IEEE Press.

[57] R. Nair. A simple yet effective technique for global wiring. *IEEE Transactions on Computer-aided Design*, 6:165–172, 1987.

[58] J.-B. Note and E. Rannaud. From the bitstream to the netlist. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA '08, pages 264–264, New York, NY, USA, 2008. ACM.

[59] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, 2006.

[60] P. Pan and C.-C. Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, pages 35–42, New York, NY, USA, 1998. ACM.

[61] J. Rose and S. Brown. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE JOURNAL OF SOLID STATE CIRCUITS*, 26(3):277–282, 1991.

[62] J. Rose, R. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *Solid-State Circuits, IEEE Journal of*, 25(5):1217 –1225, Oct. 1990.

[63] Y. Sankar and J. Rose. Trading quality for compile time: ultra-fast placement for FPGAs. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, pages 157–166, New York, NY, USA, 1999. ACM.

[64] K. Shahookar and P. Mazumder. Vlsi cell placement techniques. *ACM Comput. Surv.*, 23:143–220, June 1991.

[65] H. Singh, G. Lu, E. Filho, R. Maestre, M.-H. Lee, F. Kurdahi, and N. Bagherzadeh. Morphosys: case study of a reconfigurable computing system targeting multimedia applications. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 573–578, New York, NY, USA, 2000. ACM.

[66] J. S. Swartz, V. Betz, and J. Rose. A fast routability-driven router for fpgas. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 140–149, New York, NY, USA, 1998. ACM.

[67] W. Swartz and C. Sechen. New algorithms for the placement and routing of macro cells. In *ICCAD*, pages 336–339, 1990.

[68] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *FPGAs for Custom Computing Machines,*

1997. Proceedings., The 5th Annual IEEE Symposium on, pages 22 –28, Apr. 1997.

[69] B. Tseng, J. Rose, and S. D. Brown. Improving fpga routing architectures using architecture and cad interactions. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, ICCD '92, pages 99–104, Washington, DC, USA, 1992. IEEE Computer Society.

[70] University of California Berkeley. *Berkeley Logic Interchange Format (BLIF)*, 2005.

[71] A. Upegui and E. Sanchez. Evolving hardware by dynamically reconfiguring xilinx fpgas. In J. M. Moreno, J. Madrenas, and J. Cosp, editors, *Evolvable Systems: From Biology to Hardware*, volume 3637 of *Lecture Notes in Computer Science*, pages 56–65. Springer Berlin / Heidelberg, 2005.

[72] S. J. E. Wilton. *Architectures and algorithms for field-programmable gate arrays with embedded memory*. PhD thesis, Toronto, Ont., Canada, Canada, 1997. AAINQ28082.

[73] M. J. Wirthlin. Constant coefficient multiplication using look-up tables. *Journal of VLSI Signal Processing*, 36(1):7–15, 2004.

[74] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *FPGA '97: Proceedings of the 1997 ACM Fifth International Symposium on Field-Programmable Gate Arrays*, pages 86–92, New York, NY, USA, 1997. ACM.

[75] Y.-L. Wu, S. Tsukiyama, and M. Marek-Sadowska. On computational complexity of a detailed routing problem in two dimensional fpgas. In *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV '94, Proceedings., Fourth Great Lakes Symposium on*, pages 70 –75, Mar. 1994.

[76] 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*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

Manuals

[77] Actel. *Axcelerator Family FPGAs*, 2009.

[78] Actel. *IGLOO Low Power Flash FPGAs*, 2010.

[79] Altera. Quartus ii university interface program.

[80] Altera. *Application Note 119: Implementing High-Speed Search Applications with Altera CAM*. Altera, 2001.

[81] Altera. *Cyclone IV FPGA Device Family Overview*, 2010.

[82] Altera. *Stratix III Device Handbook*, 2010.

[83] Altera. *Stratix V FPGAs: Built for Bandwidth*, 2010.

[84] Tabula. Spacetime architecture, 2010.

[85] Xilinx. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, 2004.

[86] Xilinx. *UG208: Early Access Partial Reconfiguration User Guide*. Xilinx, 2006.

[87] Xilinx. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, 2007.

[88] Xilinx. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2007.

[89] Xilinx. *DS253: Content-Addressable Memory v6.1*. Xilinx, 2008.

[90] Xilinx. *EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design*, 2008.

[91] Xilinx. *Virtex-II Pro Libraries Guide for HDL Designs*, 2008.

[92] Xilinx. *Spartan-6 Family Overview*, 2010.

[93] Xilinx. *Virtex-5 FPGA Configuration User Guide*, 2010.

[94] Xilinx. *Virtex-6 Family Overview*, 2010.