A transformation-based approach to hardware design using higher-order functions

Rinse Wester

Members of the graduation committee:

prof. dr. ir.	G. J. M. Smit	University of Twente (promotor)
dr. ir.	J. Kuper	University of Twente (assistant-promotor)
dr. ir.	J. F. Broenink	University of Twente
prof. dr.	M. Huisman	University of Twente
prof. dr.	K. G. W. Goossens	Eindhoven University of Technology
prof. dr. -ing.	M. Hübner	Ruhr-Universität
dr. ir.	H. Schurer	Thales
prof. dr.	P. M. G. Apers	University of Twente (chairman and secretary)

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems (CAES) group

CTIT Ph.D. Thesis Series No. 15-365
Centre for Telematics and Information Technology
PO Box 217, 7500 AE Enschede, The Netherlands

This research has been conducted within the Sensor Technology Applied in Reconfigurable Systems (STARS) project (www.starsproject.nl).

This research has been conducted within the Robust design of cyber-physical systems (12700_CPS_7) project (www.stw.nl/nl/programmas/robust-design-cyber-physical-systems-cps).

This thesis was typeset using LaTeX and Ti*k*Z. This thesis was printed by Gildeprint Drukkerijen, The Netherlands.

ISBN 978-90-365-3887-9
ISSN 1381-3617; CTIT Ph.D. Thesis Series No. 15-365
DOI 10.3990/1.9789036538879

A transformation-based approach to hardware design using higher-order functions

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 3 juli 2015 om 12.45 uur

door

Rinse Wester

geboren op 24 juni 1986
te Leeuwarden

Dit proefschrift is goedgekeurd door:

prof. dr. ir.	G. J. M. Smit	(promotor)
dr. ir.	J. Kuper	(assistent promotor)

ISBN 978-90-365-3887-9

Abstract

The amount of resources available on reconfigurable logic devices like FPGAs has seen a tremendous growth over the last thirty years. During this period of time, the amount of programmable resources (CLBs and RAMs) in these architectures has increased by more than three orders of magnitude. Also, many specialized components such as DSP modules to accelerate certain parts of applications have been introduced. Reconfigurable architectures have thus evolved into heterogeneous systems.

Programming these reconfigurable architectures has been dominated by the hardware description languages VHDL and Verilog. However, it has become generally accepted that these languages do not provide adequate abstraction mechanisms to deliver the design productivity for designing more and more complex applications. To raise the abstraction level, techniques to translate high-level languages to hardware have been developed. These techniques are now commonly known as high-level synthesis where most high-level synthesis approaches are based on mainstream programming languages, in particular on the imperative programming paradigm; many high-level synthesis languages are now based on the imperative language C.

Parallelism is achieved by parallelization of for-loops. Whether parallelization of these loops is possible or not is determined by the dependencies between loop iterations. Dependency analysis is a hard problem and often, due to the imperative nature of the input language, loop iterations can not to be assumed to be independent preventing possible parallelization. To mitigate this problem, other abstractions are needed to express structure and to abstract away from the fact that imperative programming is based on state transformations which is a major source of difficulties in dependency analysis. Hence, a language that is not based on state transformations is advantageous. In this thesis, hardware is therefore designed using the functional programming language Haskell. Haskell is based on the manipulation of mathematical functions, which gives the designer more control over structure and parallelism.

In general, a function can be implemented in space (perform operations in parallel) or in time (perform operations sequential). In hardware design, the trade-off between space (chip resources) and time (execution time) is crucial. A candidate abstraction to express structure and parallelism is by means of higher-order functions which are commonly used in functional languages to express repetition and operations on lists. Using transformations of specific higher-order functions, more or less parallelism is achieved. This is under full control of the designer since the transformation distributes computations over space and time. The advantage of a functional language is that no dependency analysis is needed since the dependencies are intrinsic properties of the specific higher-order function.

The main contribution of this thesis is a design methodology for hardware based on exploiting regularity of higher-order functions. A mathematical formula, e.g. a DSP algorithm, is first formulated using higher-order functions. Then, transformation rules are applied to these higher-order functions to distribute computations over space and time. Using the transformations, an optimal trade-off can be made between space and time. Finally, hardware is generated using the CλaSH compiler by translating the result of the transformation to VHDL which can be mapped to an FPGA using industry standard tooling.

In this thesis, we derive transformation rules for several higher-order functions and prove that the transformations are meaning-preserving. After transformation, a mathematically equivalent description is derived in which the computations are distributed over space and time. The designer can control the amount of parallelism (i.e. he/she can control resource consumption and execution time) using a parameter that is introduced by the transformation. Transformation rules for both linear higher-order functions and two-dimensional higher-order functions have been derived.

In this thesis we perform several case studies using the aforementioned design methodology:

» a dot product to show the relation between discrete mathematics, higher-order functions and hardware;

» a particle filter;

» stencil computations.

A particle filter is chosen as it is a challenging application to implement in hardware due to a large amount of parallelism, data dependent computations and a feedback loop. Stencil computations are explored to extend the set of transformation rules such that the design methodology can also be applied to two-dimensionally structured applications.

In conclusion: we explored and exploited higher-order order functions as an abstraction to express structure and parallelism of hardware. Higher-order functions, combined with their transformation rules, can be an effective tool to facilitate in optimizations and trade-offs which are essential aspects of digital hardware design.

Samenvatting

Het aantal programmeerbare componenten dat gebruikt kan worden in herconfigureerdbare logica zoals FPGAs heeft een enorme groei doorgemaakt in de laatste dertig jaar. Gedurende deze periode zijn het aantal programmeerbare componenten (CLBs en RAMs) in deze architecturen met meer dan drie ordes van grootte toegenomen. Ook zijn er vele applicatiespecifieke componenten zoals DSP modules toegevoegd om specifieke computaties binnen delen van applicaties te versnellen. Herconfigureerdbare architecturen zijn dus geëvolueerd naar heterogene systemen.

Het programmeren van deze herconfigureerdbare architecturen wordt gedomineerd door de hardwarebeschrijvingstalen VHDL en Verilog. Tegenwoordig is het echter algemeen geaccepteerd dat deze talen niet de nodige abstractiemechanismes bevatten om genoeg ontwerpproductiviteit te verkrijgen voor steeds grotere en complexere systemen. Om het abstractieniveau te verhogen zijn er technieken ontwikkeld om hoog-niveau programmeertalen te vertalen naar hardware. Deze technieken staan nu bekend als hoog-niveau synthese en zijn meestal gebaseerd op veelgebruikte imperatieve programmeertalen. De meeste hoog-niveau synthese programmatuur gebruikt dan ook C of een afgeleide daarvan als invoertaal.

Parallellisme wordt verkregen door het parallel uitvoeren iteraties van for-lussen. De mogelijkheid tot parallelliseren van for-lussen hangt af van het bestaan van afhankelijkheden tussen lus-iteraties. Het analyseren van afhankelijkheden is echter erg moeilijk waardoor er vaak een afhankelijkheid moet worden aangenomen. Dit komt doordat imperatieve talen zijn gebaseerd op geheugen modificaties wat het analyse proces enorm bemoeilijkt. Om dit probleem te voorkomen zijn er nieuwe abstracties nodig om structuur uit de drukken die niet is gebaseerd op geheugenmodificaties. Door gebruik te maken van een programmeertaal die niet is gebaseerd geheugenmodificaties kunnen lastige analyse problemen worden voorkomen. In dit proefschrift wordt hardware daarom dan ook ontworpen door gebruik te maken van de functionele programmeertaal Haskell. Haskell is gebaseerd op het manipuleren was wiskundige functies wat de gebruiker meer controle geeft over structuur en parallellisme.

Functies kunnen worden uitgevoerd in ruimte (computaties worden parallel uitgevoerd) of over de tijd (computaties worden sequentieel uitgevoerd). Tijdens het ontwerpen van hardware is de afweging tussen ruimte (het aantal gebruikte componenten) en tijd (executietijd) cruciaal. Een kandidaat abstractie voor het uitdrukken van structuur en parallellisme is het gebruik van hogere-orde functies. Hogere-orde functies zijn afkomstig uit functionele programmeertalen en worden veel gebruikt voor het uitdrukken van repetitie en het toepassen van operaties op

lijsten. Door transformaties toe te passen op specifieke hogere-orde functies, kan er meer of minder parallellisme worden behaald. De ontwerper heeft hier volledige controle over omdat de transformatie computaties distribueert over zowel ruimte als tijd. Door gebruik te maken van een functionele taal is afhankelijkheidsanalyse niet meer nodig omdat de afhankelijkheden een intrinsieke eigenschap zijn van de specifieke hogere-orde functie.

De hoofdbijdrage van dit proefschrift is een ontwerpmethodiek voor digitale circuits gebaseerd op het benutten van reguliere structuren in hogere-orde functies. Een wiskundige beschrijving van een DSP algoritme wordt eerst geformuleerd met behulp van hogere-orde functies. Vervolgens worden transformatieregels toegepast op deze functies om zo de computaties te distribueren over ruimte en tijd. Met behulp van deze transformatieregels kan dus een optimale afweging worden gemaakt tussen ruimte en tijd. Vervolgens wordt er hardware gegenereerd met behulp van de CλaSH compiler waarbij de resultaten van de transformatieregels worden vertaald naar VHDL code. Gebruikmakend van programmatuur die als standaard wordt beschouwd in de industrie, wordt de VHDL code vertaald naar een FPGA configuratie.

In dit proefschrift worden transformatieregels afgeleid voor verschillende hogere-orde functies en worden bewijzen geleverd dat de transformatieregels betekenisbehoudend zijn. Het toepassen van een transformatie resulteert dus in een wiskundig equivalente beschrijving waarin de computaties zijn gedistribueerd over ruimte en tijd. De ontwerper heeft volledige controle over de hoeveelheid parallellisme (het aantal gebruikte componenten en executietijd) door het instellen van een parameter die is geïntroduceerd tijdens de transformatie. Er zijn transformatieregels afgeleid voor zowel eendimensionale als tweedimensionale hogere-orde functies.

Tevens worden er in dit proefschrift verschillende casestudies behandeld waarin de hiervoor genoemde ontwerpmethodiek wordt toegepast:

» een inwendig product om de relatie tussen discrete wiskunde, hogere-orde functies en hardware aan te geven;

» een particle filter;

» stencilcomputaties.

Er is gekozen voor een particle filter omdat dit een uitdagend algoritme is voor implementatie op hardware door de aanwezigheid van veel parallellisme, data-afhankelijke computaties en terugkoppellus. Om de verzameling transformatieregels uit te breiden, zijn er ook transformatieregels afgeleid voor stencilcomputaties.

Concluderend: voor het adequaat uitdrukken van structuur en parallellisme op hardware kan gebruik worden gemaakt van hogere-orde functies. Hogere-orde functies, gecombineerd met de bijbehorende transformatieregels, zijn een effectief middel voor het maken van essentiële afwegingen tijdens het ontwerpen van digitale hardware.

Dankwoord

Tijdens de laatste fase van het afstuderen werd ik door Jan gevraagd of ik ook interesse had in een promotie plek. Na hier enige tijd over na te hebben gedacht ben ik de uitdaging aangegaan. Intussen zijn we vier en een half jaar verder met als resultaat het proefschrift wat hier voor je ligt. Uiteraard hebben veel mensen mij geholpen tijdens deze periode en dit is dan ook de plek om ze even te bedanken.

Allereerst wil ik graag Jan bedanken voor de introductie tot de functionele aanpak van hardware ontwerp en de leuke samenwerking. Tussen rauwe hardware en abstracte wiskunde zit een enorm gebied wat een hoop leuke discussies heeft opgeleverd. Ook wil ik graag Gerard bedanken voor het creëren van onze gezellige vakgroep CAES waar ik de kans kreeg om een eigen draai aan mijn onderzoek te geven. Hoewel Gerard het altijd enorm druk had met zoveel promovendi, lukte het altijd weer om papers of hoofdstukken van dit proefschrift in een mum van tijd van goed commentaar te voorzien. Ook wil ik graag de rest van de commissie bedanken voor hun input.

Verder zijn er veel mensen die direct of indirect aan mijn werk hebben bijgedragen, bij deze wil ik hen graag ook even bedanken: Tijdens mijn onderzoek heb ik veelvuldig gebruik gemaakt van de CλaSH-hotline Christiaan, die mij altijd snel van persoonlijk CλaSH-advies kon voor voorzien. Tom, voor de interessante gesprekken over bergen en de gedeelde interesse in avontuur. De mensen die ik heb begeleid met afstuderen, Dimitrios, Floris en Erwin, voor het leuke werk dat jullie hebben verricht. Mijn oud-kamergenoot Mark, voor de leuke elektronica projecten. Mijn huidige kamergenoten Guus en Ingmar, voor alle gezelligheid en lol: het is iedere ochtend weer een verrassing in welke staat ik mijn bureau zal aantreffen. Jochem, voor het proefschrift framework en Marco voor de interessante discussies over wetenschap. De secretaresses Marlous, Thelma en Nicole, voor het regelen van alle reizen en als ik weer eens speciale wensen had m.b.t. bagage.

Er zijn twee vrienden die mij tijdens het promotieonderzoek en daarvoor erg veel hebben geholpen, mijn paranimfen Koen en Lars. Koen, bedankt voor de diepgaande discussies, gezelligheid op zowel land als water en de feestelijke aspecten van de zuid-Nederlanse cultuur. Lars, bedankt voor het gezellig biertjes drinken in Leeuwarden, leuk stappen in zowel Enschede als Leeuwarden en het altijd goed verzorgde bed & breakfast. Het doet mij dan ook erg veel plezier dat jullie mij bij staan als paranimf.

Ek wol ik graach heit en mem bedanke foar in gesellich en stimulearjend tús en fansels ek foar alle kearen dat jimme my fan en nei it stasjon brocht hawwe.

Rinse
Enschede, juni 2015

Contents

1 Introduction

The modern age is often characterized as the information-age due to the developments in electronics. Two important aspects of the information-age are communication and computation. A technology that plays an important role in both communication and computation are digital semiconductor components such as processors, memories, application specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs). FPGAs are used for two reasons: in first instance, they were used in small series for fast prototyping of digital circuits and to replace discrete components. Later, FPGAs were used to replace fixed-functionality logic. FPGAs can be found in many places like large internet routers, base stations for mobile communications and even radio telescopes. FPGAs are capable of processing a tremendous amount of data at a very high speed. An example of such a high performance FPGA-based platform is the Astron Uniboard [12], used for processing radio astronomy signals (Figure 1.1).

In contrast to CPUs, FPGAs are better able to exploit parallelism due to the large set of availale resources. Designing applications for FPGAs is therefore much closer to digital hardware design than designing software for CPUs. Compared to ASICs, using FPGAs has several advantages. The first advantage is that the same FPGA can be used in thousands of different applications resulting in a large cost reduction. Secondly, applications can even be changed when the FPGA is already installed at the customer. Thirdly, because FPGAs have a very regular structure, the latest semiconductor technology can be used. However, the wide applicability of FPGAs comes at a cost. FPGAs consume more power compared to ASICs and require more area as well. Additionally, FPGAs are difficult to program, especially for large applications.

An FPGA consists of programmable blocks, often called configurable logic blocks (CLBs), and a programmable interconnect. The gates and registers of the circuit are placed on the CLBs while the interconnect is configured in such a way that the gates in the CLBs are connected in the same way as the original circuit.

Developments in reconfigurable logic started in 1975 with the introduction of the field-programmable logic array (FPLA) by the company Intersil [61]. FPLAs were the first reconfigurable logic chips that could be programmed electronically in contrast to read-only memories of which the contents can not be changed after pro-

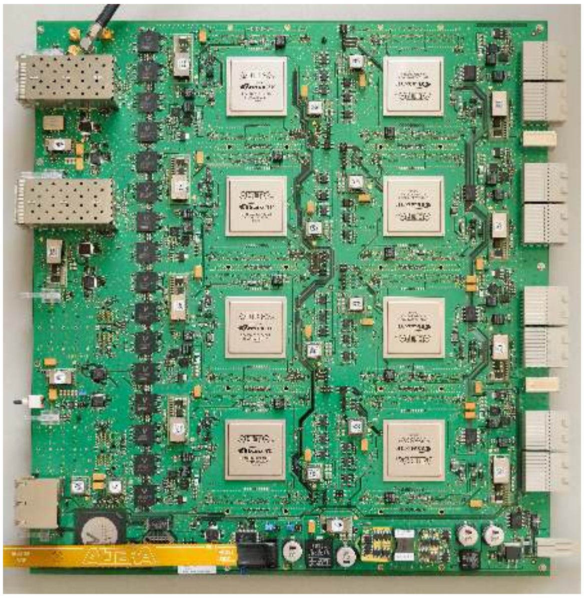


Figure 1.1 – Astron Uniboard

duction [92]. FPLAs consists of a matrix with fuses and a column of gates. A specific circuit is implemented by vaporizing fuses such that only the required connections between inputs and gates remain. The use of fuses has one drawback, once a fuse is removed it cannot be undone. FPLAs are therefore one-time programmable. This changed with the introduction of the FPGA. The configuration was no longer performed by vaporizing fuses but stored in a memory that could be changed as often as necessary.

The first commercially available FPGA, introduced in 1985, was the Xilinx XC2064 which contained 64 CLBs [119]. This FPGA had a capacity for a circuit up to 1200 gates. The configuration data (the settings of the I/O pins, CLBs and interconnect) is stored in SRAM memory cells. As SRAM memory only retains data when powered, the FPGAs has to be programmed again after power up. The configuration data is stored on a non-volatile external memory chip which is read by the FPGA during the start-up phase.

Thirty years after the introduction of the XC2064 a lot has changed, although the basic principles have stayed the same. Most FPGAs still use SRAM memory cells for configuration and use an external memory from which the configuration is read during start-up. However, the capacity in terms of CLBs has increased tremendously. Current high-end FPGAs like the Xilinx Virtex Ultrascale contain millions of CLBs [118].

1.1 Trends in reconfigurable computing

Reconfigurable computing has become a very large field of research with many applications. In this thesis we limit ourselves to the field of FPGAs. There are two important aspects of FPGAs that are important to view the trends in this field: the developments in hardware and the programming models and languages to program these architectures.

1.1.1 Hardware developments in FPGA architectures

FPGAs have seen tremendous developments the last thirty years [108]. Although a lot has changed in this period, two main trends can be observed in the hardware development. The first trend is the enormous growth in the amount of available resources in terms of CLBs, memories and interconnect. The second trend is the integration of specialized hardware to accelerate certain parts of applications.

The enormous increase of available resources becomes strikingly clear when looking at the number of logic blocks that have become available in FPGAs. During the last thirty years, the amount of logic blocks has increased from several hundreds to several millions, i.e., an increase of four orders of magnitude. In this period, also the clock frequencies have increased from several megahertz up to several hundred megahertz (depending on the design). Figure 1.2 shows the exponential increase in LUTs for Xilinx Virtex FPGAs in the last thirteen years.

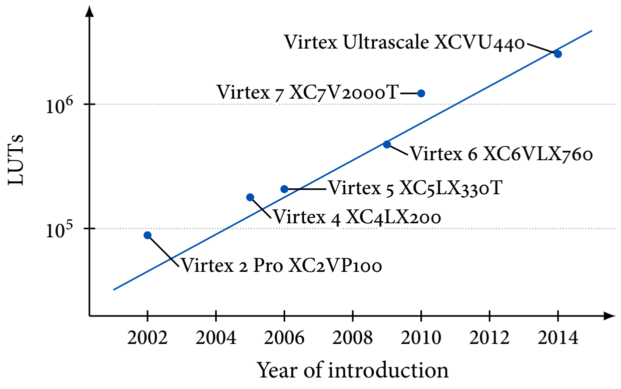


Figure 1.2 – Growth in resources of Xilinx Virtex FPGAs

Besides the increase of logic blocks, the trend has also emerged of adding more specialized hardware to accelerate certain applications. One of the first of these specialized components that have been added are special memories called block RAMs (BRAMs) and multipliers which allowed the designer to instantiate memories and multipliers much more efficiently. Special hardware for other applications soon followed in the form of components specialized for DSP operations. These DSP blocks can be configured to perform combinations of multiplication and addition with a configurable amount of bits. DSP blocks are much more area efficient and are able to run at a much higher clock frequency than their counterparts implemented using configurable logic blocks (CLBs). Currently, complete CPUs are integrated in the FPGA logic. Examples of such integrations are the Xilinx Zync FPGA [116] and the integration of an ARM Cortex in Cyclone FPGAs from Altera [8]. All these performance enhancements also requires additional bandwidth to be able keep the hardware utilized. Therefore, high-speed serial I/O standards are integrated to meet these high bandwidth demands.

In the future, both the increase in logic blocks and the addition of specialized hardware are expected to continue [101]. The amount of logic blocks is expected to scale with the advances in semiconductor technology although reliability issues are expected with smaller feature size [38]. The addition of specialized hardware is expected to continue as well. An example is the integration of multicore CPUs into FPGAs [7]. Summarizing, the once simple and regular hardware structures of FPGAs have evolved into highly heterogeneous architectures with a lot of specialized hardware.

1.1.2 Programming of FPGAs

An equally important aspect of FPGAs is the programming of these devices. This has been dominated by the hardware description languages VHDL and Verilog. However, it has become generally accepted that these languages do not provide enough productivity as demanded by current large designs. The programming of FPGAs is shaped by the targeted applications and the developments in HDLs [65].

During the last thirty years, the set of applications for which FPGAs are used has grown tremendously. Initially, FPGAs were mainly used for implementing small logic circuits. However, nowadays they are used in a wide range of applications. Many digital signal processing (DSP) algorithms are mapped to FPGAs. Examples of these applications are wireless communication, radar processing, image/video processing and radio astronomy. Given these applications, there is a clear trend towards applications that require more computational power and have higher bandwidth requirements.

Hardware description languages like VHDL and Verilog target hardware design at the RTL level. To increase productivity, languages with a higher level of abstraction are developed commonly known as high-level synthesis (HLS). Currently, most HLS tools accept a language that is derived from C [77]. Parallelism in these languages is achieved by the parallelization of for-loops. Whether or not two iterations of a loop can be run in parallel, depends on the dependencies between them. However, dependency analysis is a very hard problem and often iterations cannot be assumed independent preventing possible parallelization. Therefore, other abstractions are needed to express structure and parallelism. Additionally, the input languages for HLS are highly restricted since a lot of advanced C language features cannot be used when designing hardware. Examples of these restrictions are lack of support for pointers and, because of the reasoning above, limited support for for-loops [77].

1.2 Problem statement and approach

The developments in reconfigurable logic can be summarized as a technological arms race between developments in silicon technology on the one hand and the increasing demand for computational power on the other. FPGAs offer large performance gains compared to CPUs for applications that contain a lot of parallelism and pipelining. Applications can therefore only utilize this performance of FPGAs when parallelism can be fully exploited.

In this thesis we address the issue of deriving parallelism from the definition of an application. Achieving performance by means of parallelism is often far more complicated then just instantiating a lot of components, other factors like limited resources and length of combinatorial paths have to be taken into account as well. The languages used for hardware design should facilitate in this trade-off. In this thesis we therefore try to answer the following research question:

» *How can a designer make a transparent trade-off between resource usage (chip area) and execution time?*

We use the functional language Haskell to express circuits. Haskell is a pure functional language in which only mathematical dependencies are expressed giving a better chance of parallelization. Although a lot of work has been done on the parallelization of Haskell code for multicore [28], for FPGAs very different patterns for parallelism are required since the parallelism on FPGAs is fine grained in nature.

In this thesis, we utilize higher-order functions, an abstraction commonly used in Haskell, to express structure and parallelism. Using transformation rules, computations are distributed over space and time giving the designer full control over resource consumption and combinatorial paths. By using higher-order functions to express structure, the introduction of additional dependencies caused by the sequential nature of the input languages used in HLS tools is therefore avoided. Compared to the approach used in HLS, the approach taken in this thesis starts with a structural definition of parallelism instead of trying to deduce this structure from a sequential specification.

In order to make a transparent trade-off between resource usage and execution time, transformation rules for higher-order functions are proposed. Although such a trade-off is possible using mainstream HDLs, using transformations has some advantages: due to the mathematical nature of the specification, the transformations are provable correct. Secondly, a trade-off can be performed more rapidly when new requirements arrive by selecting new values for the parameters introduced by the transformation. Figure 1.3 shows a graphical representation of the effect of a transformation rule.

A computation mapped completely over space consumes often too many resources. By applying a transformation rule, computations are distributed over both space and time thereby limiting the resource consumption. The consequence is, however, that the execution time is increased. For some applications there is a maximum

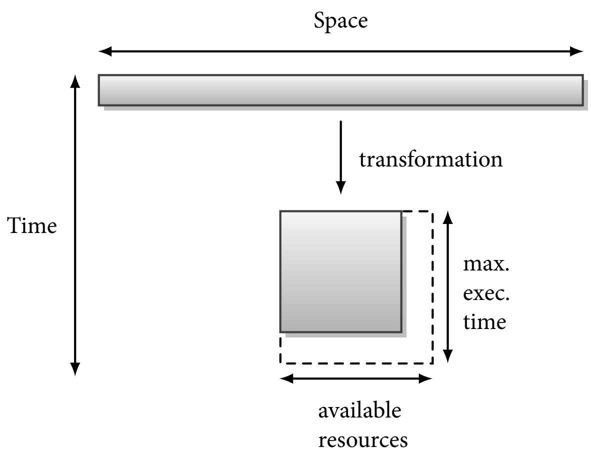


Figure 1.3 – Transformation to distribute computation over space and time

defined on the execution time. Therefore, a trade-off should be made between distributing computations over time and space.

1.3 Contributions

The main contribution presented in this thesis is a hardware design methodology targeting the implementation of digital signal processing algorithms on digital logic such as FPGAs. In the domain of DSP, algorithms are often initially defined using mathematical formulas. Before these formulas are implemented on an FPGA, they are often simulated on a PC. Usually, languages like C or Matlab are used for this purpose. The implementation on FPGAs requires another translation step: the translation of the simulation model to a model that can be translated to hardware. This translation is usually performed by hand without any formal methods to guide the process. In this thesis, design methodologies are proposed based on the use of higher-order functions to facilitate the hardware design process. Three main contributions can be distilled:

» **A design methodology for hardware based on exploiting regularity of higher-order functions.** In this thesis, a design methodology is presented showing how hardware can be designed by using a commonly used abstraction in functional languages: higher-order functions. First, a mathematical formulation of a DSP algorithm is expressed using higher-order functions to capture the structure and dependencies among operations. The second step is the transformation of this expression using transformation rules such that efficient hardware can be derived using the CλaSH compiler (chapter 3).

» **Transformation rules to distribute computations, expressed using higher-order functions, over space and time.** For several commonly used higher-order functions like *zipWith* and *foldl*, transformation rules have been derived. Additionally, these transformation rules have been proven to be meaning-preserving. The transformation rule distributes the computations, expressed using a higher-order function, over space and time. The amount of parallelism and resource usage can be fully transparently controlled by the designer using a parameter that is introduced by the transformation (chapter 4).

» **Several case studies showing the applicability of the design methodology to a large range of DSP applications.** Among others, the design methodology has been applied to a FIR filter, a particle filter and several stencil computation applications. The connection between discrete mathematics, higher-order functions and hardware is first explored in a dotproduct example after which the methodology is applied to a particle filter. Stencil computations are explored to extend the set of transformation rules such that the design methodology can also be applied to two-dimensionally structured applications (chapter 5 and chapter 6).

1.4 Outline

In chapter 2, the state of the art of hardware methodologies using transformations is presented. This chapter also gives background information on the CλaSH compiler including examples (a MAC operation and FIR filter). In chapter 3, we start with the implementation of a signal processing application with challenging characteristics for hardware implementation; a particle filter. The performance of this particle filter is increased by parallelization using an abstraction from functional programming: higher-order functions. To limit the amount of parallelism and therewith resource consumption, transformation rules are proposed in chapter 4 to perform a trade-off between execution time and area consumption by transforming higher-order functions. In chapter 5, these transformation rules are applied to the particle filter case study resulting in a large reduction in resource consumption while maintaining performance. To be able to implement more applications using the transformation-based approach, the set of transformation rules is extended such that two-dimensional data structures with overlapping data are supported as well. These additional transformation rules are proposed in chapter 6 where they are applied to a broad range of stencil applications. Finally, in chapter 7, conclusions are drawn and possible directions for future work are discussed.

1.5 Notation

In the Haskell code or mathematical definitions shown in this thesis, *xs* means plural of *x* and should be read as list of *x* elements. Similarly *xss* is the plural of *xs* and it therefore represents a list of lists containing *x* elements.

2 Background and state of the art

Abstract – *In this chapter, related work on hardware design methodologies is presented. The trends in three relevant fields are discussed, being high-level synthesis, transformational-design and functional hardware description languages. In the field of high-level synthesis, most tools converge to using a dialect of C as an input language while more specialized formalisms are used in transformational-design. Since CλaSH is the functional hardware description language used for the implementation of circuits in this thesis, the trends in functional hardware description languages are discussed and an introduction to hardware design using CλaSH is given.*

Since the beginning of automatic generation of the layout of circuits, register-transfer level (RTL) style hardware description languages (HDLs) like VHDL [11] and Verilog [30] have been the basis for circuit design. However, current circuits are becoming too complex to be written using RTL-style plain HDLs alone. Therefore, designers started to use intellectual property (IP) blocks that could be reused in several designs. Nowadays, there is support integrated in the development tools for IP blocks (like Xilinx CORE generator [117] and Altera Megafunctions [32]) which also facilitates the reuse for different designs. However, using these IP blocks still requires low-level design effort on the wire level. To increase productivity, several new approaches have arisen: transformational-design, high-level synthesis (HLS) and functional hardware description languages.

In this chapter, developments of the aforementioned approaches are discussed. Since transformations form the basis of the approach taken in this thesis, we focus mainly on several related transformation-based methodologies and the formalisms on which these methodologies are based. On the specification side, two types of input languages are discussed: imperative languages used for HLS and functional hardware specification languages. Since all implementations of hardware in this thesis are specified in the functional language CλaSH, two small circuits are specified in the CλaSH language as an introduction to CλaSH-based hardware design.

The remainder of this chapter is organized as follows. Since all hardware designs proposed in this thesis are designed and implemented using CλaSH, first background information regarding hardware design using CλaSH is presented first in section 2.1. Secondly, the state of the art in high-level synthesis (HLS) is covered in section 2.2 followed by an elaboration of transformation-based design methodologies in section 2.3. Related work regarding hardware design using functional languages is covered in section 2.4. Finally, conclusions are drawn in section 2.5.

2.1 CλaSH

All hardware designs presented in this thesis are implemented using CλaSH. The name CλaSH refers to both the language CλaSH (the CAES language for synchronous hardware) and the compiler [14, 16]. CλaSH is especially proposed for a more mathematically-based hardware design methodology [106]. The CλaSH language is a proper subset of the functional language Haskell [64, 109]. Therefore, every CλaSH design is a valid Haskell program and simulation of CλaSH hardware is essentially running a Haskell program. Using the CλaSH compiler, such a design can be translated to VHDL. Thereafter, bit files for FPGAs or full ASIC designs can be generated using industry standard tooling.

Since the initial presentation in 2009 [15], CλaSH has gone through many developments and many applications have been implemented using it. CλaSH is still under continuous development. Many abstractions have been tried and evaluated. An example of this is called arrows [46, 47]. Using arrows the composition of components is simplified since the state of each Mealy machine is hidden using a process called *lifting*. Currently, arrows have been removed in favor of signals. Using signals, composition of components is similar to function composition while the initial state can still be assigned to components. In [69], small examples of CλaSH designs are presented to show the usage of abstraction mechanisms like higher-order functions and type derivation. In [70], these abstractions are applied to a circuit. Besides relatively small designs, CλaSH has been used to design large applications as well. Among others, CλaSH has been used to implement a particle filter [RW:5], a model of the cochlea membrane [110], an FFT design for radio astronomy [RW:2], a cooperative adaptive cruise control [26] and data flow processors [84].

The fact that CλaSH is chosen as the language for implementing designs in this thesis, has several reasons. The first reason is that CλaSH uses plain Haskell as input language in contrast to other functional HDLs which are embedded languages. This has two advantages. Firstly, the simulation is a lot faster because no embedded language has to be simulated. It is also easier to handle for hardware designers since complicated types that normally arise from using an embedded DSL (EDSL) do not occur. The second advantage is extensive support for commonly used higher-order functions (HOFs) like *map*, *zipWith* and *foldl*, which are an adequate abstraction for expressing regular structures in hardware [14]. In this thesis, transformations are applied to these higher-order functions resulting in mathematically equivalent descriptions with different hardware characteristics. Using the CλaSH compiler, these HOFs can directly be mapped onto hardware without first translating HOFs to more primitive components. For furter information about the compilation process and language characteristics, the user is referred to [14].

2.1.1 Hardware design using CλaSH

Currently, CλaSH supports two machine abstractions to define hardware: a Mealy machine and signals. In this thesis, all descriptions are defined using a Mealy machine perspective as this corresponds concisely to combinatorial hardware. A Mealy machine describes hardware in terms of a function where the output and the new state is a function of the input and the current state. Mathematically, this is formulated as $(s', o) = f(s, i)$ as shown graphically in figure 2.1, where s is the current state, i is the input, o is the output and s' is the new state.

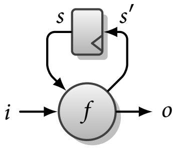


Figure 2.1 – Mealy machine

An application is implemented by defining a function f that is specific for that application. As an example of such a function f, we define a commonly used function in DSP called multiply accumulate (MAC). The MAC operator multiplies two arguments and adds the results to the previously stored result. Mathematically, this is defined as $s' = a \times b + s$ where s is the previous result, a and b the operands to be multiplied and s' the result of the calculation. In the CλaSH language, a MAC operation can be defined as shown in listing 2.1[1,2].

```
1  type Value = Signed 16
2
3  mac :: Value -> (Value, Value) -> (Value, Value)
4  mac s (a, b) = (s', o)
5      where
6          s' = a * b + s
7          o  = s'
8
9  macL = mac <^> 0
```

Listing 2.1 – MAC implemented in CλaSH

As shown on the first line of listing 2.1, the type of all values is defined as a 16 bits signed integer. This is also reflected in the type annotation of *mac* (line 3). Note that

[1] All CλaSH code in this thesis can be compiled with CλaSH version 0.3.3

[2] CλaSH is also available on http://www.clash-lang.org/

the result, the output and the new state, are shown at the end of the line in contrast to the mathematical definition of the Mealy machine. This is because the result is in Haskell is always defined last. Line 4 shows that *mac* accepts two arguments, one for the current state s and a tuple containing the inputs (a, b). The resulting tuple contains the new state s' and the output o of which the values are determined in the where-clause. In the where-clause, the actual MAC operation is performed and the result is assigned to the output (line 6 and 7). Finally, the initial state (0) is assigned to the MAC circuit using the <^> operator resulting in the component *macL*. After a reset of the circuit, the initial state of s is 0. Note that the reset circuitry is generated by the CλaSH compiler but not used during simulation. The circuit corresponding to listing 2.1 is shown in figure 2.2.

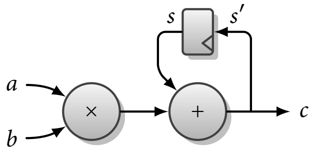


Figure 2.2 – Multiply accumulate circuit

To verify the functionality, the MAC circuit can be simulated using the predefined CλaSH function *simulateP*. Note that simulation can be performed in an interactive CλaSH environment similar to GHCI. *simulateP* takes two arguments: a lifted function representing the circuit (in this case *macL*) and a list of values acting as inputs. Since CλaSH code is valid Haskell code, simulating the architecture is equivalent to executing a Haskell program. This is also advantageous for simulation speed since no separate simulator is needed. Listing 2.2 shows the syntax to simulate the MAC circuit and the result after simulation. Note that *take* is added to stop the simulation after 3 clock-cycles since *simulateP* runs indefinitely.

```
res :: [Value]
res = take 3 (simulateP macL [(1, 2), (1,3), (2,2)])

[2,5,9]
```

Listing 2.2 – simulation of MAC

To represent array-like data structures in CλaSH, predefined typeconstructors are used to define vectors. Vectors are lists with a constant length which is encoded in the type. Commonly used higher-order functions for lists have been defined in the CλaSH languages for vectors. Examples are *vmap*, *vzipWith* and *vfoldl*.

To show the use of vectors and accompanying higher-order functions, a finite impulse response (FIR) filter is implemented. A FIR filter is a commonly used operation in the field of DSP. The operation determines a weighted sum of current and previous samples in a stream. The mathematical formulation is given in equation 2.1.

$$y_i = \sum_{n=0}^{N} c_n \times x_{i-n} \tag{2.1}$$

As shown in equation 2.1, every sample x_{i-n} is multiplied with a filter coefficient c_n after which the sum is determined. The implementation in CλaSH requires three parts: a shiftregister with the current and delayed samples, the multiplication with the coefficients and the summation. Listing 2.3 shows the implementation of the FIR filter in CλaSH.

```
1  type SRVec = Vector 3 Value
2
3  cs  = 1 :> 2 :> 3 :> 4 :> Nil
4
5  fir :: SRVec -> Value -> (SRVec, Value)
6  fir us x = (us', y)
7      where
8          us' = x +>> us
9          ws  = vzipWith (*) cs (x :> us)
10         y   = vfoldl (+) 0 ws
11
12 firL = fir <^> (0 :> 0 :> 0 :> Nil)
```

Listing 2.3 – FIR filter in CλaSH

As shown in listing 2.3, a type synonym called *SRVec* (shift register vector) is defined first. This type is used for the shift register for storing previous values of the input x. Line 5 shows the type of the FIR architecture in the form of a Mealy machine while line 6 shows the arguments and results corresponding with these types. The first argument of *fir* named *us* represents the current state and x represents the input. The output is represented by y while us' represents new state of the shift register. On line 3, the list of coefficients *cs* is defined corresponding to the list $[1, 2, 3, 4]$ in Haskell. The vector of coefficients is defined using the :> operator that puts one element in front of a vector. On line 8, the new shift register state us' is determined by shifting the current input x into *us* using the +>> operator and thereby removing the last element of *us*. Since the coefficients and *us* are combined in a pairwise pattern, a *vzipWith* is used to compute *ws*. The sum of *ws* and the output of the filter Y is determined using a *vfoldl*. Finally, on the last line, the initial state is assigned to the filter by setting all register values to zero. The resulting circuit is shown in figure 2.3 while the simulation results are shown in listing 2.4.

```
res :: [Value]
res = take 4 (simulateP firL [1,0,0,0 :: Value])

[1,2,3,4]
```

Listing 2.4 – simulation of FIR filter

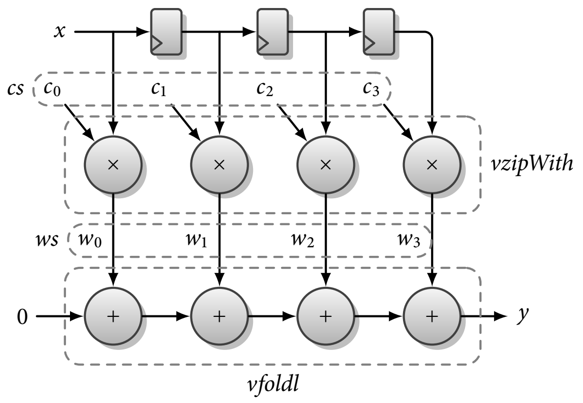


Figure 2.3 – FIR circuit

As shown in listing 2.4, the FIR filter is simulated using a stream starting with a one followed by zeroes (called an impulse). Using this stream, the impulse response of the filter is determined. For a FIR filter, the response should be equal to the set of filter coefficients. The simulation result shown on the last line of listing 2.4 shows the correct impulse response for the FIR filter.

The aforementioned FIR example shows how regular architectures can be defined. However, also more irregular applications have been defined with CλaSH, e.g., a VLIW architecture [24] and the MUSIC algorithm [62]. CλaSH is under constant development and gaining many new features. Currently, a Verilog backend is being added to better target ASIC tooling. For further information on CλaSH and the internal workings of the compiler, the reader is referred to [14].

2.2 High-level synthesis

Due to the increasing amount of resources on current FPGAs and the increasing complexity of designs, a higher level of abstraction is investigated in HDLs to keep up with productivity. High-level synthesis (HLS) is the process where a high level language is translated to gate-level hardware instead of using languages that describe hardware on the register-transfer level (RTL). These high-level input languages come in a variety of shapes, ranging from domain-specific languages for signal processing ([76] for stencil computations for example) to more generally applicable languages like C [66], SystemC [5] and Matlab [78]. Although functional HDLs can also be considered HLS languages, these get special attention in section 2.4.

2.2.1 History

High-level synthesis is an active area of research for already 30 years with the beginnings dating back to the 1970s [31]. The history of HLS can be divided into three generations with different levels of success [56, 77].

The first generation (up to early 1990) of HLS tools were mainly developed for academic research purposes and were generally ignored by industry. Several important developments like special input languages (instruction set processor language (ISPL) [89]) and force-directed scheduling [91] formed a basis for further advances in the field. First generation HLS tools became not very successful in terms of industrial use for four reasons. In this period, industry was starting to use automatic placement and routing tools resulting in a large increase of productivity and an even higher level of abstraction was not deemed necessary since place and route was the most labour intensive task. The second reason was that the input languages were considered obscure since most designers were just switching to RTL style languages and there was no need for higher abstractions. The third reason was the quality of the results, the resulting hardware was often too large due to expensive allocation and primitive scheduling of operations. The fourth reason was the fact that these tools targeted often a specific domain like DSP. These tools were therefore only used for a very small part of the whole design process.

The second generation (up to beginning of 2000s) of HLS tools focused on the translation of behavioral descriptions to hardware and did gain a lot of attention from industry [41]. One of the best known tools from that era was the Synopsys Behavioral Compiler [67]. However, also the second generation tools were not successful for reasons similar to the first generation. A main reason was that better hardware results were expected compared to handwritten design at the RTL level. The tools, however, often introduced overhead that was variable in size and often unpredictable. This made these tools cumbersome to use and the results were often of poor quality compared to hand-optimized RTL design. Similar to the first generation, the input languages were still proposed as a direct alternative to RTL design. These languages, however, were very difficult to use and therefore mostly not considered worth to learn.

For the current generation of HLS tools, the aforementioned problems have been addressed. Due to the enormous amount of resources available on chips like FPGAs, RTL style hardware description languages are becoming inadequate since they are not productive enough for large scale designs. In terms of input languages, there is a trend to switch to C-like languages to target a larger group of designers. Parallelism in for-loops is often indicated by the designer using pragmas such that the compiler can safely assume that loop iterations are independent. Although the quality of the results of these tools has increased significantly, there still are issues concerning efficiency that need attention [31]. Based on the description in the input language, it is still hard for the designer to make an estimate of the resource costs of the resulting hardware structure which is parallel in nature. This is caused by the fact that most input languages are imperative and therefore sequential in nature making it hard to relate to the resulting hardware. Complete system design involves a lot of different components with different styles and tools to define them. Currently, HLS can be cumbersome as well to use for full system design as this requires debugging possibilities that cover all levels of a complete system. An overview of current HLS tools and languages can be found in [80].

2.2.2 Example

To show how a modern high-level synthesis tool can be used to create hardware, two examples are shown. These examples are implemented using the Riverside optimizing compiler for configurable circuits (ROCCC) compiler [55, 111]. The input language for the ROCCC compiler is based on the industry standard C language. However, there are a number of limitations: only for-loops are supported, constant offsets are required when using loop-iterators and there is no support for pointers [3].

The first example shows how repetition in structure is dealt with. For-loops in ROCCC are used to replicate inputs or components. The code shown in listing 2.5 shows how to instantiate multipliers to implement a power function. This is implemented by using an input three times in a for-loop.

```
void power(int x, int& y) {
    int i;
    int total = 1;
    const int N = 3;
    for(i=0; i<N; ++i)
        total *= x;
    y = total;
}
```

Listing 2.5 – ROCCC example of *power* (based on example in [3])

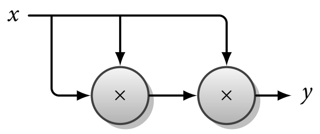


Figure 2.4 – Structural loops in ROCCC

As shown in listing 2.5, the loop has a constant number of iterations (3) and the compiler can therefore infer that all loops can be fully unrolled. Since the variable *total* is initialized with 1, the compiler eliminates one instantiation of a multiplier since 1 is the identity element of $*$. For comparison, the circuit of figure 2.4 is also implemented in CλaSH as shown in listing 2.6.

In the ROCCC example, x is used three times by referring to it in a for-loop. In CλaSH, this is implemented by constructing a vector with three x values which is

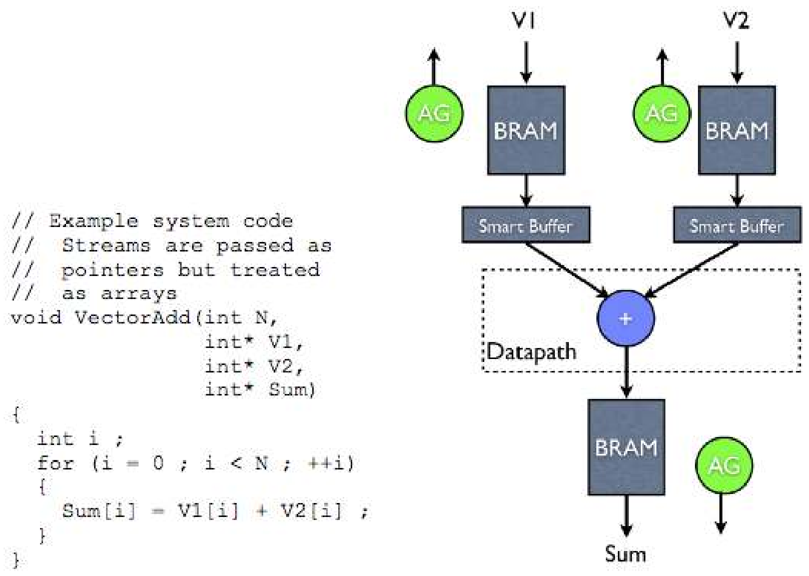
```
type Value = Signed 16

cubed :: Value -> Value
cubed x = y
    where
        xs = vcopy 3 x
        y = vfoldl (*) 1 xs
```

Listing 2.6 – ROCCC example in CλaSH

constructed using the *vcopy* function. This function creates a vector by repeating the element *x* three times. While the number of loop iterations in the ROCCC example is determined by the constant *N*, in the CλaSH program it is inferred from the length of the vector *xs*. The loop structure is implemented using a *vfoldl* HOF and uses 1 as initial value. Using the *vfoldl* HOF, the initial value 1 is multiplied with the first *x* from the vector *xs* before being multiplied with the second *x* and third *x* value resulting in the final value for *y*.

An other important concept in the ROCCC input language are *streams*. Streams are lists of data that are elements that can only be accessed sequentially over time. These streams are stored in BRAMs and can be accessed using for-loops. Data is buffered using so-called *smart buffers* [111] to increase the reuse of data and to provide a function similar to shift registers. Figure 2.5 shows an example using smart buffers where two streams (*V1* and *V2*) are added in a pairwise fashion.



```
// Example system code
//  Streams are passed as
//  pointers but treated
//  as arrays
void VectorAdd(int N,
               int* V1,
               int* V2,
               int* Sum)
{
  int i ;
  for (i = 0 ; i < N ; ++i)
  {
    Sum[i] = V1[i] + V2[i] ;
  }
}
```

Figure 2.5 – Sequential loops in ROCCC (reprint from [3])

As shown in the code of figure 2.5, the streams *V1* and *V2* are accessed sequentially using a for-loop. In this for-loop, elements from both streams are added element by element and stored in the output stream *Sum*. From this C code, the ROCCC compiler generates an adder for the addition of elements, BRAMs for storing streams, address generators (AGs) for addressing elements in the stream and smart buffers for data reuse.

Since CλaSH focuses mainly only the structural description of hardware, the address generation has to be added by hand. Listing 2.7 shows the example of figure 2.5 defined in CλaSH.

```
vectoradd cntr (v1, v2) = (cntr', (sum, addr_v1, addr_v2, addr_sum))
    where
        cntr'    = cntr + 1
        sum      = v1 + v2
        addr_v1  = cntr
        addr_v2  = cntr
        addr_sum = cntr
```

Listing 2.7 – CλaSH code of *vectoradd*

A shown in the CλaSH description of listing 2.7, a counter *cntr* is used to generate addresses to access a BRAM outside this component. In the where-clause, the new state of the counter *cntr'* is determined by adding 1 to it every clock cycle. The actual calculation is performed on the fourth line where the current values of stream *V1* and *V2* are added to find *sum*. Finally, the last three lines show that the addresses are simply the value of the counter named *addr_v1*, *addr_v2* and *addr_sum* respectively.

On a more abstract level, the aforementioned example shows how two large lists of data are added in a pairwise fashion. Performing all computations in parallel requires far more hardware then reasonable which is why all elements are added sequentially. To make better use of the hardware, a level in between these extremes is needed. In section 4.1, a transformation rule is presented that allows the designer to design hardware that performs the computations partially parallel and partially sequential.

Although many architectures can be described using the C-based input language of ROCCC, there are a few limitations. The first limitation is that many features of the C language can not be used with ROCCC. Examples of these are while-loops and pointers. Furthermore, functions representing a component are required to be formatted in a special way (far more restricted than plain C code). Like with many other HLS tools, the efficiency of the resulting hardware depends highly on the dependency-analysis of for-loops. Due to the sequential nature of the input language, dependencies might be inferred between loop iterations that are not present in the mathematical specification of the algorithm. Therefore, opportunities for performance gains by parallelization are missed. In CλaSH, all loop strucutres are expressed using higher-order functions.

2.3 Transformation-based design methodologies

Transformations are used in many hardware design methodologies and tools. In this section, the state of the art on transformation-based design methods is reviewed. Since all transformation-based design is a research area too large to review in this thesis, the focus here is on design methods based on rewriting of designs. The transformations considered in this section transform (parts of) a formal description of a design into a mathematically equivalent description resulting in different hardware characteristics. Depending on the chosen transformation and constraints set by the designer, different transformations lead to different hardware characteristics. From these different designs, the most suitable design is then chosen.

Examples of rewriting using transformations are utility directed transformationss (UDTs) [75] and the algebraic approach in [93]. UDTs are transformations that are controlled using utility functions [75]. After applying a transformation, utility functions are evaluated giving the performance metrics of the transformed design. This process is embedded in an optimization algorithm to maximize utility, e.g., minimizing area and maximizing throughput. The transformations used in UDTs result in functionally-equivalent circuits.

Sometimes, approximations are acceptable and can be exploited to derive more efficient hardware. In audio and video applications such approximations are often not observable from an audio or video quality perspective but do result in a significant saving in hardware costs. In [93], complex arithmetic operations are approximated using Taylor series and expressed as polynomials. The polynomials are then symbolically modified such that they can be mapped to more efficient hardware components like MACs.

On a fundamental level, transformational-design has a fundamental limitation regarding *completeness*. A transformation system is considered complete if the optimal solution is in principle always reachable given the set of available transformations. In general, this is not the case for any general-purpose design language as shown in [112]. However, in practice this property is commonly not considered a problem.

2.3.1 The SPIRAL framework

A particularly relevant approach of transformational-design is the approach to hardware design taken in the SPIRAL project [96, 97]. SPIRAL is a software code generator for linear transforms like discrete Fourier transforms (DFTs) with automatic optimization for different hardware platforms. The optimization process iteratively applies transformations to a mathematical definition of the DFT until a sufficiently fast implementation for a particular hardware platforms is found (figure 2.6). These transformations always result in a mathematically equivalent formulation of DFT algorithms. During transformations, a DFT formula is replaced by a new, mathematically equivalent, formula with different characteristics when mapped to hardware. Examples of supported DFT algorithms are both complex and real fast Fourier transforms (FFTs), the discrete cosine transform (DCT) and the Walsh-Hadamard transform (WHT) [95, 113].

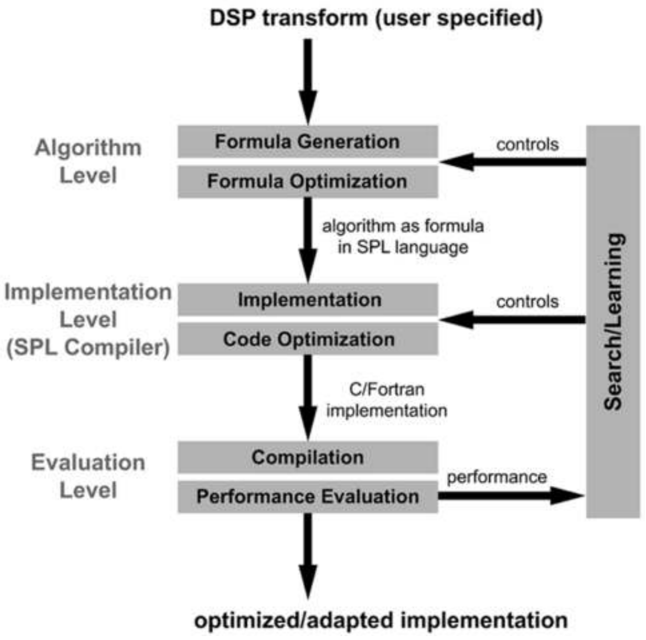


Figure 2.6 – Compilation process of SPIRAL (reprint from [96])

Figure 2.6 shows the compilation and optimization process of the SPIRAL compiler. At the algorithm-level, a user selects a DSP transform and a size. The selected DSP transform is translated into a formula to which transformations can be applied. These formulas are represented using signal processing language (SPL) [120], a language to express only DFT-like formulas. The actual imperative code of the transform is generated from the optimized SPL formulation. This implementation is compiled using standard off-the-shelf compilers after which performance metrics are derived. Simulation is performed by executing the compiled program. Performance metrics derived during simulation are used to guide the optimization process, closing the loop shown in figure 2.6.

In addition to fast software implementation of DFTs for general purpose CPUs, other hardware is targeted as well. In [35], FPGAs are targeted by generating Verilog code. Modern CPUs are often multicore architectures, SPIRAL also supports the parallelization of transforms for these architectures [43]. Similarly, the formalism used in SPIRAL has also been used to implement efficient DSP algorithms on single instruction multiple data (SIMD) architectures [98].

2.3.2 SIL

The intermediate representation of a program should facilitate the use of transformations. In the SPIRAL project the signal processing language (SPL) (which looks like an algebra language for expressing matrix operations) is used to represent formulas of DFTs. Often, a graph-based representation is used where the edges are data dependencies and the nodes operations. The SPRITE input language (SIL) [68, 82] is such a language and has been used as an intermediate representation for HLS. From a high-level language, a SIL representation is generated to which transformations are applied. Using hardware compilers, this representation can be translated to actual hardware. SIL uses control data flow graphs (CDFGs) as underlying model and can therefore model both data flow and control flow in a single graph [60]. Transformations of the SIL model are meaning-preserving but have to be done by hand [81]. In modern HLS tools, such transformations are automatically applied.

In order to map the operations onto hardware, the nodes within a SIL model have to be grouped before they are scheduled. The SIL representation of regular structures (structures from for-loops for example) is a collection of nodes with edges in between. Therefore, after obtaining a CDFG from the input language, information that the input language once contained a repeating structure is gone. The possibility to exploit this regularity, using array processing for example, is therefore much harder. The approach taken in this thesis expresses structure using higher-order functions (HOFs) to which transformations are applied thereby exploiting the regularity found in these HOFs. On the lowest level, the definition of an application can still be considered a graph but nodes may contain HOFs. Therefore more structure remains to be exploited.

2.3.3 Squigol

In chapter 4, an algebraic notation for rewriting higher-order functions will be introduced. A similar approach has been used during the development of the Bird-Meertens formalism (BMF) [18, 79] culminating in a language called Squigol. The BMF or Squigol can be described as a calculus for the construction of programs based on equational reasoning [48]. This calculus is a transformational approach to programming, in the sense that programs are first defined to be clear and understandable without focusing on efficiency. Efficiency is achieved by rewriting the original definition into a mathematically equivalent definition (equational reasoning) but with better performance characteristics. During every step in the rewrite process only mathematically proven transformations are applied which guarantee that the end result is functionally equivalent to the original definition. Many of the notational conventions in BMF can be found in Haskell as well. For example, consider the Haskell higher-order function *foldl*:

$$z = \mathit{foldl}\ (\oplus)\ x\ [y_1, y_2, y_3, y_4]$$

which is in Squigol defined as:

$$z = \oplus \not\twoheadrightarrow_x [y_1, y_2, y_3, y_4]$$

The reduction operator $\not\twoheadrightarrow$ (the *foldl* in Haskell) is defined as an infix operator accepting a binary operator $\oplus$ and a list $[y_1, y_2, y_3, y_4]$ as arguments. The parameter x is the starting value for the reduction. Using equational reasoning, the example can be rewritten to:

$$z = \oplus \not\twoheadrightarrow_x \; [y_1, y_2, y_3, y_4] \;\Rightarrow\; (((x \oplus y_1) \oplus y_2) \oplus y_3) \oplus y4$$

When the operator $\oplus$ is specified as the addition operator $+$ and x as 0, the definition can be used to determine the sum of a list of numbers. This results in $+ \not\twoheadrightarrow_0 \; [2, 4, 8, 16] \;\Rightarrow\; (((0 + 2) + 4) + 8) + 16 \;=\; 30$ and similarly in Haskell *foldl* $(+)\ 0\ [2, 4, 8, 16] = 30$.

The approach of achieving performance by equational reasoning is used a lot in more recent implementations of the Glasgow Haskell compiler (GHC). Especially in data parallel Haskell [28], a lot of transformations are applied in order to gain performance on multicore-machines [33]. The transformations in the BMF are developed in the era of single-core PCs. Therefore, different trade-offs are needed for parallel architectures like multicore machines and FPGAs.

2.3.4 Challenges

As elaborated in this section, transformation-based design methodologies have been applied to many applications in hardware and software development. Especially in the field of DSP, as seen in the SPIRAL project, transformations have been shown to be effective. However, some challenges remain. For SPIRAL, the current challenge is to widen the set of supported applications which requires changes to the formalism on which it is built. Although the compilation process used for generating DFT algorithms in SPIRAL can be used for many applications, the DFT matrices language SPL has to be extended significantly to be able to support different applications. In general, the challenge for transformation-based design is to find a formalism which allows the designer to use abstractions that fit the application domain while still delivering an efficient result. We will take up this challenge in chapter 4 of this thesis.

2.4 Functional hardware description languages

Similar to the developments in high-level synthesis, the functional programming world has also been working on increasing the productivity of hardware design. This productivity issue is targeted using the large amount of abstractions that are available in functional languages. Examples of such abstractions are polymorphism, higher-order functions and λ-abstractions. Using a functional language as the basis for hardware design has a few advantages over standard HDLs like Verilog and VHDL. The description of hardware in a functional language often describes what happens in a single clock cycle [105]. This makes the timing model much simpler for simulation resulting in quicker simulations compared to the use of delta-delay simulations in VHDL and Verilog. Simulation in a functional HDL is often just the execution of a function representing the hardware. Additionally, several abstractions that are commonly used in functional programming like type-derivation and higher-order functions are also available in the functional hardware description languages [44]. When using a pure and lazy functional programming language, the description of the hardware is side-effect free and inherently parallel. The ordering of expressions in the code is therefore irrelevant and far closer to a structural description of the hardware compared to imperative languages. Functional programming languages are also known for their advanced type systems. This allows for the use of formal methods which are hard to integrate in the industry-standard languages Verilog and VHDL.

2.4.1 A historical perspective

One of the first uses of functional languages for hardware design came with the introduction of μFP [103]. In μFP, hardware is designed by creating expressions in which primitive functions are combined using combining forms to form complete circuits. Using these combining forms, μFP also supports the production of layout. Every circuit in μFP is a function which simplified simulation tremendously. Circuits could be simulated by giving the inputs as arguments to this function returning the simulation data as result. In retrospect, this became the standard way of simulation in functional HDLs.

A similar approach to functionality and layout has been applied to the Ruby language [53]. Although not a functional language, Ruby is a language of functions and relations, circuits are constructed using primitives and composition in the same way. Layout can be expressed using these relations as well. Compared to CλaSH, the set of abstractions is rather limited since CλaSH can exploit a lot abstractions available in the modern Haskell language, like type derivation and data dependent types, while Ruby only supports primitive functions and relations.

The functional HDL that set the standard in exploiting the features of a functional language was Lava [19]. Lava is a functional HDL embedded in Haskell (an embedded domain specific language (EDSL)) that can be used to design, verify and implement circuits. Since the circuit is represented using an embedded language, it can be interpreted in many ways. Interpreters are used for simulation, verification, layout and implementation. Hardware is generated using the implementation interpreter which generates structural VHDL. Listing 2.8 shows the definition of a halfadder in Lava.

As shown in the code of the half adder, Lava relies heavily on the use of monads (an abstraction for handling non-pure operations). This is an approach to be able

```haskell

halfAdd :: Circuit m => (Bit, Bit) -> m (Bit, Bit)
halfAdd (a, b) =
    do
        carry <- and2 (a, b)
        sum   <- xor2 (a, b)
        return (carry, sum)
```

Listing 2.8 – Half adder in Lava from [19]

to support loops in the circuits description which is often the case when memory is included. It also shows the main data type in Lava, the *Bit*. Every data type of signals in Lava is formed by a composition of bits. Due to the fact that Lava is an embedded language in Haskell, a custom simulator is required to be able to test circuits. This embedding also restricts the use of pattern matching, a concise method to express choice.

In order to increase the productivity of the designer, a lot of abstractions mechanisms that are available in Haskell have also been made available in Lava. Examples of these abstractions are type derivation and higher-order functions. Higher-order functions, in particular, are an abstraction that are very intuitive for describing structure in hardware.

Following the compositional approach explored in Lava, formal methods to design hardware *correct by construction* were developed. An example of such a system is formal system design (ForSyDe) [99]. ForSyDe is based on Haskell and allows the designer to define Mealy machines using two functions: one function that determines the next state and one for the output. VHDL can be generated as well using ForSyDe but there are some restrictions on the Haskell constructs that can be used like pattern matching. A comparison of the capabilities of ForSyDe in relation to CλaSH can be found in [14].

Similar to Lava, the Hydra language [86] is also based on Haskell. Hydra is an embedded language especially developed for simulation of control-oriented digital circuits like CPUs. To represent connections between components, Hydra makes use of a data-type called *stream*. Streams are infinite lists of values but can be used since Haskell uses lazy evaluation. Registers and memory are implemented by delaying the data in streams.

The same techniques used in Lava have also been transferred to other languages than Haskell. HWML [115] is a functional HDL developed using the functional language ML. HWML also has support for streams, type derivation, VHDL generation and higher-order functions. However, the same disadvantages arise as are found in EDSLs like Lava, e.g., complicated use of types for the user due to data types.

2.4.2 State of the art

Many of the aforementioned techniques are exploited in current functional HDLs. In particular, the modern implementation of Lava, Kansas Lava [51], makes extensive use of modern features in Haskell. Kansas Lava makes use of two types of embedding, shallow and deep embedding. Using shallow embedding, a lot of functions are made available for the designer for composing circuits in a transparent way. For hardware generation deep embedding is used where the whole circuit is represented using a tree-like data structure. This data structure is then traversed to generate VHDL. For more details regarding the implementation of embedded HDLs in Haskell, the reader is referred to [49] and [9].

In contrast to compiling functional languages to hardware, hardware specifically for the execution of functional languages has been developed as well. Examples of this are the processors *Reduceron* [83] and the currently under development *PilGRIM* [20]. These processors have been developed to accelerate the execution of a basic functional language by focusing particularly on the memory access.

Although not based on a functional programming language anymore (it was based on Haskell as well), Bluespec [85] is a state of the art language focused on high-level specification of hardware. The input language of Bluespec is similar to SystemVerilog with which the designer describes circuits using *guarded atomic actions*. In Bluespec, *guards* are used to trigger actions based on certain conditions where an *action* is a change of the global state. Every action is *atomic* meaning that the state remains consistent. Using these properties, Bluespec can be used to compose larger systems based on actions. Bluespec has become an industrial-strength tool and has been used for large scale applications like H264 video decoding [74] and wireless communication [36].

The newest Haskell based HDL currently available is CλaSH [14]. What makes CλaSH distinct from other approaches is that the CλaSH language is not embedded in a host language. Instead, plain Haskell code is transformed to VHDL from an internal representation of the Glasgow Haskell compiler (GHC). Therefore, no embedded language is required making it more intuitive for the users.

2.4.3 Challenges

Although functional hardware description languages have been shown to be an adequate tool for hardware design, some challenges remain. Several abstractions from functional languages have been added to functional HDLs but there is still a need for more [105], to improve the expression of algorithms on hardware without too many RTL style details. In particular, recursion is an abstraction that would be very useful for many algorithms [14]. Also the integration of more formal methods like theorem provers in hardware design is desired. Through the use of formal methods it is expected that the costs of testing can be reduced [44].

2.5 Conclusions

Several different hardware design methodologies have been covered in this chapter that are particularly relevant to the hardware design methodologies presented in this thesis. First, CλaSH is introduced and two examples are given to show how circuits are defined using Mealy machines and how regular structures are specified with higher-order functions. Following the CλaSH section, the developments in high-level synthesis, transformation-based design methodologies and functional hardware description languages have been presented.

The main trend to be observed in the field of HLS is the support for mainstream imperative programming languages like C. Although these tools have become much more accepted by industry and the quality of the results has increased, there are fundamental limitations with using C as input language. Many features like pointers and data-dependent loops can not be used or only under very strict conditions. Also many assumptions about the execution model of C (imperative) do not apply when used for hardware design.

More formal approaches can be found in the area of transformation-based design where an underlying formalism is used to which transformations can be applied. The SPIRAL project is an example of this where a formal system is developed for efficient code generation of DFT-like transforms. The main trend that can be observed is the widening of supported applications. This is achieved by extending the formalism on which it is built.

Functional HDLs combine the high level of abstraction from HLS and the formal methods that can be found in transformational design. Most functional HDLs are embedded in a functional host language and can therefore exploit a lot abstractions from the host language. However, using an embedded language imposes some restrictions. For example, features like pattern matching can not be used. CλaSH, on the other hand, does not have these restrictions since CλaSH translates plain Haskell to hardware.

In this thesis, CλaSH is used for all hardware designs. The motivation for this is that CλaSH has a lot of abstractions that are specifically useful for hardware design. Since CλaSH is based on a pure functional language, there are no side effects which simplifies the use of transformations. Another advantage is the support for higher-order order functions which can be used directly to structurally describe hardware.

3 A Fully Parallel Particle Filter

Abstract – *In this chapter, a two-step design method is introduced to maximize parallelism by exploiting the mathematical structure of a signal processing application. A particle filter is used as an example application. During the first step, the mathematical formulation of a particle filter is reformulated in Haskell using higher-order functions, producing a model that can be transformed and trivially simulated. To derive hardware, small parts of the description have to be altered such that hardware can be generated automatically using the CλaSH compiler. By implementing a particle filter tracking algorithm, the feasibility of this method is shown. In particular, higher-order functions are a useful abstraction to express mathematical dependencies and parallelism.*

Research on particle filters, a state estimation methodology, has become popular since the publication of [52]. This popularity has also spawned research on designing hardware to accelerate these applications. FPGAs in particular, are a popular target due to the increasing amount of parallelism available on these chips. Several challenges arise when implementing particle filters on FPGAs. Particle filters contain a lot of parallelism, data dependent processing and a feedback loop. All these properties require different trade-offs to derive efficient hardware.

In this chapter, we propose a two-step design method using CλaSH applied to a particle filtering application. Performance is achieved by maximizing the amount of parallelism. The first step is formulating the mathematical definition of the signal processing application in Haskell. This Haskell description is then modified slightly such that it is accepted by CλaSH and hardware can be generated. The design method is shown graphically in figure 3.1.

The reason for splitting the design into two steps is that fundamental changes in the mathematical definition are easier to perform in the Haskell specification than in the CλaSH specification. The Haskell specification uses double-precision floating point operations while the implementation in CλaSH uses fixed-point operations. For efficiency reasons, only fixed-point calculations are currently supported

Large parts of this chapter have been published in [RW:1].

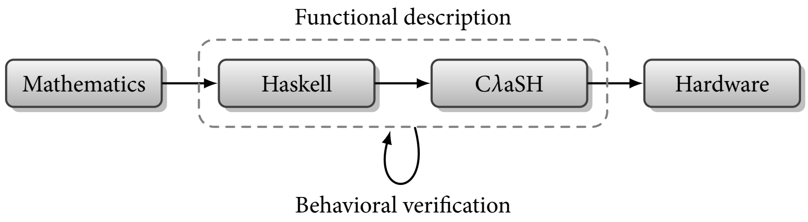


Figure 3.1 – Hardware design method

by CλaSH. Fundamental changes to the application like the architecture of the application are therefore performed in the Haskell definition while the hardware implementation details are covered by the CλaSH implementation. The benefit of two distinct steps is a clear division between architectural design and low-level hardware details like fixed-point representation.

The application chosen for evaluation of the design method towards a fully parallel implementation is a particle filter as it is challenging due to excessive parallelism, data dependencies and feedback. In order to achieve high performance, the focus of this chapter is on the parallelization of all steps of the particle filter. Data-dependent operations in the resampling step, in particular, are challenging to parallelize.

Before diving into the details of the hardware design methodology, an introduction to particle filtering is given in section 3.1 followed by related work presented in section 3.2. In section 3.3, the hardware design method is presented while hardware results are presented in section 3.4. Finally in section 3.5, conclusions are drawn regarding the effectiveness of the design methodology.

3.1 Particle Filtering

Particle filtering is a Bayesian filtering technique to determine the state variables of a system based on noisy measurements [10]. For each measurement, the belief of the state is recursively updated using a measurement and the previous state. This results in a posterior probability about the state of the system, e.g., the position and speed of an object. The more measurements are used, the more accurate the prediction becomes. Since these measurements contain noise, the resulting belief will be in the form of a probability density function (PDF). Examples of these measurements are frames from video streams and range-Doppler images from radar. These usually arrive at regular time intervals. Analytically finding the posterior probability is often mathematically intractable (the integrals used in the formulation cannot be solved) which is why approximation methods are used. Particle filtering is a Monte Carlo approach that repeatedly generates random samples and eliminates these partially according to a selection function. The number of particles is kept constant. Mathematically, the filtering problem is to find the PDF of the state vector $\vec{x}_k$ given the measurement z_k (k is the iteration number of the filter):

$$p(\vec{x}_k \mid z_k) \tag{3.1}$$

In a particle filter, this PDF is approximated by a collection of particles $x_k^{(i)}$ where $i = 1 \ldots N$ is the index of a particle. A higher density of particles represents a higher probability in the continuous state space. In fact, for N approaching ∞ the particle filter approximation is equal to the exact solution in terms of mean square error [34]. Figure 3.2 shows the continuous PDF and figure 3.3 the particle filter approximation.

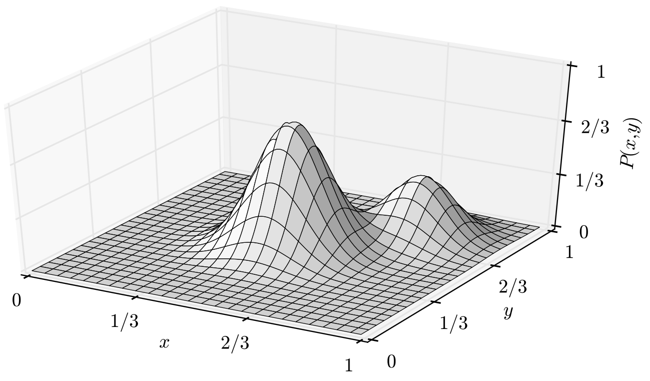


Figure 3.2 – Continuous PDF

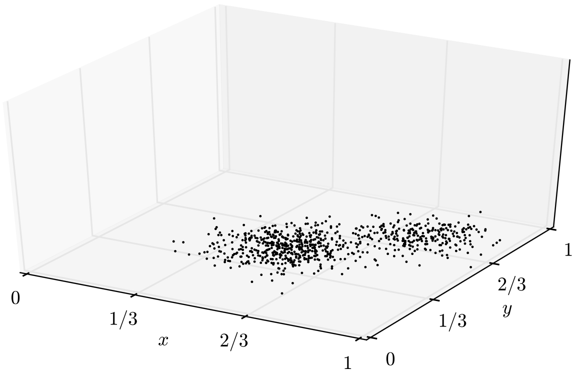


Figure 3.3 – Particle filter approximation of PDF

A commonly used type of particle filter is the sequential importance resampling filter (SIRF) which consists of four steps: *prediction*, *update*, *normalization* and *resampling* [27]. Each time a measurement arrives (the sequential part), these four steps are performed and alter the particles for the next measurement forming the feedback loop shown in figure 3.4.

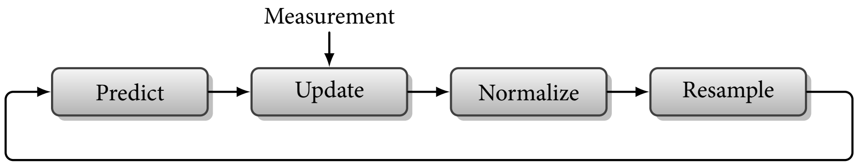


Figure 3.4 – Structure of a particle filter

Prediction

During *prediction*, the next state is derived from the current state using the known dynamics of the system. The dynamics of the system describe how the state space changes during a single iteration. Examples are anything from continuing on a straight path to complex nonlinear paths. Often, this path also includes some uncertainty. The behavior of the system during a single iteration is expressed mathematically by drawing samples from a PDF. Equation 3.2 shows this abstract definition.

$$x_k^{(i)} \sim p(\vec{x}_k \mid \vec{x}_{k-1}) \tag{3.2}$$

In practice, drawing particles from this distribution is performed by evaluating the System Dynamics function f for all particles, $x_k^{(i)} = f(x_{k-1}^{(i)}, u_k^{(i)})$. $u_k^{(i)}$ is noise sampled from some probability distribution, not necessarily a Gaussian distribution. The dependence between $x_k^{(i)}$ and $x_{k+1}^{(i)}$ now comes from f combined with the distribution of $u_k^{(i)}$.

Update

When a new prediction has been made, a measurement is used to *update* this prediction during the update step. In this step, weights $\omega_k^{(i)}$ are assigned to all particles representing the importance of a particular particle. Equation 3.3 shows the generic mathematical formulation of this. Note that the weight $\omega_k^{(i)}$ is directly determined by the PDF, the weights are not drawn from it.

$$\omega_k^{(i)} = p(z_k \mid x_k^{(i)}) \tag{3.3}$$

Similarly to the prediction step, equation 3.3 shows the generic mathematical formulation of the update step. To find the actual weights, an application-dependent update function g is needed. This function returns, given a particle $x_k^{(i)}$, a single measurement z_k and noise sample v_k, a weight $\omega_k^{(i)}$ for each particle $x_k^{(i)}$. Given an application dependent update function g, equation 3.4 shows the specific implementation of equation 3.3.

$$\omega_k^{(i)} = g(x_k^{(i)}, z_k, v_k), \quad \text{for} \quad i = 1 \dots N \tag{3.4}$$

Normalization

The integral of any real PDF should be 1 hence this should also hold for the sum of all the weights. This is realized in the normalization step where every normalized weight $\tilde{\omega}^{(i)}$ is found by:

$$\tilde{\omega}^{(i)} = \frac{\omega^{(i)}}{\omega_{tot}} \quad \text{for} \quad i = 1 \dots N$$
$$\text{where} \quad \omega_{tot} = \sum_{n=1}^{N} \omega^{(n)} \tag{3.5}$$

Resampling

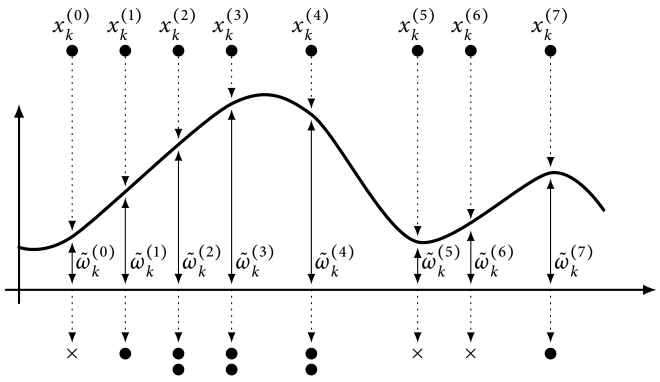
The last step performed in a particle filter iteration is the resampling step, which is needed to prevent degeneracy of weights [27]. Degeneracy occurs when particles with a high weight are given an even higher weight over several iterations. This makes the particles with a low weight insignificant. Resampling prevents this from happening by mixing particles. Particles are replicated 0, 1 or more times proportional to their normalized weight $\tilde{\omega}^{(i)}$, while keeping the total number of particles constant. Mathematically, the resampling process is selecting particles as formulated in equation 3.6:

$$p\left(\tilde{x}_k^{(i)} = x_k^{(i)}\right) \propto \tilde{\omega}_k^{(i)} \quad \text{for} \quad i = 1 \dots N \tag{3.6}$$

The probability that a particle $x_k^{(i)}$ is replicated (the particle after resampling is denoted as $\tilde{x}_k^{(i)}$) proportionally to its weight $\tilde{\omega}_k^{(i)}$ is expressed in equation 3.6. Figure 3.5 shows the process of resampling, as expressed in equation 3.6, graphically.

As shown in figure 3.5, particles with a low weight are discarded ($\times$) while particles with a high weight are copied (•) or replicated (••). This process includes some randomness to ensure some variation in how many times a particle is replicated. The total number of particles is kept constant.

Resampling is highly data-dependent which is challenging for a parallel hardware implementation [22]. There exist several techniques to implement resampling [58], of which residual systematic resampling is most commonly used. In short, residual systematic resampling replicates particles according to the amount of fixed intervals $\frac{1}{N}$ are within the range of a single weight given a random offset $0 < u_0 < \frac{1}{N}$. The resampling technique used in this chapter is called residual systematic resampling (RSR), a modified version of Systematic Resampling but mathematically equivalent [22].



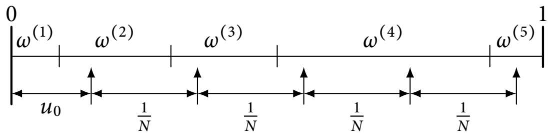
$x_k^{(0)}$ $x_k^{(1)}$ $x_k^{(2)}$ $x_k^{(3)}$ $x_k^{(4)}$ $x_k^{(5)}$ $x_k^{(6)}$ $x_k^{(7)}$
$\tilde{\omega}_k^{(0)}$ $\tilde{\omega}_k^{(1)}$ $\tilde{\omega}_k^{(2)}$ $\tilde{\omega}_k^{(3)}$ $\tilde{\omega}_k^{(4)}$ $\tilde{\omega}_k^{(5)}$ $\tilde{\omega}_k^{(6)}$ $\tilde{\omega}_k^{(7)}$

Figure 3.5 – Graphical representation of resampling

RSR consists of two steps: first the replication factor is determined based on the weight of a particle, followed by the actual replication of particles. Equation 3.7 gives an expression to determine the replication factor r_i for a single weight $\omega^{(i)}$.

$$\begin{aligned} r_i &= \lfloor(\omega^{(i)} - u_{i-1}) * N\rfloor + 1 \\ u_i &= u_{i-1} + \frac{r_i}{N} - \omega^{(i)} \\ \text{for} \quad i = 1 \dots N \quad &\text{and} \quad u_0 \sim \mathcal{U}\left(0, \frac{1}{N}\right) \end{aligned} \tag{3.7}$$

Figure 3.6 shows a graphical representation of RSR expressed in equation 3.7. Basically, the replication factor is determined by the amount of arrows pointing into the range expressed by the normalized weight. The randomization in resampling is implemented by the random offset u_0 sampled from the uniform distribution ($\mathcal{U}$). Figure 3.6 shows how the weights $\vec{\omega} = \{0.09, 0.21, 0.2, 0.4, 0.1\}$ are translated into replication factors $\vec{r} = \{0, 1, 1, 2, 1\}$.



0
1
$\omega^{(1)}$ $\omega^{(2)}$ $\omega^{(3)}$ $\omega^{(4)}$ $\omega^{(5)}$
u_0
$\frac{1}{N}$ $\frac{1}{N}$ $\frac{1}{N}$ $\frac{1}{N}$

Figure 3.6 – Graphical representation of Residual Systematic Resampling

When all replication factors r_i have been found, the actual replication of particles can be performed. During replication, every particle $x_k^{(i)}$ is replicated r_i times and the resulting sets are merged into a single set of new particles $\tilde{x}_k^{(i)}$. The mathematical formulation of replication using the concatenation operator || is shown in equation 3.8.

$$\{\tilde{x}_k^{(1)}, \tilde{x}_k^{(2)} \cdots \tilde{x}_k^{(N)}\} = \mathop{\Big\|}_{n=1}^{N} replicate\left(x_k^{(i)}, r_i\right) \tag{3.8}$$

Since the number of times a particle is replicated is directly determined by the replication factor r_i, a fully parallel hardware implementation of resampling is expected to be the most expensive component.

3.1.1 Example filter

All implementations in this chapter are based on the simple particle filter from [2]. This filter implements the tracking of a white object (in this case a square) on a dark background in the presence of noise (see figure 3.7). In the SIRF, both the prediction step and the update step are parameterized by application-specific functions: the prediction step describes the system dynamics for the square while the update step determines new weights based on a new measurement.

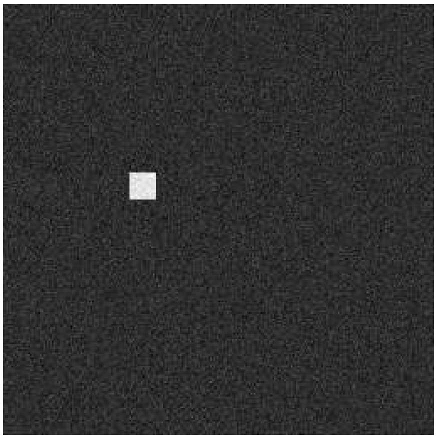


Figure 3.7 – Example of measurement

In the prediction step, the movement of the square is assumed to be uniformly distributed in an area of 32×32 pixels. It follows that the new state can be determined by adding a uniformly sampled deviation to the current position of the square. The state vector now represent an estimate of the position of the square (x, y). Equation 3.9 shows the mathematical formulation of the system dynamics function f.

$$f\left(x_k^{(i)}, u_k\right) = x_k^{(i)} + u_k$$
$$u_k = \langle \delta x, \delta y \rangle \quad \text{where} \quad \delta x, \delta y \sim \mathcal{U}(-16, 16) \tag{3.9}$$

As shown in equation 3.9 the next position $x_{k+1}^{(i)}$ of the square represented by the particle $x_k^{(i)}$ is found by adding u_k to it where u_k is the deviation from the current position. This deviation has a δx and a δy component, both sampled from $\mathcal{U}(-16, 16)$. Therefore the square can at most move 16 pixels in a single filter iteration.

Similar to the prediction step, the update step is also parameterized with a function that determines the importance of a particle given a measurement. For tracking the square, the measurement function g assigns weights to particles given a measurement. As shown before in figure 3.7, this measurement is a single frame containing the white object on a dark background with some noise.

The goal of this simple filter is to track the white object. Therefore, particles within the boundaries of the white object should be assigned a large weight while others should be assigned a lower weight. The function g that implements this behavior is shown in equation 3.10.

$$g\left(x_k^{(i)}, z_k^{(i)}\right) = \frac{1}{1 + (255 - z_k^{(i)}[x, y])^2}$$
$$\text{where} \quad x, y \in x_k^{(i)} \tag{3.10}$$

As shown in equation 3.10, the weight $\omega^{(i)}$ of a particle is determined by the color (a grayscale value between black being 0 and white being 255) of a pixel from measurement $z_k^{(i)}$. From the measurement $z_k^{(i)}$, a pixel is selected positioned at x, y. As shown in equation 3.10, a lighter color will result in a higher weight and vice versa.

3.2 Related work on particle filters

A lot of work on parallelization of particle filter for FPGAs has been performed at Stony Brook University. Their work covers the design of generic hardware architectures [13, 21] and alterations of the resampling step for efficient hardware implementation [23]. Other people have been focusing on the real-time aspects of particle filter implementations [6, 29]. Also FPGA-related implementation details of particle filters have been covered [73, 87]. The following text elaborates on the aforementioned literature and how it compares to the parallelization approach in this chapter. Although this section is not an exhaustive survey on parallel particle filters, the most important approaches to designing parallel particle filters are covered.

At Stony Brook University, generic hardware architectures have been developed [13, 21]. These architectures can be parameterized with application specific modules to implement an actual application. Also parallel processing of particles is included for the prediction and update step to boost performance. The resampling step is performed by a single central module where particles are sent to before being resampled. Also a resampling step optimized for hardware implementation has been proposed called residual systematic resampling (RSR) which can be described by a single for-loop [23]. On hardware, this forms a regular linear structure which will be further exploited in the rest of this thesis.

Particle filters are also used a lot in real-time tracking video applications requiring low latency and high frame rates. Since particle filters are often computationally intensive, optimizations are required before they can be implemented on FPGA. A reduction of the number of particles gives a big performance gain as shown in [29] and [6].

To increase the speed of development, industry-standard system on chip (SoC) design tools have been used to handle communication and synchronization between the particle filter steps. An example is the use of the Altera Avalon bus, a network on chip technology that can be designed using a graphical layout tool [73]. To support dynamic reconfiguration of the particle filter, a Reconfigurable system on chip has been proposed using partial reconfiguration of an FPGA [87]. Parts of the particle filter can be reconfigured at run time which gives both a dynamic advantage and an area advantage. The area advantage comes from the fact that the parts that are changed during reconfiguration can be optimized for the specific application.

Although many design methods, frameworks and optimizations are presented, the step from the mathematical definition to hardware involves a lot of manual translations. However, in this chapter a method is proposed where the amount of manual translations is significantly reduced by first translating the mathematical formulation to Haskell. These translations require only small changes and are therefore less error prone.

3.3 Design method

In order to derive hardware from the equations presented in section 3.1, a two-step design method is proposed. To achieve as much performance as possible, we try to maximize the amount of parallelism. To do so, it is important to preserve the mathematical structure of the filter throughout the whole process. The first step is to derive a Haskell description from the mathematics as given in section 3.1. This Haskell description can now be simulated and acts as a reference for the eventual hardware. The second and final step is reformulating the Haskell description such that it is accepted by the CλaSH compiler. During this final step, parts that involve data-dependent execution and usage of lists are transformed such that they can be translated to hardware.

3.3.1 From mathematics to Haskell

The first step of the design method is the translation of the equations of the particle filter to Haskell. As explained in section 3.1, a particle filtering algorithm consists of four stages: 1. prediction, 2. update, 3. normalization and 4. resampling. Combined with equation 3.9 and equation 3.10, the mathematical definition of the object-tracking filter is complete.

During the prediction step, the position of the object is predicted based on the current estimate and system dynamics. This position is predicted by applying the system dynamics function f to all particles combined with deviation u. The prediction step in Haskell is generic, i.e., the actual system dynamics function f is given as a parameter. Since all particles are combined with deviations in a pair-wise fashion, the higher-order function *zipWith* is used. Listing 3.1 shows how to express the prediction step in Haskell.

```haskell
predict f ps us = zipWith f ps us
```

Listing 3.1 – Prediction step in Haskell

As can be seen in listing 3.1, the prediction step accepts a function f expressing the system dynamics, a list of particles *ps* and a list of random deviations *us* which are produced using a linear feedback shift register (LFSR) [30]. Every particle and every offset is pairwise combined by f using *zipWith* resulting in an list of estimates of the new position.

Translating the system dynamics function f of equation 3.9 into CλaSH is trivial as shown in listing 3.2.

```haskell
f (x,y) (nx, ny) = (x + nx, y + ny)
```

Listing 3.2 – System dynamics function f in Haskell

The results of the prediction step are combined with a measurement in the update step. Again, the update step formulated in equation 3.3 is also generic by leaving the actual update function g as argument. As formulated in equation 3.3, every particle is combined with a single measurement to find the weight for each particle. In Haskell, this structure can be expressed using the higher-order function *map*. Listing 3.3 shows the Haskell definition of the update step.

```haskell
update g z ps = map (g z) ps
```

Listing 3.3 – Update step in Haskell

As can be seen in listing 3.3, the update step accepts three arguments: the update function g, a measurement z and a list of particles *ps*. The update function g combined with a measurement z is applied to all particles *ps* using the higher-order function *map*. Since g is a binary function, i.e., it accepts two arguments, the first argument z is already given to g, resulting in a single parameter function (better known as partial application). This function now only accepts a particle and can therefore be applied to all particles *ps* using the higher-order function *map*.

To complete the translation of the mathematics of the update step, the update function g also has to be translated to Haskell. The Haskell code shown in listing 3.4 is a direct translation of equation 3.10.

```haskell
g z (x,y) = 1 / (1 + (255 -p)^2)
  where
    p = z !! y !! x
```

Listing 3.4 – Update function g in Haskell

As shown in listing 3.4, a pixel containing a grayscale value p is selected from the image z using the index operator !!. Although this is not very efficient in Haskell (this operator has computational complexity $\mathcal{O}(n)$), it will be replaced by a similar operator as soon as hardware is generated. Indexing on hardware is then implemented using a multiplexer which is much more efficient.

Translating the normalization step from equation 3.5 is performed in a similar way as can be seen in listing 3.5.

```haskell
normalize ps = ps'
  where
    omegatot   =   sum (map weight ps)
    ps'        =   map (\ (x,y,omega) -> (x,y,omega/omegatot)) ps
```

Listing 3.5 – Normalization step in Haskell

As shown in listing 3.5, the total weight ω_{tot} is determined by first selecting only the weights of all particles *ps* using the *weight* function. The *weight* function is implemented as $weight(x, y, \omega) = \omega$. All weights are then accumulated in ω_{tot}. In the last line, a lambda expression is applied to all particles *ps* using *map*. The lambda expression accepts a particle and replaces only the weight by the normalized weight.

Remaining to formulate is the resampling step which consists of two parts. First, the replication factor is determined based on the weight of a particle. Thereafter, the actual replication of particles is performed. Equation 3.7 gives an expression to determine the replication factor $r^{(i)}$ for weight $\omega^{(i)}$.

The depth of the recursion in equation 3.7 depends only on the length of the weight list. We use a functional language feature called *pattern matching* ($\omega : \omega s$) to terminate the recursion. Listing 3.6 shows the two phases in the recursion. Either, not all weights have been processed yet (line 1), or the last weight has been processed and an empty list [] is left (line 2). During processing, the list of weights $(\omega : \omega s)$ shrinks every time by taking the first element ω and calculating a replication factor based on that element. Calculation continues recursively with the remainder of the weights ωs until no weights are left [].

```
1 ws2rfs u []             = []
2 ws2rfs u (omega:omegas)  = r : (ws2rfs u' omegas)
3   where
4     r   = floor ((omega-u)*N) + 1
5     u'  = u + r / N - omega
```

Listing 3.6 – Haskell code to determine replication factors

Reformulating the replication of particles in Equation 3.7 to Haskell, comes down to translating replication and the concatenation operator $\|$ to Haskell. Each particle p is replicated $r^{(i)}$ times and all those sets of particles are concatenated into a single set of particles. Listing 3.7 shows the Haskell definition of the replication process.

```
replps ps rs  = concat pss
  where
    pss = zipWith replicate rs ps
```

Listing 3.7 – Replication of particles

As shown in listing 3.7, every particle in *ps* is replicated $r \in rs$ times using *replicate*, i.e., *replicate r p* yields a list of *r* copies of the particle *p*. Since the particles in list *ps* and the replication factors in list *rs* are combined in a pairwise fashion, the higher-order function *zipWith* is used. Note that replicate is a binary function. This results in a list of list of particles *ps* which is concatenated into a single list.

The complete resampling step is formed by composing the function *ws2rfs* and *replps*. All replication factors only depend on the weights ωs; these are extracted from the particles *ps* (first line in the *where* clause of listing 3.8). The resulting list of replication factors *rs* is then used for replication by *replps* (third line in the *where* clause). Finally, the last line replaces the weight by $\frac{1}{N}$ since all particles are of equal importance after resampling.

```
resample ps ws = ps'
  where
    rs      =   ws2rfs ws
    ps_r    =   replps ps rs
    ps'     =   map (\ (x,y,_) -> (x,y,1 / N)) ps_r
```

Listing 3.8 – Complete resampling step in Haskell

```
ws2rfs a omegas  = vscanl rf (0,u0) omegas

rf (u,r) omega = (u',r')
  where
    r'  =   floor ((omega-u)*N)+1
    u'  =   u + frac r N - omega
```

Listing 3.9 – Determining replication factors in CλaSH

3.3.2 From Haskell to Hardware

Now that all components of the particle filter are defined in Haskell, the final step is to adapt the code such that hardware can be generated. For hardware generation, the CλaSH compiler is used which translates Haskell code to VHDL. However, CλaSH does not support all language features available in Haskell. Therefore, language features like lists and recursion have to be removed since they do not result in efficient hardware. In this section, recursion is removed by replacing all list operations by vector operations, i.e., operations acting on lists with a fixed length. Floating point operations are removed as well in order to derive more efficient hardware.

The first step is the replacement of lists by vectors. For most of the steps in the filter this does not change much as most higher-order functions are simply replaced by their vector equivalent functions (for example *map* becomes *vmap*, *foldl* becomes *vfoldl*, etc.). The conversion of the prediction and update step is simply the replacement of *zipWith* and *map* by *vzipWith* and *vmap* respectively. The resampling step, on the other hand, is more complicated to translate since it is expressed using data-dependent list operations.

Similar to the Haskell reference, resampling starts by determining the replication factors, followed by the actual replication of particles. As shown in the Haskell formulation of listing 3.6, determining the replication factors is done using a tail recursive function. Currently, CλaSH does not support recursion. Therefore, this recursion has to be reformulated using functions that are supported by CλaSH. Since the length of the recursion only depends on the amount of particles and the amount of particles is fixed, it can be replaced by a higher-order vector-based function called *vscanl*. Although *scanl* is directly supported in CλaSH using *vscanl*, it is interesting to show the structural correspondence between the mathematical formulation and the resulting hardware. *vscanl* accepts a function *rf*, a starting value u_0, and a vector with weights ωs. The function argument of *vscanl* is applied to each element in the vector while accumulating intermediate values and sending this to the output thus being equivalent with the recursive definition of listing 3.6. The CλaSH implementation of equation 3.7 is shown in listing 3.9. The corresponding hardware structure is shown in figure 3.8.

The actual replication of particles is performed by N parallel multiplexers implemented in CλaSH using the index operator ! (Listing 3.10). Each multiplexer selects

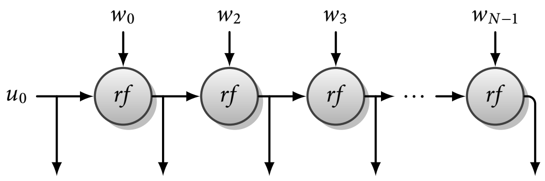


Figure 3.8 – Structural view of *vscanl*

```
replicate ps is = ps'
  where
    ps' = map (ps !) is
```

Listing 3.10 – Replication in CλaSH

a single particle and puts this on the output depending on a list of multiplexer indices calculated from the set of replication factors.

As shown in listing 3.10, the *replicate* functions accepts two arguments, the list of particles *ps* and a list of indices *is*. Given the whole list of particles *ps* and a single index from *is*, a particle is selected using the index operator !. As shown in the CλaSH code, *map* is used to perform the multiplexing using each index in *is* using partial application. This is applied in listing 3.10 at the index operator !, the list of particles is already given since it is used for every index. The ! operator has only a single argument left which is supplied using the *map* function since it is applied to every index in *is*. Figure 3.9 shows the resulting hardware:

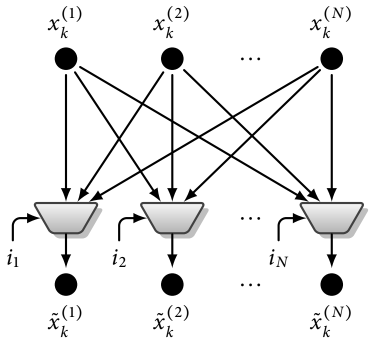


Figure 3.9 – Particle selection

To complete the translation, the conversion from replication factors to indices still has to be implemented. Since CλaSH does not support general recursion, only higher-order functions like *viterate* and *foldl* can be used. Listing 3.11 gives the CλaSH formulation of the function *rs2is* that determines the set of indices.

```haskell
-- convert list of replication factors to indices
rs2is rs = vmap fst (viterate rs2i (rs2i (0,rs)))

-- find next index and decrease a replication factor
rs2i (ind, rs) = (ind', rs')
  where
    ind' = nextreplin ind rs
    rs' = vreplace rs ind' (rs ! ind' - 1)

-- find index of first replication factor>0 at index >= io
nextreplin io rs = snd res
  where
    res = vfoldl (frepl io) (False, 0) rs
    frepl io (found, cind) r = if found
                               then (True, cind)
                               else  if cind >= io && r>0
                                     then (True, cind)
                                     else (False, cind+1)
```

Listing 3.11 – Conversion from replication factors to indices.

To determine the set of indices to select particles, the function *rs2is* is used. Using *viterate*, a set of tuples containing an index and an altered list of replication factors is produced. From these tuples, only the indices are selected using the function *fst* (select the first element of a tuple). *viterate* repeatedly applies *rs2i* which determines a single index given the current index and list of replication factors. For every application of *rs2i*, a new index is found and a replication factor in *rs* is decreased by one. In total *rs2i* is instantiated N times, equal to the sum of all replication factors.

rs2i accepts and returns a tuple with and index and a list of replication factors. This function consists of two steps; determining an index using the function *nextreplin* and decreasing the replication factor with index *ind*. The replication factor with index *ind* is replaced by a new replication factor that is one less than the old value.

Finally, *nextreplind* determines the next index where the replication factor is bigger bigger than zero with offset *io*. This is implemented using a *foldl* parameterized using the function *frepl*. For every instantiation of *frepl* in *foldl*, the next replication factor bigger than zero is either found or not. When it is found, the current index *cind* is simply forwarded to the end of *foldl*. When this is not the case, the current index is increased for the next step and *found* remains false.

3.4 Results

In the previous sections, the mathematical definition of a tracking particle filter has been reformulated in Haskell. Using this Haskell program, the minimum number of particles for this particular application has been determined to be 32 particles using simulation. The number of particles have been reduced until the particle filter

Table 3.1 – Area of components

Component	Slice LUTs
Prediction	704
Update	954
Normalization	1402
Resampling	35978
Total	39038

was not able to track the object anymore. Additionally, all recursive functions have been rewritten using higher-order functions, e.g., *vzipWith* and *vscanl*, such that the code can be compiled by the CλaSH compiler. The resulting CλaSH description was tested to have the same external behavior as the reference description, using a testbench that was used to verify the functional correctness of the Haskell program. This testbench contains a set of images where a white square follows a Lissajous curve. The feasibility of the parallel CλaSH implementation has been determined by synthesizing the design for a Xilinx Virtex 6 FPGA (XC6VLX240T). An overview of the resource usage for the different parts of the particle filter is shown in table 3.1.

In addition to the LUT results of table 3.1, 1116 slice registers have been used. In terms of performance, the synthesized particle filter achieves a throughput of 24 million particles per second. However, fully parallel resampling uses a lot of FPGA area and is therefore the biggest bottleneck in this design: due to all data dependencies in the resampling step, all possible replications have to be considered, resulting in the large area and low clock frequency (around 1 MHz). The reason for this very low operating frequency is the long combinatorial path in the resampling step of the particle filter. The resampling step has a long combinatorial path because there are many data dependencies within the mathematical specification, and hence also within the parallel implementation. Although the particle filter has not been implemented in VHDL directly, similar resource consumption is expected based on results in [84].

3.5 Conclusions

In this chapter, a two-step step design methodology is proposed to derive hardware from a mathematical specification of a particle filter. Using this design methodology the available parallelism in the mathematical definition is preserved as much as possible. We started with a formulation from mathematics to Haskell, resulting in a design that closely matches the mathematical description of a particle filter and can be simulated by executing a single function. To be able to generate hardware using the CλaSH compiler, some adaptations to the Haskell code had to be performed to remove the use of lists and recursion. All lists have to be replaced by vectors and recursion has to be implemented using statemachine or using HOFs.

The results show that deriving a large amount of parallelism is relatively straightforward using the methodology described in this chapter. In particular, the use of higher-order functions gives the designer an intuitive way of describing structure available in the mathematical definition of the application and the resulting hardware generated later on in process. First formulating the mathematics of the particle filter in plain Haskell results in a model that can easily be simulated. This can then be used as a reference for the CλaSH implementation. Since only relatively small adaptations have to be made to the Haskell reference, the verification of the implementation is straightforward.

Hardware is generated for a filter with 32 particles. All particles are processed in parallel such that a complete iteration of the filter lasts only one clock cycle. The VHDL code generated by CλaSH has been successfully synthesized for an FPGA showing the feasibility of the approach. However, synthesis results show that the resampling step requires a lot of area and has a very long combinatorial path resulting in a very low clock frequency.

Although the proposed design method shows a clear path from mathematics to hardware, a fully parallel approach requires a lot of hardware and therefore severely limits the amount of applications covered by the design methodology. In order to support more complex applications, a trade-off is needed to control the amount of parallelism. In the following chapters, the design methodology is altered such that a trade-off between execution time and area can be made.

4 Trade-off rules

Abstract – *In this chapter, several trade-off rules are presented to distribute computations of higher-order functions over time and space. During transformation a mathematically equivalent higher-order function is derived to enable an efficient mapping to FPGA hardware. For modeling of applications, an embedded language with types for space and time is introduced which should prevent the erroneous composition of functions. To be able to design hardware using the transformation rules, a CλaSH library has been developed with components in which the transformation rules have already been applied to commonly used higher-order functions. This library is used to implement a small case study, a dotproduct operation.*

As expressed in chapter 3, higher-order functions are a powerful abstraction to express structure of hardware. Since list-based higher-order functions are very regular in structure, they are a good candidate for transformations. Using these transformations, a complex calculation can be split into smaller parts and then combined. However, (FPGA) area usage has to be taken into account as well since a fully parallel approach requires a lot of hardware (FPGA) resources.

Splitting up these computations can be implemented using transformation rules where higher-order functions of the original specification are replaced by a description that performs computations over space and time. In this chapter, several transformation rules are proposed based on commonly used higher-order functions in Haskell and CλaSH. These transformations take area into account in contrast to the fully parallel approach of chapter 3. A similar approach is taken in Squigol [18] where transformations for speed optimizations are presented. More general, the transformation-based approach presented in this thesis is related to [17] where parts of the program are rewritten into equivalent descriptions as well.

The transformation rules proposed in this chapter transform higher-order functions into a new description containing three parts. The first part in the description

Large parts of this chapter have been published in [RW:4, 5].

resulting from the transformation is a list split into smaller sublists for the distribution of data. The second part performs the actual computations where each sublist is processed one-by-one while a complete sublist is processed at once (fully parallel). In the third and final part, depending on the higher-order function under transformation, the data from the computations of the sublists is concatenated into a single list again. Although higher-order functions cover far more applications than processing lists, the transformations presented in this chapter are limited to lists as these are the underlying datastructure for the applications covered in this thesis.

In section 4.1, the transformation rules for the higher-order functions *zipWith* and *foldl* are introduced while section 4.2 presents proofs that the transformed functions remain mathematically equivalent. In section 4.3 a shallow embedded language is introduced to give some type safety when modeling applications using the transformation rules. A CλaSH library with hardware components is presented in section 4.3 as well. To show how this library can be used in practice, an example is presented in section 4.4 in the form of a dotproduct operation. Finally in section 4.5, conclusions are drawn about our transformation-based design methodology.

4.1 Rewriting Higher-Order Functions

By rewriting higher-order functions, computations can be distributed over space and time. In this section two commonly used higher-order functions, *zipWith* and *foldl*, will be mapped over space and time. Table 4.1 shows an overview of commonly used higher-order functions and their structure on hardware.

Rewriting zipWith

A commonly used higher-order function in Haskell is *zipWith*. Using *zipWith*, two lists are pairwise combined using a function resulting in a single new list. Mapping *zipWith* completely to space results in a lot of resource consumption as there are as many instantiations of f as there are elements in the list. Figure 4.1 shows the structure of the higher-order function *zipWith*.

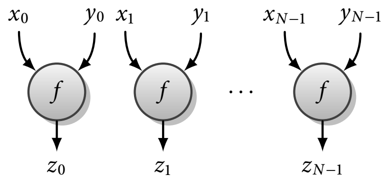
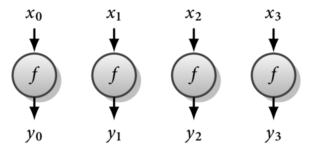
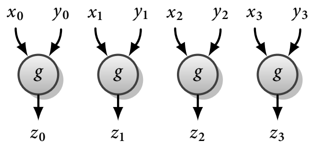
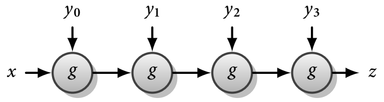
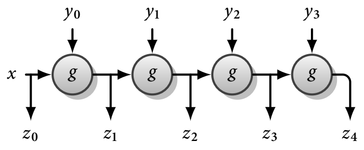
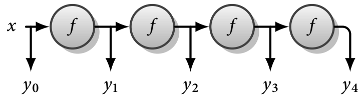


Figure 4.1 – Structure of $zs = zipWith\ f\ xs\ ys$

As shown in figure 4.1, both list xs and list ys contain N elements. When N is divisible by P, the following holds: $N = P \times Q$ where $Q = \frac{N}{P}$. It follows that both

Table 4.1 – Commonly used higher-order functions

Name	Structure	Haskell
map		*ys = map f xs*
zipWith		*zs = zipWith g xs ys*
foldl		*z = foldl g x ys*
scanl		*zs = scanl g x ys*
iterate		*ys = iterate f x*

xs and *ys* can be divided into Q sublists, each containing P elements. Note that lists which are not divisible by P can be split as well but this introduces a small overhead since the last sublist to process is not complete. The division is depicted in figure 4.2.

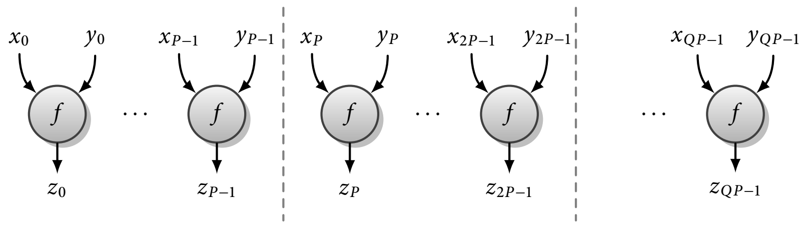


Figure 4.2 – Splitting of *zs = zipWith f xs ys*

As shown in figure 4.2, each sublist contains P elements, numbered from 0 to $P-1$. After this division and renumbering, a new Haskell definition of figure 4.2 can be formulated. The initial list *xs* is divided into smaller lists with length P using the function *split* such that *split* $2\ [1,2,3,4] = [[1,2],[3,4]]$. For each sublist, the structure can still be formulated using the higher-order function *zipWith*. The combination of these sublists is also performed using *zipWith*. *zipWith* is now expressed in terms of itself. Listing 4.1 shows the new definition of *zipWith*.

```
zipWith' p f xs ys = zs
    where
        xss = split p xs
        yss = split p ys
        zss = zipWith (zipWith f) xss yss
        zs  = concat zss
```

Listing 4.1 – New formulation of *zipWith*

As shown in listing 4.1, the input lists *xs* and *ys* are divided into smaller lists of *P* elements using the split function resulting in a list of lists of x elements and y elements named *xss* and *yss* respectively. The third line in the where-clause shows how the list of lists is processed by two *zipWith* instantiations. Finally, the last line in the where-clause concatenates all sublists from the *zipWith* instantiations. The result type of both *zipWith'* and *zipWith* are now the same.

Even though this rewrite rule is a way to divide the problem into smaller problems, it does not yet incorporate parallelism. Therefore, parts of the computations have to be performed in space ($\mathbb{S}$), i.e., in parallel and parts over time ($\mathbb{T}$), i.e., sequential. By selecting specific mappings of *zipWith*, listing 4.1 can be formulated as follows:

$$zss = zipWith_{\mathbb{T}}\ (zipWith_{\mathbb{S}}\ f)\ xss\ yss \tag{4.1}$$

In the expression $zs = zipWith_{\mathbb{T}}\ f\ xs\ ys$, every pairwise computation is executed sequentially. This has minimal hardware requirements since only one instantiation of f is needed but consequently also the largest execution time. The expression $zs = zipWith_{\mathbb{S}}\ f\ xs\ ys$ defines the exact opposite since all computations are performed in parallel resulting in maximum area usage. Consequently, assuming that f can be executed in single clock cycle, only one clock-cycle is required for the whole computation. The ordering can also be reversed by processing the elements in the sublists sequentially but performing this on several sublists at the same time:

$$zss = zipWith_{\mathbb{S}}\ (zipWith_{\mathbb{T}}\ f)\ xss\ yss \tag{4.2}$$

In the first example, all elements in a sublist are processed in parallel while all sublists are processed one by one, i.e., sequential. The number of elements processed in parallel is determined by the width of the sublists generated by *split*. This width is controlled by the parameter P and therefore also determines the resource usage. Increasing P increases the amount of parallelism but also the resource usage ($P \times R(f)$ where $R(f)$ represents the amount of resources for an instantiation of f). Decreasing P, on the other hand, reduces parallelism and resource usage at the cost of a higher execution time to process the whole input list. The number of cycles Q to exec the whole expression can be found by $Q = \frac{N}{P}$. Figure 4.3 shows a general depiction of the aforementioned distribution of computations over space and time while Figure 4.4 shows it specifically for *zipWith* as expressed in equation 4.1.

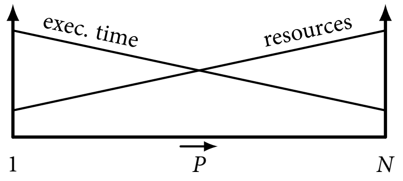


Figure 4.3 – Trade-off between resource consumption and execution time

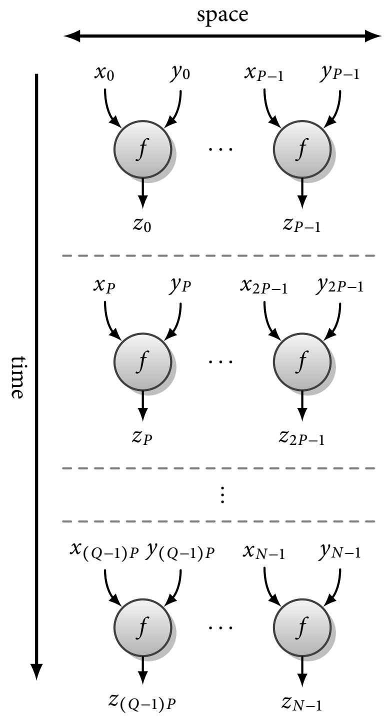


Figure 4.4 – $zs = zipWith\ f\ xs\ ys$ distributed over space and time

During each time step, a pair of sublists each containing P elements is processed completely. This process is repeated until all Q sublists are processed. In terms of communication, P elements are required to be available each iteration. Although a larger value for P results in more parallelism, it also requires more bandwidth. Figure 4.5 shows the resulting architecture which will be put on an FPGA.

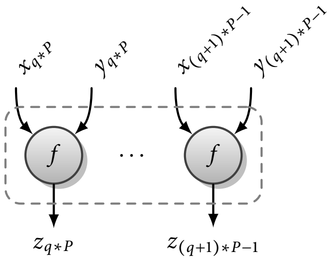


Figure 4.5 – FPGA architecture of *zs* = *zipWith f xs ys*

As shown in figure 4.5, all sublists are sequentially fed into the architecture. At the same time, the results are sent to the output. For a complete list, this process is executed for Q time steps, when each sublist is executed in one time step. Note that the reuse of f over time is only allowed when f contains no state. Since f is defined as a pure function in Haskell, this is always the case.

Rewriting foldl

Another commonly used higher-order function is *foldl*. Using *foldl* a list is incrementally processed given some starting value. Other than *zipWith*, *foldl* forms a chain where all subsequent nodes depend on previous nodes. A fully parallel mapping to FPGA therefore not only requires potentially a lot of area but also introduces a long combinatorial path which reduces the maximum achievable clock frequency. Distributing *foldl* over space and time therefore not only reduces the amount of resource usage but also the length of the longest combinatorial path. Figure 4.6 shows the structure of *foldl*.

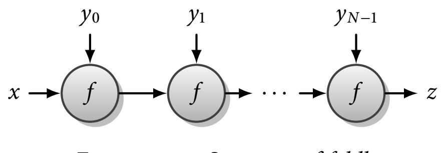


Figure 4.6 – Structure of *foldl*

As shown in figure 4.1, the list *ys* contains N elements. Similar to *zipWith*, when N is decomposed as: $N = P \times Q$. *ys* can be divided into Q sublists, each containing P elements. Such a division is depicted in figure 4.7.

Figure 4.7 shows that *ys* is split into sublists each containing P elements. After division and renumbering, the structure of figure 4.7 can be formulated in Haskell. Similar to *zipWith*, each sublists is still processed using the higher-order function *foldl*. Also on a larger scale, the lists are processed using *foldl*. A *foldl* can therefore be expressed in terms of itself. Listing 4.2 shows this new formulation of *foldl*.

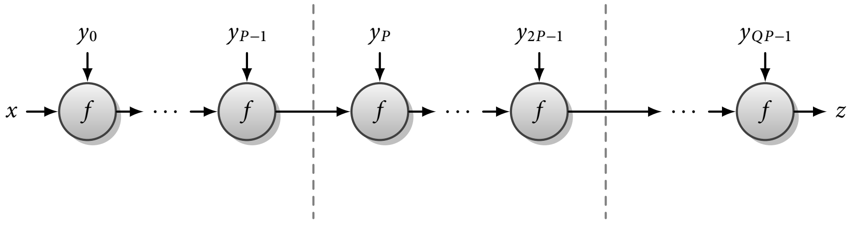


Figure 4.7 – Splitting of *foldl*

```
foldl’ p f x ys = z
    where
        yss = split p ys
        z   = foldl (foldl f) x yss
```

Listing 4.2 – New formulation of *foldl*

As shown in listing 4.2, the list *ys* is first split into smaller lists with P elements using the *split* function. The last line shows how this list of lists is processed using two instantiations of *foldl*. Note that no concatenation of elements is needed since *foldl* reduces a list to a single value.

By choosing specific implementations of *foldl*, the computations can be mapped partially over space and partially over time. Using these specific implementations, a trade-off is made between area and execution time. Note that *foldl* is in principle a fully sequential operation and the transformation rules do not alter any dependencies among operations. An example a trade-off between execution time and area can be formulated as:

$$z = foldl_{\mathbb{T}}\ (foldl_{\mathbb{S}}\ f)\ x\ yss$$

Similar to the transformation rules for *zipWith*, the ordering can be reversed by processing the elements in the sublists sequentially but performing this on several sublist at the same time:

$$z = foldl_{\mathbb{S}}\ (foldl_{\mathbb{T}}\ f)\ x\ yss$$

In the first example, all elements in a sublist are now processed in parallel while all sublists are processed one by one, i.e., sequential. The amount of parallelism is determined by the size of the sublist, which is parameter P of the *split* function. This parameter therefore also determines the amount of resource usage and the length of the longest combinatorial path. Figure 4.8 shows the aforementioned distribution of computations over space and time graphically.

During each time step, a sublist containing P elements is processed completely. This process is repeated until all Q sublists are processed. Every time a partial result

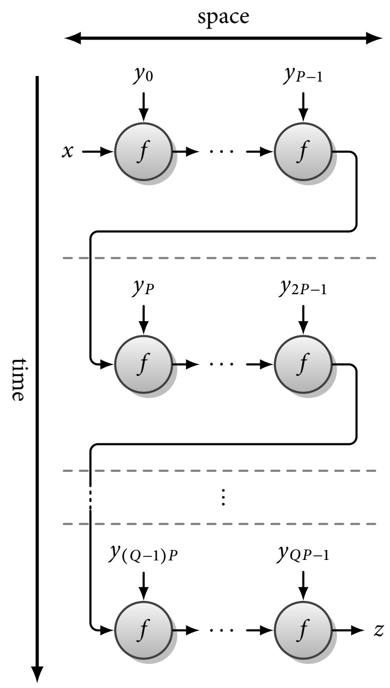


Figure 4.8 – *foldl* distributed over space and time

is stored for the next cycle until eventually the results z is produced. Figure 4.9 shows the resulting architecture which will be put on an FPGA.

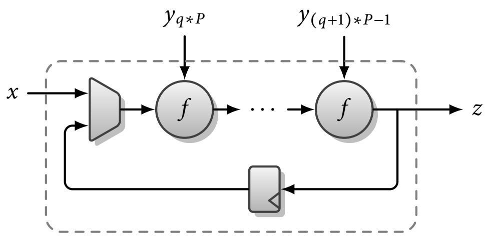


Figure 4.9 – FPGA architecture of *foldl*

As shown in figure 4.9, all sublists are sequentially fed to the architecture. During the first cycle, x is used as starting value while the value from the register is used during all subsequent cycles. To process a whole list, at least Q cycles are required.

Rewriting other HOFs

Besides *zipWith* and *foldl*, a plethora of higher-order function exist that are useful for hardware design. In particular *map*, *iterate* and *scanl* are used in the designs presented in this thesis. The transformation rules for these higher-order functions follow the same procedure as *zipWith* and *foldl*. Each higher-order function can also be transformed into different, mathematically equivalent structures. Depending on the requirements of the target hardware, a different transformation rule can be chosen. In this section, two new transformation rules for *foldl* are presented resulting in different hardware structures.

The *foldl* transformation rule presented in this section leaves the chain structure unchanged. When the function parameter g is *commutative* ($g\ a\ b = g\ b\ a$ for example $a + b = b + a$) and the initial value x is the *identity element* of g ($g\ x\ b = b$ for example $0 + b = b$), the chain structure can be altered such that hierarchies can be introduced. Using the aforementioned properties, the sequential chains in *foldl* can be broken and $zs = foldl\ g\ x\ ys$ can be transformed into the following:

$$y = foldl\ g\ x\ (map\ (foldl\ g\ x)\ yss)$$

Every sublist in *yss* is reduced to a single value using the right *map* and *foldl*. These intermediate results are reduced to a single value z using the left *foldl* again. Figure 4.10 shows the corresponding structure.

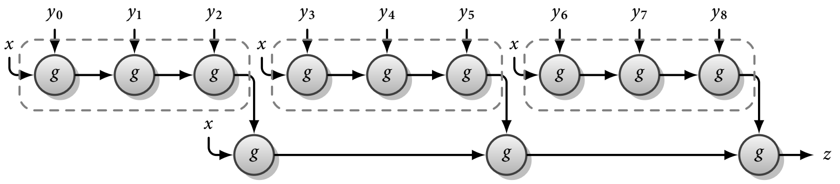


Figure 4.10 – Alternative transformation of *foldl*

Figure 4.10 shows a partitioning into three subproblems before the final result is determined. Similar structures are used in map-reduce applications where subproblems are parallelized (mapped) over several machines and then reduced to a single result [37].

Similarly, the previous example can be rewritten such that *map* is removed and an additional *foldl* is introduced. Note that g is still assumed to be commutative and x is still the identity element. A second alternative for *foldl* now becomes:

$$y = foldl\ g\ x\ (foldl\ (zipWith\ g))\ [x, x, x]\ yss$$

Similar to the previous example, the final result is determined by a *foldl*. The intermediate values however, are determined by incrementally processing sublist from *yss*. Figure 4.11 shows the corresponding structure.

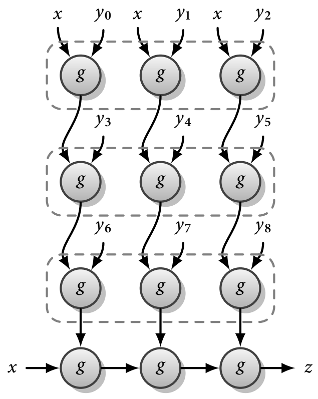


Figure 4.11 – VLIW transformation of *foldl*

Figure 4.11 shows how the sublist from *yss* are incrementally processed using a *foldl* in a downwards direction. The elements in each sublist are combined with the binary function *g* using the the higher-order function *zipWith* (horizontal). By executing the grouped parts sequentially from top to bottom and the horizontal parts in space then a schedule arises that can be efficiently executed on a very large instruction word (VLIW) machine [57].

So, by rewriting the initial specification *foldl f x ys* we can derive various mathematically equivalent versions of the same specification (figure 4.6, figure 4.8, figure 4.10 and figure 4.11). The designer can select the most optimal specification for his or her hardware architecture.

4.1.1 Composition using dataflow

Although parallelism and therefore resource consumption and execution time can now be controlled using a parameter for each higher-order function, two new problems appear when composing these functions: synchronization of data and scheduling of computations. Synchronization of data ensures that the correct data is available when required while ordering entails the execution of operations over time. To solve these two problems, dataflow principles are used [71]. Due to the data-dependent production of tokens in the resampling step, no static scheduling can be performed.

Using dataflow, computations are triggered by the availability of data. Applications are modeled as a graph. The nodes contain the computations while data is sent over edges in the form of tokens. When sufficient data is available on all incoming edges of a node, all required data is available and a computation in that node can start. This state of a node is called *enabled* and the condition for execution is called *firing rule*. During execution, data at the inputs are consumed and used as argument.

After computation, the nodes produce a token with the result which is put onto the output edge. Figure 4.12 shows the execution phases of a simple dataflow graph.

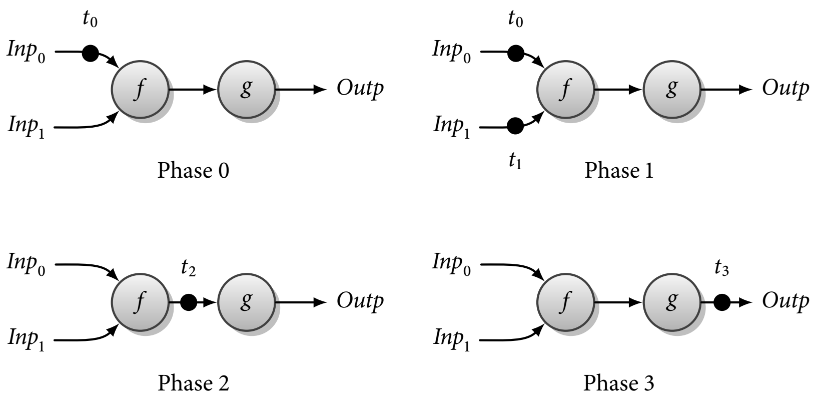


Figure 4.12 – Dataflow firing rule

As shown in figure 4.12, in phase 0 at the inputs of node f one token has arrived. Since Inp_1 has not yet received a token, the firing rule is not satisfied. In phase 1, a token has arrived and execution of f can start. Once completed, a token t_2 is produced and sent to the next node (Phase 2). Node g has only one input and is therefore enabled as soon as a token is available on the input. In phase 3, the execution of g has produced another token at the output.

Formally, an edge of a dataflow graph can contain an infinite number of tokens. In practice, tokens are usually stored using first-in-first-out buffers (FIFOs) with a fixed length. Therefore, the firing rule should not only take into account the availability of data on the input, but also whether there is space available in the FIFOs on the output. Formally, FIFOs are often modelled as a double-edge [114] where tokens on the back-edge represent available positions in the FIFO. Figure 4.13 shows a dataflow graph where the edges are implemented using FIFOs. A black dot in the FIFO indicates the presence of a token.

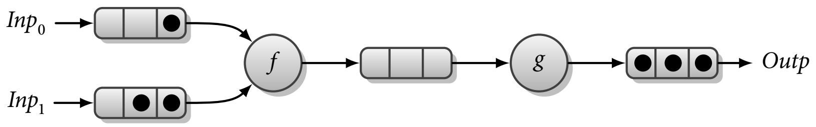


Figure 4.13 – Dataflow graph with FIFOs

By wrapping the transformed higher-order functions in dataflow nodes and extending the firing rule, synchronization and scheduling is handled by dataflow.

However, the input list(s) for a node has to be distributed over space and time as well. A list distributed over space and time is represented by a sequence of tokens where each token contains a whole sublist. First, a whole list is split into smaller sublists. Each sublist is packed in a single token which are sent sequentially to the node for processing. Tokens are therefore processed sequentially while the sublist in the token can be processed in parallel. Figure 4.14 shows the process of dividing a list into a sequence of tokens.

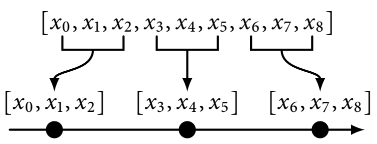


Figure 4.14 – From list to tokens

4.2 Proofs of Equivalence

According to referential transparency, an expression in Haskell can be replaced by a mathematically equivalent expression without changing its functionality. To be able to do this with the transformation rules as well, we must first prove that these expressions are indeed equivalent. In this section, proofs for *zipWith* and *foldl* are presented showing the equality between the higher-order function before and after the transformation.

4.2.1 Equivalence proof of zipWith

As formulated in listing 4.1, the transformation rule for *zipWith* states that $zipWith\ f\ xs\ ys = concat\ (zipWith (zipWith\ f)\ (split\ p\ xs)\ (split\ p\ ys))$. The remainder of this section shows the proof using a mathematical reformulating of *zipWith*.

Definition 1. Mathematically, $zs = zipWith\ \star\ xs\, ys$ can be defined as $zs = xs \mathbin{\widehat{\star}} ys$ where $\star$ is a binary function.

Definition 2. Assuming that lists *xs* and *ys* have the same length, $\widehat{\star}$ is recursively defined as:

$$[\,] \mathbin{\widehat{\star}} [\,] = [\,]$$
$$(x : xs) \mathbin{\widehat{\star}} (y : ys) = (x \star y) : (xs \mathbin{\widehat{\star}} ys)$$

where $[\,]$ is the empty list and : is the operator that puts an element in front of a list.

Definition 3. Using the same recursive structure, the concatenation function *concat* used to merge a list of lists into a single list is defined as:

$$concat\ [\,] = [\,]$$
$$concat\ xs : xss = xs +\!\!+\ concat\ xss$$

where $+\!\!+$ merges two list into a single list.

Lemma 4.

$$(as \mathbin{\widehat{\star}} bs) \mathbin{+\!\!+} (cs \mathbin{\widehat{\star}} ds) = (as \mathbin{+\!\!+} cs) \mathbin{\widehat{\star}} (bs \mathbin{+\!\!+} ds)$$

Proof. Assuming that *length as* = *length bs* and *length cs* = *length ds* this lemma can be proven using induction on lists *as* and *bs*:

First, let $as = []$ and $bs = []$:

$$\begin{aligned}([] \mathbin{\widehat{\star}} []) \mathbin{+\!\!+} (cs \mathbin{\widehat{\star}} ds) = [] \mathbin{+\!\!+} (cs \mathbin{\widehat{\star}} ds) &= (cs \mathbin{\widehat{\star}} ds)\\ ([] \mathbin{+\!\!+} cs) \mathbin{\widehat{\star}} ([] \mathbin{+\!\!+} ds) &= (cs \mathbin{\widehat{\star}} ds)\end{aligned}$$

then, by using the induction hypothesis:

$$\begin{aligned}((a : as) \mathbin{\widehat{\star}} (b : bs)) \mathbin{+\!\!+} (cs \mathbin{\widehat{\star}} ds) &= (a \star b) : ((as \mathbin{\widehat{\star}} bs) \mathbin{+\!\!+} (cs \mathbin{\widehat{\star}} ds))\\ &\overset{IH}{=} (a \star b) : ((as \mathbin{+\!\!+} cs) \mathbin{\widehat{\star}} (bs \mathbin{+\!\!+} ds))\\ &\overset{\text{def}\widehat{\star}}{=} (a : (as \mathbin{+\!\!+} cs)) \mathbin{\widehat{\star}} (b : (bs \mathbin{+\!\!+} ds))\\ &= ((a : as) \mathbin{+\!\!+} cs) \mathbin{\widehat{\star}} ((b : bs) \mathbin{+\!\!+} ds) \qquad \square\end{aligned}$$

Theorem 5. $concat(xss \mathbin{\widehat{\widehat{\star}}} yss) = (concat\ xss) \mathbin{\widehat{\star}} (concat\ yss)$

Proof. By induction on both *xss* and *yss* assuming the *xss* and *yss* have the same length:

First, let $xss = []$ and $yss = []$:

$$\begin{aligned}concat([] \mathbin{\widehat{\widehat{\star}}} []) &= concat\ [] = []\\ (concat\ []) \mathbin{\widehat{\star}} (concat\ []) &= [] \mathbin{\widehat{\star}} [] = []\end{aligned}$$

then, by using the induction hypothesis:

$$\begin{aligned}concat\ ((xs : xss) \mathbin{\widehat{\widehat{\star}}} (ys : yss)) &\overset{\text{def}\widehat{\widehat{\star}}}{=} concat\ ((xs \mathbin{\widehat{\star}} ys) : (xss \mathbin{\widehat{\widehat{\star}}} yss))\\ &\overset{\text{def } concat}{=} (xs \mathbin{\widehat{\star}} ys) \mathbin{+\!\!+} concat(xss \mathbin{\widehat{\widehat{\star}}} yss)\\ &\overset{\text{IH}}{=} (xs \mathbin{\widehat{\star}} ys) \mathbin{+\!\!+} ((concat\ xss) \mathbin{\widehat{\star}} (concat\ yss))\\ &\overset{\text{L4}}{=} (xs \mathbin{+\!\!+} concat\ xss) \mathbin{\widehat{\star}} (ys \mathbin{+\!\!+} concat\ yss)\\ &\overset{\text{def } concat}{=} (concat\ xs : xss) \mathbin{\widehat{\star}} (concat\ ys : yss) \qquad \square\end{aligned}$$

4.2.2 Equivalence proof of foldl

As formulated in section 4.1, the transformation rule for *foldl* states that $foldl\ f\ x\ ys = foldl\ (foldl\ f)\ x\ (split\ p\ ys)$. The remainder of this section shows the proof using a mathematical reformulating of *foldl*.

Definition 6. Mathematically, $z = foldl\ f\ x\ ys$ is defined as: $z = x \circledast ys$

Definition 7. $\circledast$ is recursively defined as:

$$\begin{aligned} x \circledast [\,] &= x \\ x \circledast (y : ys) &= (x \star y) \circledast ys \end{aligned}$$

Lemma 8.

$$x \circledast (ys + \!\! + zs) = (x \circledast ys) \circledast zs$$

Proof. By induction on the list ys:

First, let $ys = [\,]$:

$$\begin{aligned} x \circledast ([\,] + \!\! + zs) &= x \circledast zs \\ (x \circledast [\,]) \circledast zs &= x \circledast zs \end{aligned}$$

then, by using the induction hypothesis:

$$\begin{aligned} x \circledast ((y : ys) + \!\! + zs) &= x \circledast (y : (ys + \!\! + zs)) \\ &\overset{def\,\circledast}{=} (x \star y) \circledast (ys + \!\! + zs) \\ &\overset{IH}{=} ((x \star y) \circledast ys) \circledast zs \\ &\overset{def\,\circledast}{=} (x \circledast (y : ys)) \circledast zs \end{aligned}$$

□

Lemma 9.

$$x \circledcirc yss = x \circledast (concat\ yss)$$

Proof. By induction on the list yss:

First, let $yss = [\,]$:

$$x \circledcirc [\,] \overset{def\,\circ}{=} x \overset{def\,\circledast}{=} x \circledast [\,] = x \circledast (concat\ [\,])$$

then, by using the induction hypothesis:

$$\begin{aligned} x \circledcirc (ys : yss) &= (x \circledast ys) \circledcirc yss \\ &\overset{IH}{=} (x \circledast ys) \circledast (concat\ yss) \\ &\overset{L8}{=} x \circledast (ys + \!\! + concat\ yss) \\ &= x \circledast (concat(ys : yss)) \end{aligned}$$

□

Theorem 10.

$$x \circledcirc (split_P\ ys) = x \circledast ys$$

Proof. Using the fact that

$$concat\ (split_P\ ys) = ys$$

it follows that:

$$x \circledcirc{\star}\ (split_P\ ys) = x \circledast\ (concat(split_P\ ys)) = x \circledast\ ys \qquad \square$$

4.3 Embedded Language for type-safe composition

To model applications to which these trade-off rules can be applied, a small embedded language has been developed. Both time and space types are introduced to express which operations are mapped to space or time. During composition, these types help the designer to prevent making erroneous compositions, i.e., composing time and space directly. A shallow embedded language is implemented using Haskell purely for modelling. To design actual hardware, a CλaSH library is developed where higher-order function components can be parameterized with functions and composed using dataflow.

4.3.1 Embedded language with space and time types

As shown in section 4.1, the transformation rules use *split* to distribute data over space and time. Similarly, higher-order functions are executed over time or space. Therefore, space and time types are introduced to prevent erroneous composition of these functions.

When a list is mapped completely over time, all elements arrive sequentially. To express this in the type of the list, a new type *LstTime* is defined. The definition and use of *LstTime* is shown in listing 4.3.

```haskell
type LstTime a = [a]

xs_t :: LstTime Int
xs_t = [1,2,3]
```

Listing 4.3 – Type for mapping list over time

As shown in figure 4.15, all elements in the list are represented as separate dataflow tokens. These tokens arrive in the ordering as is present in the list, e.g., 1 before 2, and 2 before 3.

In contrast to processing all elements in a list purely sequential, they can also be processed completely in parallel by sending them all at once. For this completely spatial distribution, the type *LstSpace* is introduced. In terms of hardware, all elements can be found as distinct components on an FPGA. Listing 4.4 shows the definition of *LstSpace* in Haskell and how it can be used in a definition of a list.

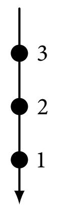


Figure 4.15 – Visualization of *LstTime*

```haskell
type LstSpace a = [a]

xs_s :: LstSpace Int
xs_s = [1,2,3]
```

Listing 4.4 – Type for mapping list over space

As shown in figure 4.16, all elements in the list are represented as a single token. In this token, all elements are present in parallel. The node receiving this token can therefore also process all elements in parallel.

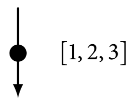


Figure 4.16 – Visualization of *LstSpace*

The types are particularly useful when using data that is distributed over space and time. By nesting the types *LstTime* and *LstSpace*, the ordering in time and in space of lists can be expressed. Using the function $split_p$, a list is split into P sublists. Depending on the variant of $split_p$, data is partially mapped over time and partially over space. Listing 4.5 shows the type of a list distributed over space and time.

```haskell
xs = [1,2,3,4,5,6]

-- xs_st = [[1,2], [3,4], [5,6]]
xs_st :: LstTime (LstSpace Int)
xs_st = split 2 xs
```

Listing 4.5 – Types of list distributed over space and time

The code from listing 4.5 is shown visually in figure 4.17. The whole list $[1 \ldots 6]$ is split into sections of two elements. These sections are tokens, thereby forming a stream of three tokens each containing two elements in parallel. The dataflow node consuming these tokens therefore can process two elements in parallel.

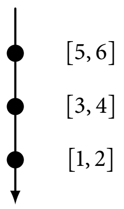


Figure 4.17 – Combing types *LstSpace* and *LstTime*

The types for the aforementioned list examples also apply to the data arguments of higher-order functions like *map*. All computations in *map* can be either fully sequential, fully parallel or a combination of the two using a transformation rule. The input data should be ordered in the same way which can be performed using annotations with the same types. Listing 4.6 shows the types of *map* distributed over either space or time.

```
map_s :: (a -> b) -> LstSpace a -> LstSpace b
map_s = map

map_t :: (a -> b) -> LstTime a -> LstTime b
map_t = map
```

Listing 4.6 – Types of *map* distributed over either space or time

In figure 4.18, a fully parallel and a fully sequential instantiation of *map* are shown. Let $f = +1$, then all additions are either processed in parallel (as shown on the left), or all sequential (the right example).

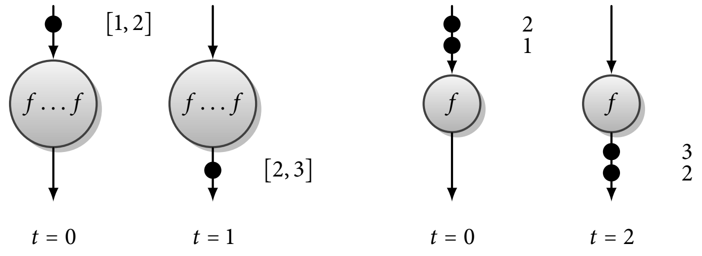


Figure 4.18 – Types of *map* over space or time

When *map* is transformed using a transformation rule and assuming that the input list is already mapped over space and time, the type of the input list should reflect this mapping. Since the input list is distributed over space and time and no accumulation of tokens is performed, the output of *map_st* has the same type as the input (*LstTime (LstSpace a)*). The example of *map* distributed over both space and time is shown in listing 4.7.

```
map_st :: (a -> b) -> LstTime (LstSpace a) -> LstTime (LstSpace b)
map_st f xss = yss
        where
                yss = map_t (map_s f) xss
```

Listing 4.7 – Types of *map* distributed over both space and time

Figure 4.19, shows the visualization of the tokens as expressed using type in listing 4.7. When assuming $f = +1$, every token contains two elements which are processed in parallel (type *LstSpace*). All tokens, however, are sent sequentially (*LstTime*). Combing these results in the composed type *LstTime (LstSpace Int)*.

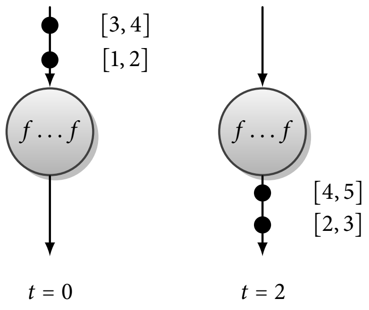


Figure 4.19 – Types of *map* distributed over space and time

To recap, the types *LstSpace* and *LstTime* are introduced in a shallow embedded language to prevent the erroneous composition of components. *LstTime* represents a list type where all elements are processed sequentially while *LstSpace* represents the fully parallel processing of elements. A combination of theses two list types can be used to represent data and computations that are distributed over both space and time. Implementation details of this shallow embedded language can be found in appendix A.

4.3.2 CλaSH library with space and time types

To be able to use the trade-off rules for actual hardware, a CλaSH library has been developed. In this library, several higher-order functions can be found to which the corresponding trade-off rules have been applied. These higher-order functions can be instantiated as hardware components and parameterized as a normal higher-order function. For composition and synchronization, dataflow principles are included. A complete application is implemented by instantiating the higher-order functions and composing them using FIFOs. Listing 4.8 shows the implementation of the *zipWith* component.

```
zipWith_st ::   (a -> b -> c) ->
                (Bool, Bool, Bool, VecTime (VecSpace n a), VecTime (VecSpace
                     n b))
                -> (Bool, Bool, Bool, VecTime (VecSpace n c))
zipWith_st f (xs_empty, ys_empty, zs_full, xs_data, ys_data) = (xs_read,
     ys_read, zs_write, zs_data)
  where
    canfire = not zs_full && not xs_empty && not ys_empty
    (xs_read, ys_read, zs_write) = (canfire, canfire, canfire)
    zs_data = vzipWith f xs_data ys_data
```

Listing 4.8 – *zipWith* component

Similar to *zipWith* from the Haskell Prelude, the space time version accepts a binary function *f* and two lists *xs_data* and *ys_data*. The data of *xs_data*, *ys_data* and *zs_data* are distributed over space and time, denoted by the type *VecTime (VecSpace n a)*. Since *zipWith_st* is a dataflow component, execution is triggered by a firing rule. The firing rule states that execution can start when all inputs have data available and the FIFOs on the outputs are not full. The first two lines in the where-clause show the firing rule and the reading and writing values. For reading from and writing to the FIFOs some additional signals are introduced (*empty*, *full*, *read* and *write*). The actual calculation is shown on the last line, here the actual computation is performed using a instantiation of *zipWith* mapped completely in space (*vzipWith*). The mapping over time is implemented by sequentially processing data.

4.4 Example: Dot Product

To show the transformation based approach using types, a dotproduct operation is implemented. The dotproduct operation combines two vectors by pairwise multiplication of the elements and summing the results. Mathematically, this is denoted in equation 4.3.

$$z = \sum_{i=0}^{N-1} x_i \times y_i \tag{4.3}$$

Reformulating equation equation 4.3 in Haskell is done using the higher-order functions *zipWith* and *foldl*:

```
dotpr xs ys = z
    where
        ws  = zipWith (*) xs ys
        z   = foldl (+) 0 ws
```

Listing 4.9 – Dotproduct in Haskell

In order to implement the dotproduct from listing 4.9 in CλaSH, components from the aforementioned library have to be used and instantiated by hand. The transformation rule is already performed on the components in the library which can be instantiated directly. A *zipWith_st* and *foldl_st* are instantiated and parameterized with a multiplication and addition. Depending on the size of the sublists that are processed by these components, more parallelism can be achieved. These components are then composed with each other and two data sources using FIFO buffers. Listing 4.10 shows how the higher-order components are parameterized and instantiated for the dotproduct.

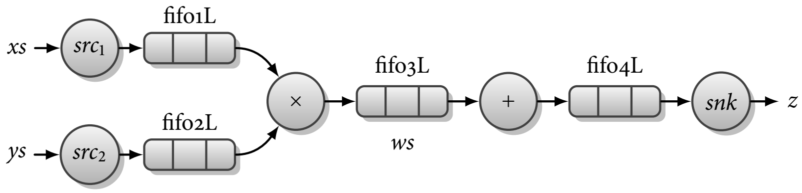


Figure 4.20 – Dataflow graph of dotproduct with FIFOs

Figure 4.20 shows the graph with FIFOs of the dotproduct. All components are interspersed with FIFOs for buffering of tokens. The sources src_1 and src_2 send *xs* and *ys* respectively in smaller parts to the multiplier where these parts are multiplied. The results of the multiplication are forwarded to the adder which determines the sum of all parts and forwards this to the sink *snk*. *snk* forwards the final result to the output. Note that only the direction of flow is shown using the arrows in figure 4.20. As shown in listing 4.10, for each FIFO the status signals *full* and *empty* and modification signals *read* and *write* are required as well.

```
multiplierL = (zipWith_st (*)) <^> L

adderL      = (foldl_st 4 (+) 0) <^> (0, 0)

arch _ = (valid, z)
    where
        (xs_write, xs_data)         = src1L xs_full
        (ys_write, ys_data)         = src2L ys_full
        (xs_full, f1_empty, f1_data) = fifo1L (xs_data, xs_write, f1_read)
        (ys_full, f2_empty, f2_data) = fifo2L (ys_data, ys_write, f2_read)
        (f1_read, f2_read, zs_write, zs_data)
                                    = multiplierL (f1_empty, f2_empty,
                                        zs_full, f1_data, f2_data)
        (zs_full, f3_empty, f3_data) = fifo3L (zs_data, zs_write, f3_read)
        (f3_read, qs_write, qs_data) = adderL (f3_empty, f4_full, f3_data)
        (f4_full, f4_empty, f4_data) = fifo4L (qs_data, qs_write, f4_read)
        (f4_read, valid, z)      = sinkL (f4_empty, f4_data)
```

Listing 4.10 – Dotproduct using CλaSH higher-order dataflow library

Listing 4.10 shows the implementation of the dotproduct using the CλaSH library. As shown in the first and third line, the higher-order components are parameterized with a binary function ((∗) and (+)) specifically for the dotproduct. In this example, every lists is split into 4 tokens which is given as a parameter to the adder *adderL*. *adderL* therefore produces a token with the sum for every four tokens it receives. The last part of the first and third line is the lifting of the components with an initial state using the <^> operator. The dummy state multiplier component is initialized with the bit low (L) since no counters are needed in the internal state. On line 3, the *foldl_st* component is parameterized with an adder and starting value 0. The initial state is a tuple containing two zeroes: one for the token counter and one for the intermediate result. *arch* describes the whole CλaSH implementation of the dotproduct. In the where-clause, two sources (*src1L* and *src2L*) are instantiated and connected to the multiplier (*multiplierL*) using FIFOs (*fifoL*). Data produced by the pairwise multiplications (*zs_data*) is forwarded to the summation step (*adderL*) using yet an other FIFO. Finally, all resulting data is consumed by a sink (*sinkL*) that presents the data on the output *z*.

Using the CλaSH compiler, the code from listing 4.10 has been translated to VHDL. This VHDL code is synthesized for an Altera Cyclone II 2C20F484C6 FPGA using the Quartus tool. Figure 4.21 shows the resulting register-transfer level (RTL) view.

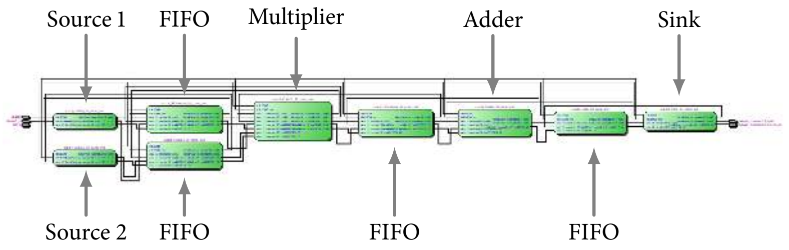


Figure 4.21 – RTL view of dotproduct

Comparing the RTL view from figure 4.21 with the code from listing 4.10 shows the direct correspondence between the CλaSH definition and the resulting hardware. When the size of the sublists is defined as four, four multipliers should be instantiated. Figure 4.22 shows the four multipliers and the firing rule of *multiplierL*.

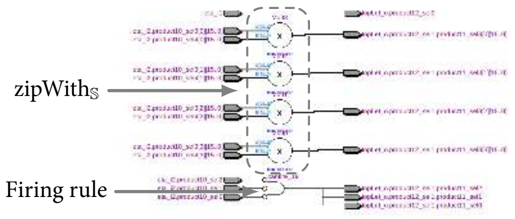


Figure 4.22 – RTL view of *multiplierL* component

4.5 Conclusions

In this chapter, the principle of trade-off rules based on the transformation of higher-order functions has been proposed. Using these transformation rules, a trade-off between parallelism and execution time can be made. To facilitate the use of these transformation rules and to model applications, an embedded language is developed for Haskell where space and time can be modeled using types. A CλaSH library has been developed containing higher-order components in which the aforementioned trade-off rules have been already applied. By parameterizing these components with application specific functions and composing these using FIFOs, actual hardware is implemented. Results of this method are presented using the implementation of a dotproduct.

Currently, the transformation rules map computations on both time and space. However, the spatial dimension can be split up into different parts. The transformation rules could therefore be extended for mapping to chips, boards and racks.

5 Case study: particle filter

Abstract – *In this chapter, the transformation rules to facilitate the trade-off between execution time and area usage on FPGAs are applied to the particle filter as a case study. All higher-order functions are transformed, resulting in parameterized nodes where the amount of parallelism and thereby performance and resource consumption can be controlled. For composition and scheduling of operations, dataflow principles are used. The resulting architecture is much more feasible in area compared to a fully parallel approach as the clock frequency in increased by a factor* 25 *while the area in LUTs is reduced by a factor* 5.7.

In chapter 3, a fully parallel design method for a particle filter has been explored. Although a fully parallel particle filter is feasible theoretically, some components require a lot of FPGA resources. Therefore, a more scalable approach is required and should include a trade-off between time and space. Transformation rules to derive this trade-off have been developed and elaborated in chapter 4. Using these new transformation rules the particle filter has been implemented once more.

The Haskell formulation of the particle filter presented in chapter 3 is again taken as reference. Most of the steps in the particle filter are described using higher-order functions. Using the transformation rules presented in chapter 4 the computations are distributed over space and time thereby limiting the amount of parallelism. This also limits the amount of resource usage resulting in a more feasible implementation. The resulting components are composed using dataflow principles to schedule operations and synchronize data.

The rest of this chapter is structured as follows. First, related design methodologies are discussed in Section 5.1 after which the application of the design methodology is presented in section 5.2. The results are presented in section 5.3 including a comparison with the parallel approach of chapter 3 and related work. Finally, in section 5.4 conclusions are drawn and directions for future work are discussed.

Large parts of this chapter have been published in [RW:5].

5.1 Related design methodologies

Design methodologies related to the particle filter design approach presented in this chapter can be divided in three categories: dataflow, transformations and the use of higher-order functions. In [59], dataflow principles have been used to design a particle filter using similar methods as presented in this chapter. Transformation based approaches are covered in [75, 96] while the use of higher-order functions for hardware design are extensively covered in [47, 104].

In [59], a particle filter is designed using dataflow mechanisms. All functions of the particle filter are wrapped into dataflow nodes and are connected using buffers. An example is presented containing two types of particle filters in a single architecture allowing it to be easily reconfigured. Compared to the design methodology presented in this thesis, there is a major difference: the parallelization of the dataflow nodes. The parallelization in [59] has to be performed by manually changing the dataflow graph while the approach advocated in this thesis generates nodes with parallelization parameters that can be chosen by the designer.

Hardware design using transformations has drawn a lot of attention in recent years. Two design methods related to the approach presented in this chapter are discussed. The first method is the rewriting mathematical formulas used in the Spiral project [96]. This method starts with a formula for discrete Fourier transform (DFT)-like computations which are iteratively rewritten until only a small set of Fourier transforms are left. These are then used to generate software or hardware. Although the method is very similar to rewriting higher-order functions, only DFT-like operations are supported. The second relevant method is a hardware design technique based on utility directed transformations (UDT). Utility directed transformations are transformations that are parameterized differently based on performance measures. In [75], a matrix-matrix multiplication is used as a test case where a better performing FPGA implementation is derived compared to related work. Although UDT has the same goal, increase performance by means of maximizing parallelism, the input model is very different compared to using higher-order functions. In [75], the input model is based on a Handel-C [25] like language while all transformations in this thesis are applied to higher-order functions only where data dependencies are known.

Using higher-order functions for hardware design is far from new since many ideas have been developed already in the eighties [104]. Higher-order functions turn out to be an adequate way in describing more or less regular structures in hardware. Especially DSP applications can be described concisely using higher-order functions. In [47], higher-order functions are used in CλaSH to design a reducer circuit for a floating point pipeline. These papers use higher-order functions to describe the combinatorial structure of hardware and essentially describes the behavior of the circuit for a single clock cycle. I contrast to previous approaches, the transformation-based approach of this thesis allows higher-order functions to be used to represent computations over multiple clock cycles. Time is therefore included in the approach.

5.2 Design methodology

As already elaborated in chapter 3, the whole Haskell description of the particle filter can be divided into two groups, higher-order functions and first-order functions. Higher-order functions are used to express structure and repetition with other functions as argument. First-order functions (functions that do not accept function-arguments) on the other hand are used as discrete components and correspond to combinatorial circuits like an adder for example. On hardware, first-order functions correspond to combinatorial circuits. By using transformations to reduce the amount of parallelism, computations of the particle filter are distributed over space and time. This methodology is summarized in figure 5.1.

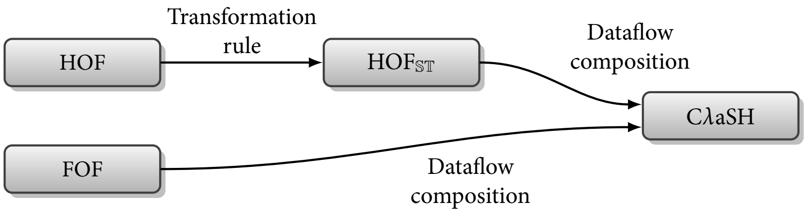


Figure 5.1 – Hardware design method

As shown in figure 5.1 a Haskell description of an application using a set of higher-order and first-order functions acts as starting point before transformation. Each higher-order function is distributed over time and space where possible and the resulting components are composed using dataflow nodes and FIFOs using the CλaSH library mentioned in section 4.3.2. First-order functions (functions accepting plain data as argument) are also wrapped in dataflow components such that these can be composed in the same way. The CλaSH code describing this composition is translated to VHDL and synthesized for FPGA.

5.2.1 Transformation of higher-order functions

All components of the particle filter in Haskell are defined by either higher-order functions or first-order functions. For components constructed using higher-order functions, a transformation rule can be applied. Figure 5.2 shows a graph containing all components of the particle filter with corresponding higher-order function.

As shown in figure 5.2 most components of the particle filter are constructed using a higher-order function. The components *Noise*, *Update* and *Norm* are constructed using *map* while *Predict* is constructed using *zipWith* and *Ws2Rs* using *scanl*. The reciprocal in *Recipr* is not an operation that processes data in some form of a list but accepts only a single value and produces one value for each input value. Therefore, no transformation rule can be applied and the function is kept in its original

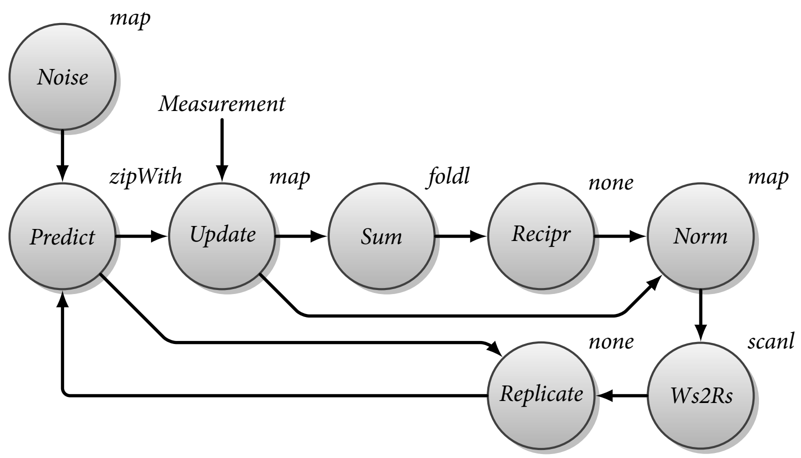


Figure 5.2 – Particle filter components an their higher-order functions

form. Although the component that replicates particles based on replication factors (*Replicate*) is constructed using the higher-order function *zipWith*, the resulting lists containing replicated particles have different lengths depending on the replication factor. This length changes at runtime which can not be be translated to hardware with CλaSH using a higher-order functions. Therefore no transformation rule is applied and a special implementation has to be selected when implementing this component using CλaSH. This special implementation is not based on the transformation of a higher-order function but sequentially replication particles and buffering until a set of particles can be sent out in parallel in the form of a token.

Taking into account the constraints that arise when designing for an FPGA, some components of the particle filter require a more efficient implementation. This is the case for the *Update* and *Replicate*. Although the computations of *Update* can be distributed over space and time, the size of a measurement should be taken into account as well during a transformation. A measurement is a complete image and therefore too large to be replicated. Therefore, the amount of parallelization is kept at 1 and the image is fed pixel by pixel to the update component where the particle are updated sequentially. As shown in section 3.3.2, the resampling step (*Replicate* and *Ws2Rs* in figure 5.2) can be implemented completely in parallel but this requires a lot of area and limits the clock frequency substantially. Therefore, the replication step is implemented sequentially (particle by particle) and particles are buffered such that tokens containing several particles can be generated.

Transformation of Predict component

To show the transformation based method, the process is shown for the prediction step. Since the the prediction step is based on the higher-order function *zipWith*, the corresponding transformation rule is used. Recall from section 4.1 that *zipWith* can be rewritten as follows:

$$zss = zipWith_{\mathbb{T}}\ (zipWith_{\mathbb{S}}\ f)\ xss\ yss$$

where *xss* an *yss* are lists with elements distributed over both space and time. The elements of these distributed lists arrive sequentially in the form of tokens containing several elements. In terms of hardware architecture, $zipWith_{\mathbb{S}}$ consists of instantiations of the system dynamics function f while $zipWith_{\mathbb{T}}$ is implemented using a firing rule for handling the sequential execution. Figure 5.3 shows the hardware architecture for the prediction component.

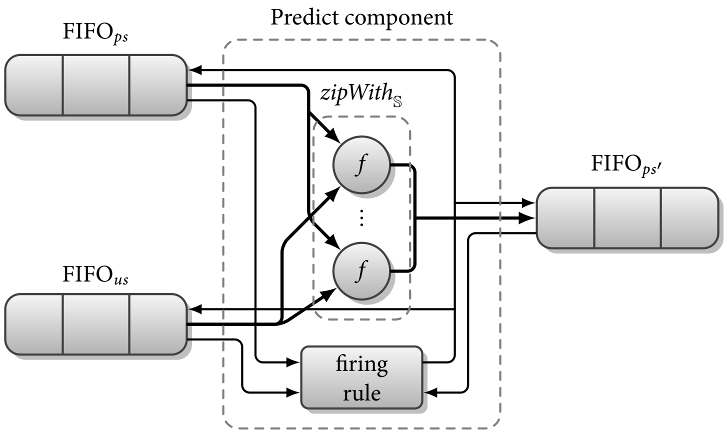


Figure 5.3 – Dataflow composition of Predict component

As shown in figure 5.3, the prediction component is connected to three FIFOs, two at the inputs and one at the output. Every instantiation of f accepts a particle from FIFO$_{ps}$ and a deviation from FIFO$_{us}$. All results are combined in a single token and sent to the FIFO on the output (FIFO$_{ps'}$) which is connected to the update component. The status signals connecting FIFOs with *firing rule* are the *empty* and *full* signals while the *read* and *write* signals are connected in reverse. The node can only fire if both input FIFOs have data available and the FIFO to which the result is sent is not full. Equation 5.1 expresses this mathematically.

$$fire = \overline{empty(FIFO_{ps})} \wedge \overline{empty(FIFO_{us})} \wedge \overline{full(FIFO_{ps'})} \qquad (5.1)$$

All other components shown in figure 5.2 are transformed using their corresponding trade-off rules as well. Though some components require a slightly altered approach during the implemented using CλaSH to be able to derive efficient hardware.

5.2.2 Implementation using CλaSH

After transforming each component using the corresponding higher-order function, the resulting components are composed using FIFOs forming a dataflow graph. Most components of the particle filter can be implemented by instantiating higher-order components from the CλaSH library as presented in section 4.3.2. For example, the prediction step is based on *zipWith* and is therefore implemented using the *zipWith* component from the CλaSH library by parameterizing it with the system dynamics function. However, two components require a special implementation for efficiency reasons: the update and replication components.

Although parallelism for the update component is easily achieved using the transformation rule for *zipWith*, it does not take into account the hardware costs of the measurement argument. This is because a measurement is a single frame and therefore contains a lot of pixels. Representing these using wires and registers requires a lot of hardware which is why we chose to store measurements in block RAMs (BRAMs). Every particle represents a position in the measurement. The address of the corresponding pixel can be calculated based on this position. Using the grayscale value of this pixel, a weight is calculated which is forwarded to the next step. Figure 5.4 shows the architecture of the update component.

Particles arrive at the input from a FIFO at the top. These particles arrive in the form of tokens each containing *P* particles (in this case 4). Each particle is selected sequentially using a multiplexer controlled by *contrl* in a round-robin ordering. When all *P* particles in the token are processed, the FIFO is instructed to drop this token and forward the next using the *read signal*.

Since each particle represents a position, it can be used directly as an address for selecting a pixel from the measurement. This address is forwarded to the BRAM containing the image. Based on this address, a pixel is returned from which a weight is calculated using the measurement function *g*. Since the amount of parallelization at the output is the same as the input, *P* weights have to be calculated before a token can be formed and forwarded to the output FIFO. These *P* weights are stored in a shift register forming a complete token after *P* cycles. This makes the output compatible with the dataflow-based communication. The firing rule is implemented in the *contrl* component taking into account the buffering of weights as well.

As shown in chapter 3, fully parallel replication requires a lot of area and introduces a very long combinatorial path. Although circuits where several particle could be replicated in parallel have been designed, the resulting hardware became to complex to gain performance from the parallelization. Therefore, a completely sequential implementation is chosen to minimize area usage. This sequential implementation uses only a single counter and can therefore run on a high clock frequency. Figure 5.5 shows the architecture for the replication of particles.

The replication component has two input FIFOs and one output FIFO. The first input contains tokens of particles while the second FIFO contains the replication factor. Similar to the update step, the particles are processed using a multiplexer

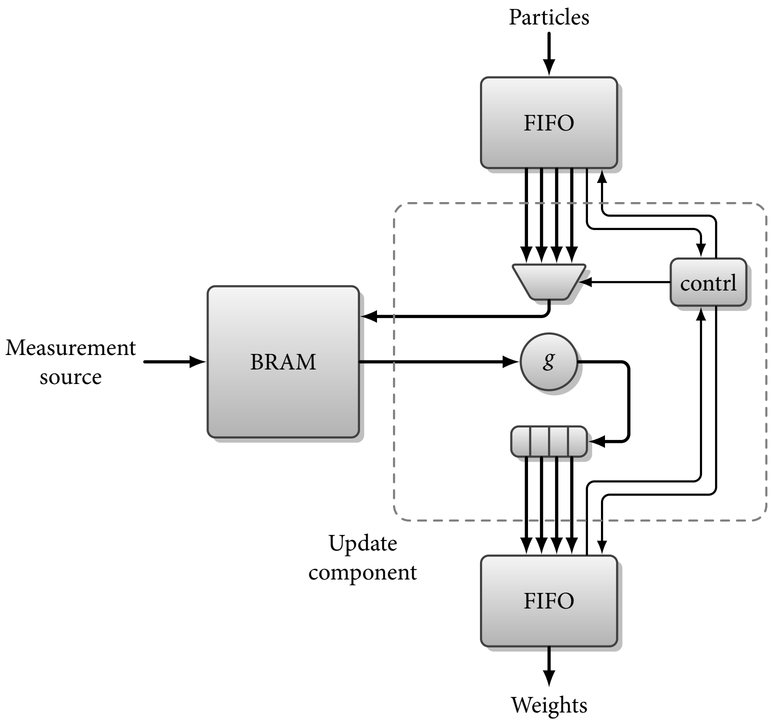


Figure 5.4 – Architecture of update step

that selects the particle in the token in a round-robin ordering. This ordering is generated by *contrl* which also takes into account the firing rule of the whole component. Replication of particles is achieved by sequentially copying particles into the shift register depending on the replication factor. When P particles are produced this way, a complete token is formed and can be written into the output FIFO. Once a particle is replicated as many times as indicated by the replication factor, the next particle and replication factor are selected from the input FIFOs.

After applying the transformation rules and implementing the update and replication components as presented before, the whole system can be composed using FIFOs. No central control is required since all computations are data-triggered. Since the resulting graph contains a cycle, some initial tokens have to be present on this cycle for the graph to be able to start. This initial set of tokens are present in the FIFO between the replication and prediction component and contain the initial particle cloud. Note that the total amount of particles remains constant over all iterations. As shown in figure 5.6, in the first few cycles the noise generating component *Noise* creates tokens containing deviations used in the prediction component. The prediction component then sends particles to both the update and replication component. Based on the positions represented by a particles, the update step cre-

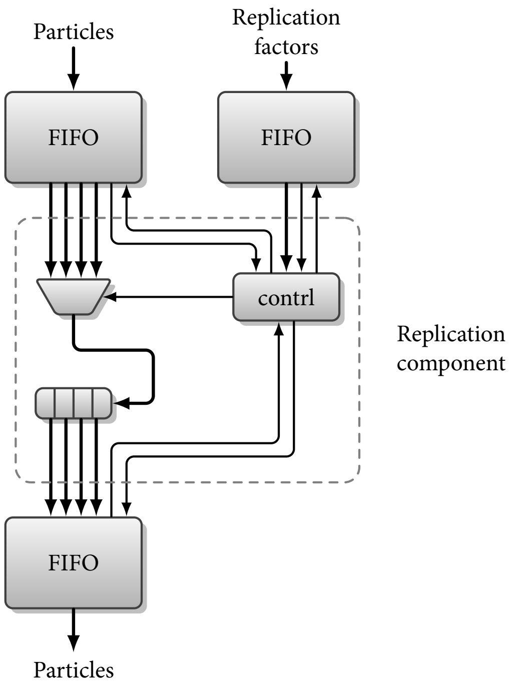


Figure 5.5 – Architecture of replication component

ates weights which are forwarded to *sum* and *Norm*. *Norm* uses the weights from the update component and the reciprocal of the sum of weights to normalize each weight. These normalized weights are forwarded to the component *Ws2Rs* where they are converted to replication factors. Finally, in *Replicate* particles are replicated according to their replication factor. The resulting particles are forwarded to the prediction components again, completing the cycle.

This dataflow graph of the whole particle filter as shown in figure 5.6 is implemented using the CλaSH library as presented before. Using the the CλaSH compiler, this design is translated to VHDL such that actual hardware metrics can be derived.

5.3 Results

To simplify verification, a small simulation framework has been built where a reference particle filter in plain Haskell is compared to a simulation of the dataflow particle filter implemented using CλaSH. Since CλaSH code is also valid Haskell code, no additional simulation tools are required and the simulation results can be directly compared in a single Haskell environment. During simulation, this framework produces a single set of images (256×256 pixels) for both particle filters to track. For each image, both particle filters generate a set of particles which are

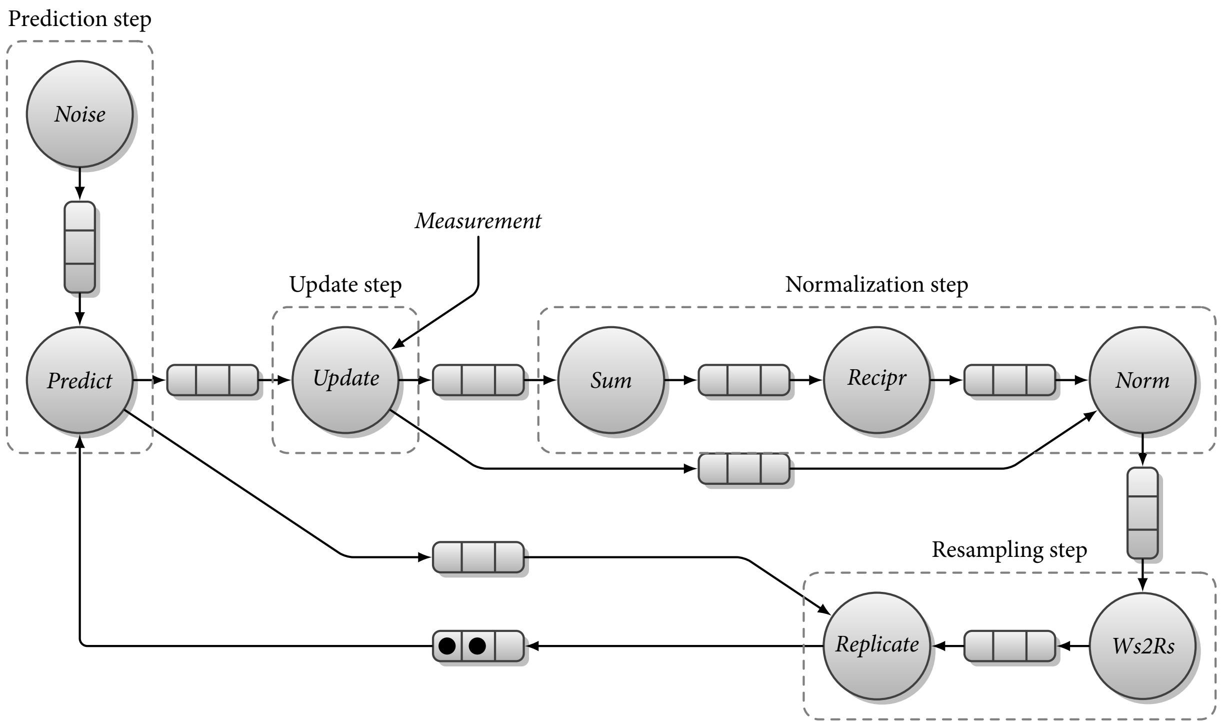


FIGURE 5.6 – Complete design of dataflow based particle filter

averaged to produce an estimate of the location of the square that is being tracked. The resulting tracks are displayed in figure 5.7 and figure 5.8.

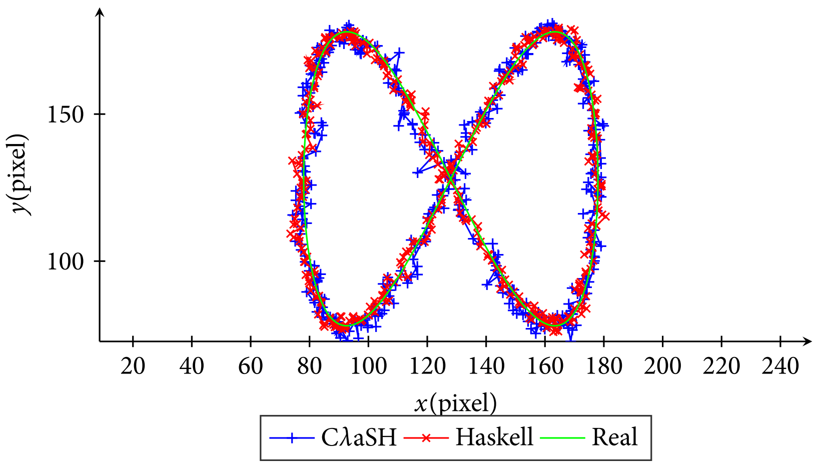


Figure 5.7 – Tracking of a Lissajous curve

As shown in figure 5.7, the path taken by the simulation of the dataflow particle filter (blue) is very similar to the reference implementation in plain Haskell (red). Both filters are able to track the square on the Lissajous path (shaped like the infinity symbol) within a few pixels. However, the dataflow particle filter deviates sometimes a few pixels more from the path than the Haskell reference. The main difference in the paths is caused by the stochastic nature of a particle filter. Other deviations are caused by the fixed-point implementation of the arithmetic operations since the dataflow particle filter uses 18 bits to represent particle weight while the plain Haskell implementation uses double precision floating point operations. Figure 5.8 shows the absolute error during tracking of both particle filters. Both filters deviate, on average, around 3 pixels from the actual path. Therefore the use of fixed-point operations on hardware has only a marginal effect on the tracking quality for this particular particle filter.

To determine the throughput of the dataflow-based particle filter, the token flow between nodes is monitored. Since the particle filter behaves periodic, a new state estimate is produced for every new measurement. Therefore, the average time to produce a set of particles with a new position estimate determines the throughput. By looking at the *write* signal of the FIFO between the replicator and the predictor, a graph can be produced showing when a token is written into the FIFO. Based on the average time between two sets of resampled particles, the performance of the complete filter can be calculated.

In figure 5.9, the communication activity between the resampling and prediction step with parallelization factor $P = 4$ (when active, a value is written into the FIFO) is shown. The resampled particles are sent in groups of 8 tokens to the predictor

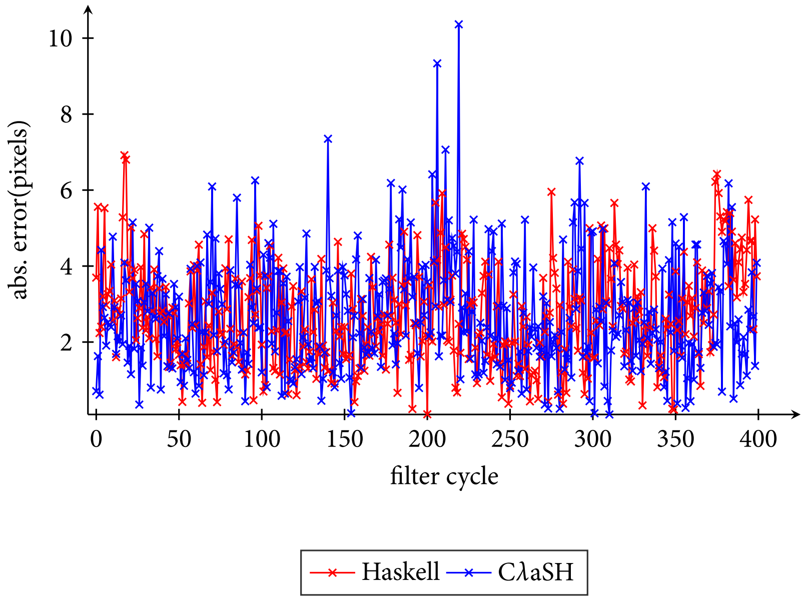


Figure 5.8 – Difference between actual path and predicted paths

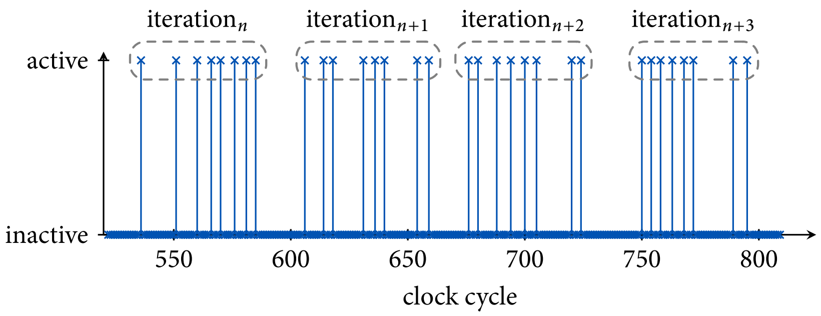


Figure 5.9 – Activity between *Replicate* and *Predict*

since the total numbers of particles is 32. Since $P = 4$, each token sent to the predictor contains 4 particles. By determining the arrival time of the first token for each group of 8, the average throughput of the whole particle filter can be derived. Averaging over the differences between arrival times of each first token results in an average cycle time as shown in table 5.1.

Increasing the parallelization P increases the throughput of the particle filter. The throughput is only marginally increased by parallelization due to the fact that some steps are implemented sequentially (the update and replication step). For $P = 8$ a higher throughput than $P = 4$ is expected. However, due to automated schedul-

Table 5.1 – Throughput of particle filter

	$P = 2$	$P = 4$	$P = 8$
Avg. nr. of clock cycles	77.1	69.5	70.9
Particles per clock cycle	0.42	0.46	0.45

ing of the dataflow components the throughput happens to be slightly lower. The maximum performance is therefore achieved for $P = 4$.

5.3.1 Hardware results

After successfully simulating both particle filters, the dataflow-based filter in CλaSH been translated to VHDL by the CλaSH compiler. The resulting VHDL code has been synthesized for a Virtex 6 XC6VLX240T FPGA available on the ML605 development board [1]. Using the transformation rules of chapter 4, three instantiations of the dataflow-based particle filter have been derived, having a sublist width of $P = 2, 4$ and 8 respectively. All instantiations are able run at a clock frequency of approximately $25MHz$ currently limited by the reciprocal operation. The reciprocal operation, found in the normalization step, determines the reciprocal of the sum of all weights and contains the longest combinatorial path of the whole circuit. This combinatorial path can be reduced by pipelining the reciprocal function since it is now implemented purely combinatorially. Table 5.2 shows the amount of LUTs used for the dataflow based particle filter. The number of hardware multipliers, used in the normalization step, is directly determined by the parallelization factor P. Therefore, the instantiations with $P = 2, 4, 8$ require $2, 4, 8$ 18-bits hardware multipliers respectively.

	$P = 2$		$P = 4$		$P = 8$	
Component	**LUTs**	**FFs**	**LUTs**	**FFs**	**LUTs**	**FFs**
Noisegen	70	64	138	128	274	256
Predict	37	-	69	-	133	-
Update	44	28	44	50	61	94
Sum	81	22	116	21	187	20
Recipr	923	-	923	-	923	-
Norm	20	4	29	3	48	2
Ws2Rs	204	30	333	29	592	28
Replicate	70	42	126	76	214	142
FIFOs	5210	4021	4650	3707	4435	3538
Total:	6659	4211	6428	4014	6867	4080

Table 5.2 – Resource usage of dataflow based particle filter

Figure 5.10 shows a graphical representation of the numbers in table 5.2. The figure shows that, for components based on a higher-order function, the number of LUTs required scales more or less linear with P. Similarly, Figure 5.11 shows the amount of FlipFlops graphically. Again, most of he components scale linear in flip flop consumption with P. Most LUTs and flip flops (FFs) are currently used by the FIFOs. Therefore, to have a better view on the effect of changing the parallelization P, the results of the FIFOs are left out of the graphs. Note that the costs of the reciprocal component in figure 5.10 do not change since it is not affected by a transformation rule.

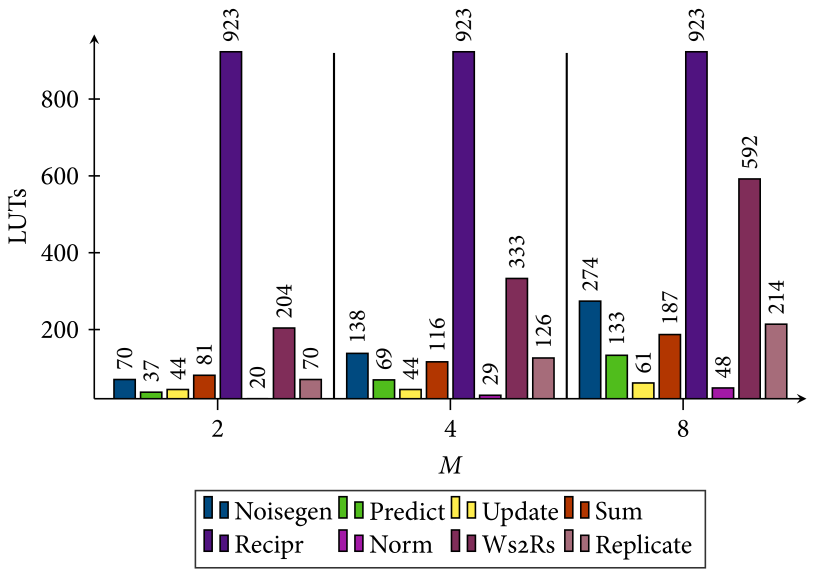


Figure 5.10 – LUTs used by components of particle filter

Since all instantiations are able to run at $25MHz$ and the throughput for $P = 4$ is the highest (table 5.1), the actual throughput of the dataflow-based particle filter can be calculated. The throughput now becomes $0.46 * 25 * 10^6 = 11.5 * 10^6$ particles per second.

5.3.2 Comparison to related work

By comparing the synthesis of the dataflow based particle filter with other work, some indication on relative performance can be given. Table 5.3 shows the synthesis results from a similar, but fully parallel particle filter from chapter 3. Comparing table 5.2 and table 5.3 shows that the dataflow approach is $39038/6867 \approx 5.7$ times as small as a fully parallel filter in terms of LUTs for $P = 8$. The clock frequency is also improved by a large amount, $1MHz$ for the parallel implementation versus $25MHz$ for the dataflow-based implementation. Additionally, most of the LUTs of the dataflow particle filter are used by FIFOs since these are implemented using logic and not (yet) by BRAMs. Due to quadratic scaling, the most area consuming part of the parallel particle filter was the resampling step. With the sequentialization

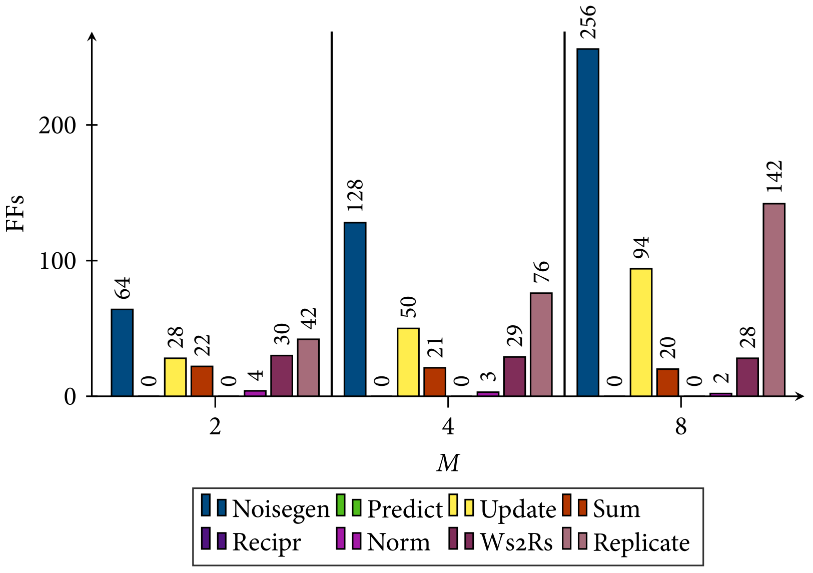


Figure 5.11 – FlipFlops used by components of particle filter

using the dataflow approach, the resampling step is much more balanced in size compared to the other steps.

Component	**Fully parallel PF**	**Dataflow-based PF** $P = 8$
Prediction	704	407
Update	954	61
Normalization	1402	1158
Resampling	35978	806
FIFOs	-	4435
Total	39038	6867

Table 5.3 – LUTs usage of particle filters

Currently, the bandwidth to the BRAM limits the update rate of the particle filter and not the algorithm itself. The implementation is therefore now bandwidth limited similar to results found in [29] where a particle filter based video tracker is implemented.

5.4 Conclusion

The design method based on the transformation of higher-order functions as proposed in chapter 4 has been applied to a particle filter application. The transformation rules produce dataflow nodes with a parallelization parameter. By choosing a proper value for this parameter, a trade-off between execution time and FPGA area is made. When applied to the particle filter example, the design method produces scalable hardware in terms of throughput and FPGA area consumption. Since transformations work on the level of higher-order functions, low-level details like fixed-point implementation of arithmetic operations remains unchanged. All mathematical dependencies are preserved, only the way of execution is changed. Higher-order functions are therefore an adequate abstraction level to express mathematical dependencies as it facilitates the trade-off between time and space.

All transformations and implementations of dataflow nodes have now been performed by hand. A possible direction for future work is to automate this process in the form of an embedded language for example. This embedded language would allow the designer to easily express designs using higher-order functions and the tooling processing this language then applies the transformation rules to generate optimized hardware that fits within the limitations of the FPGA.

Currently, both the update step and replicator process at most one particle per clock cycle and have therefore great influence on the total performance. By using dual port BRAMs and replicating it, the bandwidth to read pixels from the image-BRAM can easily be increased. The throughput of the updater can therefore by increased by a factor of four at the cost of an additional BRAM. An other possible optimization to increase the throughput of the whole filter is to use several clock domains. Nodes containing a long combinatorial path can be placed in domains with a lower clock frequency while other nodes can be placed in a faster clock domain. Currently, the *Recipr* node limits the throughput of particle filters since it contains the longest combinatorial path. By pipelining the *Recipr* node, the clock frequency can be increased such that other parts of the particle filter can achieve a higher throughput.

6

Stencil computations

Abstract – *In this chapter, the set of transformation rules is extended to incorporate stencil computations. Two transformation rules are proposed to facilitate the trade-off between execution time and chip area specifically for stencil computations. The design method for designing stencil computation hardware using these transformation rules consists of two steps. First the transformation rule is applied to the higher-order function thereby making a trade-off between time and space. The second step is to include buffering to prevent excessive communication. Using these two steps efficient hardware is derived with performance and resource consumption figures that are similar to related work.*

Up to this chapter, only one-dimensionally structured higher-order functions are considered. In this chapter, the set of transformation rules will be extended to include stencil computations. Stencil computations are often two-dimensional and have overlap in data. These properties require special attention when using transformation rules to derive FPGA hardware.

Two higher-order functions specifically designed for stencil computations are proposed with accompanying transformation rules. These transformation rules are applied to the higher-order functions to distribute computations over space and time. To increase the communication efficiency, the resulting circuit is altered such that data is buffered, preventing excessive communication. The result is a higher-order component that can be parameterized with an application-specific kernel function. Among others, applications of the transformation rules for heatflow and cellular automata are developed and evaluated.

The remainder of this chapter is structured as follows. First, an introduction to stencil computations is given in section 6.1 after which related work is discussed in section 6.2. The transformation rule for stencil computations is introduced in

Large parts of this chapter have been published in [RW:6].

section 6.3 followed by several case studies in section 6.4. Actual hardware is designed for a heatflow case study and results are discussed in section 6.5. Finally, in section 6.6 conclusions are drawn and possible directions for future work are discussed.

6.1 Stencil Computations

Stencil computations are a family of algorithms in which arrays are processed using a sliding window approach, i.e., a slice of array-elements is used to determine one new value. Such computations can be found in many scientific and engineering fields like digital signal processing (DSP), digital image processing and computational fluid dynamics (CFD).

During a stencil computation, a function (also known as kernel) is applied to all elements in the array to find values of a new array. For each application of the kernel, surrounding elements of the center point in the array are used. The pattern of surrounding elements used by the kernel is called a stencil and is application dependent. However, at runtime, this pattern does not change. This can be exploited when designing hardware for it.

The kernel is usually described using a single function. This function depends on a sequence of elements from the array surrounding the center element. Assuming that the stencil is symmetrical, stencil computations on one-dimensional arrays can be defined as shown in equation 6.1 in which X_i are the elements of the input array.

$$Y_i = f([X_{i-M} \dots X_i \dots X_{i+M}]) \tag{6.1}$$

As shown in equation 6.1, the i^{th} element in Y is computed based on the sequence $X_{i-M} \dots X_{i+M}$. The width of such a stencil is therefore $2M + 1$ elements. Figure 6.1 shows equation 6.1 graphically for three subsequent elements of the resulting array Y Note that only mathematical dependencies are shown and not the order of execution.

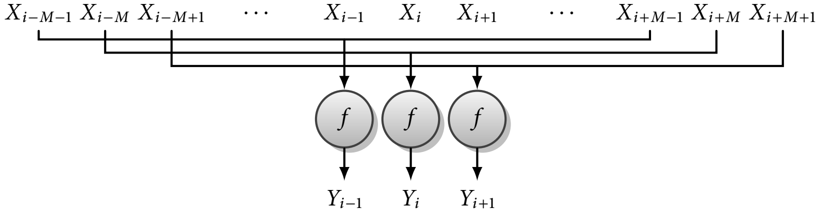


Figure 6.1 – Structure of one-dimensional stencil computation

A small example of a one-dimensional stencil computation is a sliding average. Equation 6.2 shows the formula for a sliding average stencil with a width of 3 elements.

$$Y_i = \frac{1}{3} \sum_{n=-1}^{1} X_{i+n} \tag{6.2}$$

Similar to a one-dimensional stencil computation, higher-dimensional stencil computations also depend on surrounding values. The general formulation of a two-dimensional computation is stated in equation 6.3 where X_i refers to the center point of window.

$$Y_{i,j} = f\left(\begin{bmatrix} X_{i-M,j+N} & \cdots & X_{i+M,j+N} \\ \vdots & X_{i,j} & \vdots \\ X_{i-M,j-N} & \cdots & X_{i+M,j-N} \end{bmatrix}\right) \tag{6.3}$$

As shown in equation 6.3, a two-dimensional stencil computation uses elements from both horizontal and vertical dimensions. All elements are selected from a square with $(2M + 1) \times (2N + 1)$ elements. The structure of dependencies between elements and computations of equation 6.3 is shown in figure 6.2.

An example of a two-dimensional stencil computation is the Gaussian-blur filter [107]. A Gaussian-blur filter determines a new pixel based on a weighted set of surrounding pixels. The larger this set of surrounding pixels, the more blurry the resulting image becomes. Equation 6.4 shows the mathematical definition of a Gaussian filer with $M = N = 2$, $\mu = 0$ and $\sigma = 1$.

$$Y_{i,j} = \sum_{n=-2}^{2} \sum_{m=-2}^{2} \frac{1}{2\pi} e^{-\frac{x^2+y^2}{2}} \times X_{i+n,j+m} \tag{6.4}$$

There exist a plethora of applications that fit the stencil computation pattern. For one-dimensional stencil computations examples are elementary cellular automata, finite impulse response (FIR) filters and one-dimensional heatflow algorithms. Examples of two-dimensional stencil computations are the Conways's game of life, two-dimensional convolution, many image processing algorithms and two-dimensional heatflow algorithms.

6.2 Related work

The use of higher-order functions to describe stencil computations in Haskell is well developed in the PASTHA framework [72]. This framework contains a parallelization tool for stencil computations on multicore machines. Similar to the VHDL

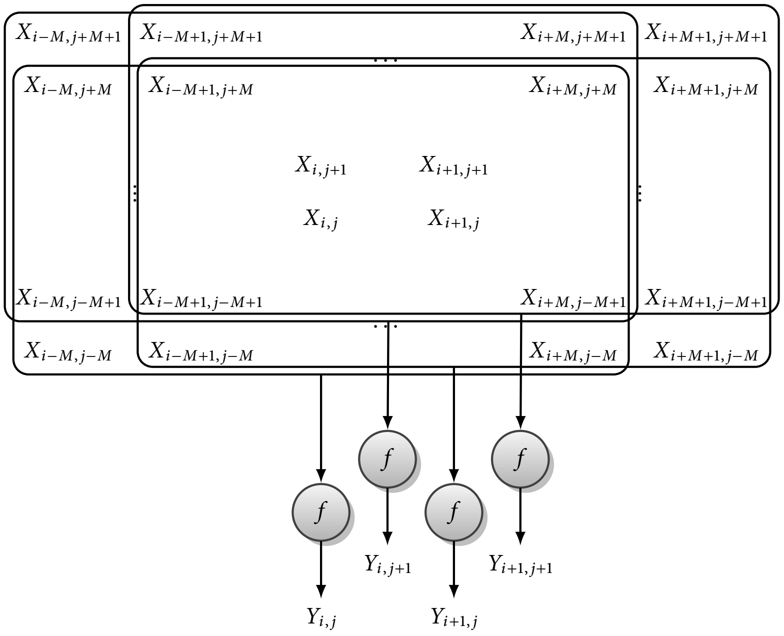


Figure 6.2 – Structure of two-dimensional stencil computation

hardware template of [102], the hardware resulting from the transformations presented in this chapter is parameterizable in the amount of parallelism taking into account memory-structure and communication like in [39] and [121]. Memory and communication aspects impose constraints on the FPGA implementation as shown in [54], [40] and [63]. Especially in designs with multiple FPGAs, communication patterns become a very important factor to achieve high performance [100].

Most design methods for stencil computation use, at some stage, an imperative description (in C for example) of the operations. Listing 6.1 shows the pseudocode of such an imperative description. For every point at x, y in a frame v at time t, the new value $v(t+1, x, y)$ is calculated using the stencil function F. Although this description is very similar to the formal definition in equation 6.3, the code is inherently sequential. Therefore, a lot of analysis is required to determine the dependencies between loop iterations before parallelization can be performed. Also details of intermediate stages are often hidden or hard to modify. Therefore, the description of the generated hardware looks very different compared to the initial definition. By using the methodology proposed in this chapter, the intermediate results following the transformation retain their structure.

Another approach to implementing stencil computations on FPGAs is the use of a domain specific language (DSL) [76]. A compiler takes care of parallelization and scheduling. For parallelization of stencil computations the approach taken in

```
for(x=0; x<WIDTH; x++) {
  for(y=0; y<HEIGHT; y++) {
    v(t+1,x,y) = F(w in stencil(v,x,y));
  }
}
```

Listing 6.1 – Imperative code for stencil computations

this chapter is similar to the parallelization of FIR filters in [88, 90] but requires no array index computations. Since the methodology makes use of higher-order functions, it is convenient to use a hardware description language with higher-order function support built in. Therefore, the CλaSH language [16] is used. Other work on translating Haskell to hardware is Lava [19] and the more recent Kansas Lava [50]. As discussion in section 2.1, the main difference between Lava and CλaSH is that Lava is a language embedded in Haskell while the CλaSH compiler takes plain Haskell code as input and thereby supporting more language features.

6.3 Transformations for Stencil Computations

Similar to the transformation rules in chapter 4, the derivation of hardware from a higher-order stencil function is performed in two steps. First, a transformation rule distributes the stencil computations over two domains; a part over space by parallelization and a part over time by sequential execution of partial stencil computations. Adding synchronization after the first transformation results in a description that can directly be translated to hardware. However, this results in a lot of communication overhead which is why a second step is needed to derive efficient hardware. This second step consists of transforming the description to increase data reuse.

The transformations proposed in this section distribute the stencil computations over space and time in a fully parameterizable way. A generic higher-order function for stencil computations is parameterized using an application-specific kernel function. An application is therefore implemented by specifying a kernel function and passing this to the higher-order function for stencil computations.

6.3.1 Space/Time Transformation

The starting point is a definition of the stencil computations in Haskell. Stencil computations are performed by the *stencil* function which takes three arguments; an application-specific kernel function f, a window width w and a list of inputs xs. The first three lines of listing 6.2 show the definition and implementation of a one-dimensional moving average filter being applied to xs. As shown on the first line, *stencil* is supplied the application-specific function *avg* to define a sliding average filter. The remainder of listing 6.2 gives the Haskell implementation of *stencil*. *stencil* is expressed recursively, where the stencil kernel function f is applied

```
res   =  stencil avg 3 xs
  where
    avg xs = 1 / 3 * sum xs

stencil f w xs
  | length xs >= w  = (f (take w xs)) : (stencil f w (tail xs))
  | otherwise      = []
```

Listing 6.2 – Definition of average filter and stencil higher-order function

to the beginning of the list. Then, *stencil* is again applied to the list excluding the first element. This process continues until the number of elements left becomes smaller than the window width. In the remainder of this chapter, *stencil* should be considered a native function.

Performing all operations in the stencil computation in a single clock cycle would require a lot of area on an FPGA when synthesized directly. Therefore, a trade-off between area and execution time is required. This trade-off is found by applying a transformation rule to *stencil*. As shown in the visualization of a one-dimensional stencil computations (figure 6.1), the function f is applied to every sublist of xs with N elements including overlap. Note, that the resulting list is smaller such that corner conditions do not occur (i.e., the result has $2M$ fewer elements: the size of the overlap is M elements).

In order to save FPGA resources, the kernel functions f are distributed over space $\mathbb{S}$ and time $\mathbb{T}$ by applying the transformation of *stencil* as shown in listing 6.3. After this transformation, $stencil_{\mathbb{S}\mathbb{T}}$accepts four parameters: a parallelization factor p, kernel function f, stencil width w and the list of inputs xs. The input data is split into smaller lists using the *splitt* function taking into account the overlap between them. This is visualized in figure 6.3. These sublists are then processed sequentially (mapped over time), by the $map_{\mathbb{T}}$function. This function applies the *stencil* function to a complete sublist in one clock cycle using a single (native) combinatorial component $stencil_{\mathbb{S}}$. The amount of parallelism is therefore determined by the size of each sublist which is a parameter for *split* (p). This transformation-based method therefore does not need loop unrolling or dependency analysis of *for* loops.

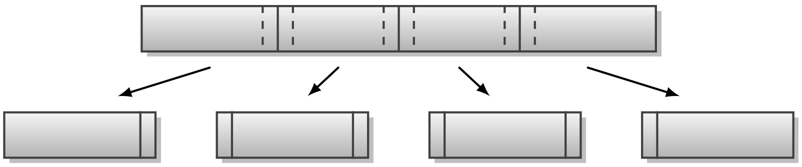


Figure 6.3 – Splitting of one-dimensional data

A similar trade-off between time and space can be derived for two-dimensional stencil computations. Again, the trade-off rule ensures that all input data is divided into smaller blocks which are processed sequentially. As shown in figure 6.4, two-

```
stencilst p f w xs = concat_t ress
  where
    xss   =  splitt w p xs
    ress  =  mapt (stencils f w) xss
```

Listing 6.3 – Trade-off rule for *stencil*

```
stencil2dst (hp, vp) f (w, h) img = concat2dt img'
  where
    imgss  =  split2dt (w, h) (hp, vp) img
    ress   =  map2dt (stencil2ds (w, h) f) imgss
```

Listing 6.4 – Trade-off rule for *stencil2d*

dimensional data is split into blocks with additional borders to ensure that the overlap between data is maintained. Listing 6.4 shows the trade-off rule for two-dimensional higher-order stencil function *stencil2d*. Note the introduction of an additional parallelization parameter vp (vertical parallelism) due to the additional spatial dimension. Similar to *stencil*$_{\mathbb{ST}}$, *stencil2d*$_{\mathbb{ST}}$ accepts four arguments as well: a tuple with the horizontal and vertical parallelization factors (hp, vp), the stencil function f, a tuple with stencil width and height (w, h) and the two-dimensional input data img.

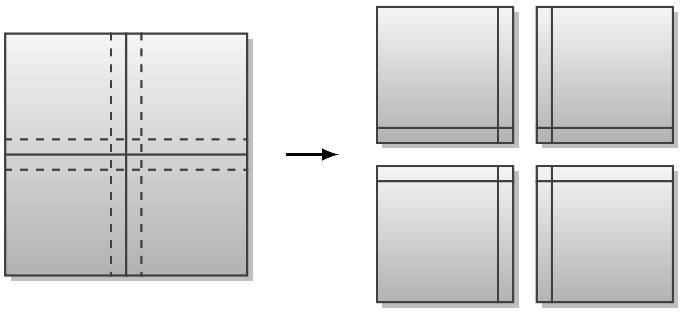


Figure 6.4 – Splitting of two-dimensional data

stencil$_{\mathbb{ST}}$can be translated directly to hardware by adding dataflow logic to *stencil*$_{\mathbb{S}}$ for easy synchronization. However, due to overlap between sublists, additional bandwidth is required to keep the *stencil*$_{\mathbb{S}}$ component utilized. Therefore, the next step is to rewrite *stencil*$_{\mathbb{ST}}$ to an architecture that buffers data for reuse, thereby minimizing communication.

6.3.2 Deriving the Architecture

In order to reduce communication overhead, data must be kept as close to the computation as possible. This is usually implemented by buffering data from previous cycles [39, 76, 121]. The architecture derived in the second step of the design

```
1 stencilarch :: ([a] -> b) -> [a] -> [a] -> ([a], [b])
2 stencilarch f xs inp = (xs', outp)
3   where
4     xs'   =  inp +>>> xs
5     outp  =  stencils f
```

Listing 6.5 – One-dimensional stencil architecture.

methodology should therefore take this buffering into account as well. Deriving the architecture is performed by wrapping *stencil*$_\mathbb{S}$ of listing 6.3 into an architecture to handle communication and synchronization.

Figure 6.5 shows a stencil computation architecture with parallelization $p = 2$. Elements of p samples are accepted each clock cycle and processed by *stencil*$_\mathbb{S}$. These elements are shifted into a shift register such that they can be used in subsequent clock cycles. The actual stencil computation is implemented combinatorially by *stencil*$_\mathbb{S}$.

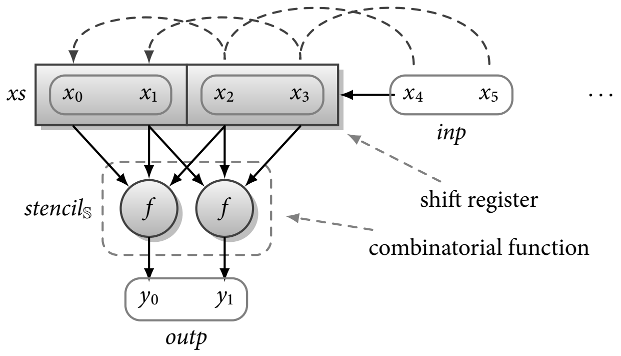


Figure 6.5 – Architecture for one-dimensional stencil computation ($P = 2$)

Listing 6.5 shows the code for the one-dimensional stencil computation architecture of figure 6.5 in the form of a function named *stencilarch*. *stencilarch* accepts three arguments: the kernel function f, the current state of the shift register xs and the new input sample(s) inp (shown on line 2 while line 1 gives the type). The result tuple consists of two parts: the new state of the shift register xs' and the computed output $outp$. xs' is found by shifting the input list *inp* (with p samples) completely into xs such that the last p samples are dropped off. Finally, the output $outp$ is found by applying the kernel function f to all buffered input samples xs. Depending on *stencil*$_\mathbb{S}$, P incarnations of the function f are performed in parallel. Also, the type of $outp$ is a list containing p elements. *stencilarch* is a higher-order function since it can be parameterized with a specific kernel function f. However, it now represents an actual architecture in the form of a Mealy machine instead of an abstract mathematical function.

```
stencilarch2d :: (Img a -> b) -> (Img a, Img a) -> Img a -> ((Img a, Img a),
     Img b)
stencilarch2d f (ws, ss) inp = ((ws', ss'), outp)
  where
    outp  =  stencil2ds f ws
    ws'   =  mergeh (droph 2 ws) (mergev (takeh 2 ss) inp)
    ss'   =  mergeh (droph 2 ss) (takeh 2 (dropv 1 ws))
```

Listing 6.6 – Two-dimensional stencil architecture.

A similar architecture is derived for two-dimensional stencil computations. The architecture consists of three parts: a window buffer for holding data to which the kernel function f is applied, line buffers for storing complete lines of the input and a part where the actual output is calculated. Two-dimensional stencil computations can be parallelized by processing several elements at once. Given a parallelization of $p = 2$, an architecture as shown in figure 6.6 is derived.

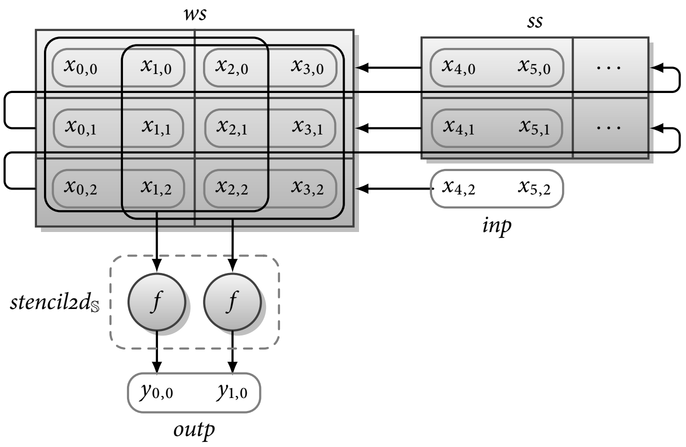


Figure 6.6 – Architecture for two-dimensional stencil computation ($P = 2$)

As shown in figure 6.6 the *stencil2d*$_\mathbb{S}$ is the only part in the architecture where computations are performed. All other parts are used for buffering. Since $p = 2$, two elements of the input data are sent to the architecture every cycle. This also means that all buffers forward the data in packets of two. Also *stencil2d*$_\mathbb{S}$ processes two stencils at once resulting in two output samples ($y_{0,0}$ and $y_{1,0}$) being produced at the same time. The code to implement and simulate this architecture is shown in listing 6.6.

Listing 6.6 shows the Haskell implementation of the two-dimensional stencil computation architecture. As shown by the type, the first argument f is a function that takes an image of type a and produces an element of type b. The second argument represents the state consisting of two buffers while the third argument is part of the input image with type *Img a*. For completion of the Mealy machine structure, the result consists of the new state of the buffers and the actual result. The first line of the where-clause implements the parallel processing of the window buffer *ws*. The new state of the window and line buffers (ws' and ss') are found by shifting in and out pairs of elements using the functions for merging and slicing images (*mergeh*, *mergev*, *droph*, *dropv* and *takev*).

In order to generate actual hardware, the Haskell descriptions of *stencilarch* and *stencilarch2d* have to be altered slightly such that it is accepted by the CλaSH compiler. Lists are not supported by the CλaSH compiler which is why they are replaced by vectors. A small library with higher-order functions specifically for vectors has been developed such that the code can be compiled by the CλaSH compiler more or less unchanged. More details on altering code using lists to vectors can be found in [RW:1].

6.4 Case Studies

To show the generality of using the *stencil* and *stencil2d*, several case studies have been implemented. Convolution, cellular automata and heatflow have been implemented in the form of an application specific function that can be given as parameter to either *stencil* or *stencil2d*.

6.4.1 Convolution

Convolution is a commonly used operation in digital signal processing and image processing and fits the stencil computation pattern quite well [94]. As shown in equation 6.5, during convolution, two signals, f and g, are shifted along each other, multiplied and added for every position i. In practice, signals are often finite which is why the summation is limited between $-N$ and N.

$$y_i = (f * g)_i \stackrel{\text{def}}{=} \sum_{n=-\infty}^{\infty} f_n \, g_{i-n} \approx \sum_{n=-N}^{N} f_n \, g_{i-n} \tag{6.5}$$

For a two-dimensional convolution, the mathematical formulation is very similar. As shown in equation 6.6, an additional summation is introduced to iterate over elements of the second dimension. This second summation is limited in range as well ($-M \ldots M$).

$$y_{i,j} = (f * g)_{i,j} \stackrel{\text{def}}{=} \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} f_{n,m} \, g_{i-n,j-m} \approx \sum_{n=-N}^{N} \sum_{m=-M}^{M} f_{n,m} \, g_{i-n,j-m} \tag{6.6}$$

Using the definitions of convolution from equation 6.5 and equation 6.6, the application specific parameter function for *stencil* and *stencil2d* can be formulated in Haskell. For a one-dimensional convolution, this function accepts an array of $2M + 1$ elements while the two-dimensional function accepts a two-dimensional array of $2M+1$ by $2N+1$ elements. Listing 6.7 and listing 6.8 show the Haskell implementations of a convolution with a Gaussian window in one and two dimensions respectively.

```haskell
gauss xs = dotpr xs cs
  where
    cs = map (\x -> 1 / sqrt (2 * pi) * exp (-0.5 * x^2)) [-2,-1,0,1,2]
    dotpr ps qs = foldl (+) 0 (zipWith (*) ps qs)

res = stencil 5 gauss inplist
```

Listing 6.7 – One-dimensional Gaussian function Haskell

As shown in listing 6.7, the one-dimensional Gaussian function *gauss* is implemented using a dotproduct of input elements *xs* and coefficients *cs*. The coefficients *cs* are calculated by applying the Gaussian window function to a range of indices $[-2,-1,0,1,2]$. On the last line is shown how the function *gauss* can be used as parameter to *stencil* filtering a list *inplist*.

```haskell
gauss2d xss = dotpr2d xss css
  where
    css = chunksOf 5 [1/(2*pi)*exp (-0.5*(x^2+y^2)) | x <- [-2.0..2.0], y <-
        [-2.0..2.0]]
    dotpr2d pss qss = foldl (foldl (+)) 0 (zipWith (zipWith (*)) pss qss)

res = stencil2d (5,5) gauss2d inpimage
```

Listing 6.8 – One-dimensional Gaussian function Haskell

Similar to the one-dimensional Gaussian function shown in listing 6.7, the two-dimensional Gaussian function is implemented using a two-dimensional dotproduct. This dotproduct combines the input matrix *xss* with the Gaussian window *css*. On the last line of listing 6.8, the use of the function *gauss2d* is shown by supplying it as parameter to the function *stencil2d*.

6.4.2 Cellular Automata

Cellular automata work on a grid of elements by executing simple rules to alter the state. All the new states of the cell are found by applying a function to a cell and a set of neighbours. It therefore matches the stencil computation pattern as shown by two implementations: the one-dimensional elementary cellular automaton rule 110 [4] and game of life [45].

Rule 110 is a elementary cellular automaton where the new state of a cell is determined based on the current state of the cell and the state of the adjacent cells. A cell can only be in one of two states: 1 or 0. There exist $2^{2^3} = 256$ elementary cellular automata since there exist $2^3 = 8$ possible input patterns. For each of those patterns the state can be either 1 or 0. Rule 110 states that the new state of the cell is 1 if the current pattern is in $\{110, 101, 011, 010, 001\}$ and 0 otherwise. The Haskell code is shown in listing 6.9 while the result of a simulation is shown in figure 6.7.

```haskell
rule110 xs = if xs ‘elem‘ [[1,1,0], [1,0,1], [0,1,1], [0,1,0], [0,0,1]]
  then 1
  else 0
```

Listing 6.9 – Rule 110

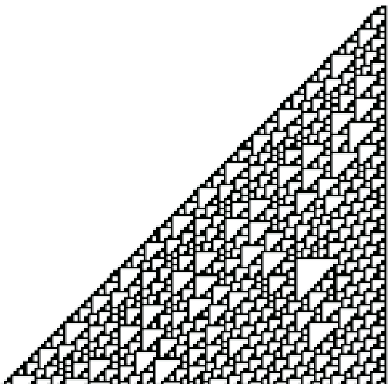


Figure 6.7 – Simulation of rule 110

Game of life

Game of life is a stencil computation applied to a grid with cells as well. Each cell can be either dead or alive. The new state of the cell is determined based on all direct neighbours using the following rules:

» a cell dies when it has four or more neighbours due to overcrowding
» a cell dies when it has one or zero neighbours due to loneliness
» a cell is born or stays alive when it has three or two neighbours

In listing 6.10, the Haskell implementation of game of life is shown.

As shown in listing 6.10, the number of neighbouring living cells *nc* is determined first followed by the current value of the middle cell *ccv*. Finally the rules are used to determine the new state. Figure 6.8 shows the simulation results of the four stages of a glider (an object that follows a straight line).

6.4.3 Heatflow

The flow of heat through materials is described using the general heat equation of equation 6.7. In this equation, $T(t, \vec{x})$ describes the temperature of a point $\vec{x}$

```haskell
golkernel inp = res
  where
    nc = sum (inp !! 0) + (inp !! 1 !! 0) + (inp !! 1 !! 2]) + sum (inp !! 2)
    ccv = inp !! 1 !! 1
    res = if ccv == 1
        then
          if nc == 2 || nc == 3 then 1.0 else 0.0
        else
          if nc == 3 then 1.0 else 0.0
```

Listing 6.10 – Game of life

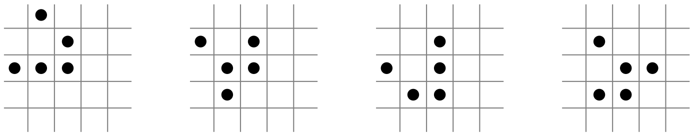


Figure 6.8 – Four stages of a glider in game of life simulation

in the material with thermal diffusivity α at time t. After discretization over both time and space, the formulation of equation 6.8 is derived for a one dimensional heatflow problem and equation 6.9 for a two dimensional problem [42].

$$\frac{\partial T(t,\vec{x})}{\partial t} = \alpha \nabla^2 T(t,\vec{x}) \tag{6.7}$$

$$T_{k+1,i} = T_{k,i} + c \times \left(T_{k,i-1} - 2T_{k,i} + T_{k,i+1}\right) \tag{6.8}$$

Equation 6.8 shows the discretization of equation 6.7 for a one-dimensional heatflow problem. It uses two neighbouring points ($T_{k,i-1}$ and $T_{k,i+1}$) to determine the temperature for the next time instance $T_{k+1,i}$. The heat diffusivity is captured in the constant c, the time instance is represented by k and the position in the one-dimensional space is represented by i.

$$T_{k+1,i,j} = T_{k,i,j} + c \times \left(T_{k,i-1,j} - 4T_{k,i,j} + T_{k,i+1,j} + T_{k,i,j+1} + T_{k,i,j-1}\right) \tag{6.9}$$

Similar to equation 6.8, the position in the x-dimension is represented by i and the time by k. Since equation 6.9 describes a two-dimensional heatflow problem, an additional spatial dimension is required which is represented by j. The new temperature depends not only on the current temperature, left and right neighbouring temperatures, but also on the upper and lower temperatures $T_{k,i,j+1}$ and $T_{k,i,j-1}$.

Formulating the one-dimensional heatflow problem of equation 6.8 in Haskell is relatively straightforward. As shown in listing 6.11, a stencil of three temperatures ($x0$, $x1$ and $x2$) is used to determine the new temperature.

```
heatflow [x0, x1, x2] = x1 + c * (x0 - 2 * x1 + x2)
```

Listing 6.11 – One-dimensional heat flow kernel

As shown in listing 6.12, the Haskell implementation for the two-dimensional case is very similar. The first and single argument is the stencil holding the current and surrounding temperatures (*xss*). Current and neighbouring temperatures are selected using the index operator !!. The result is a linear combination of these temperatures.

```
hfk2d xss = x11 + c * (-4 * x11 + x01 + x12 + x21 + x10)
  where
    (x01, x21)      = (xss !! 0 !! 1, xss !! 2 !! 1)
    (x10, x11, x12) = (xss !! 1 !! 0, xss !! 1 !! 1, xss !! 1 !! 2)
```

Listing 6.12 – Two-dimensional heat flow kernel

Both the one-dimensional and two-dimensional Haskell implementation of the heatflow equation have been simulated to verify their functionality. Figure 6.9 shows the simulation of a one-dimensional problem like a metal rod while figure 6.10 shows the heat propagation through a metal plate.

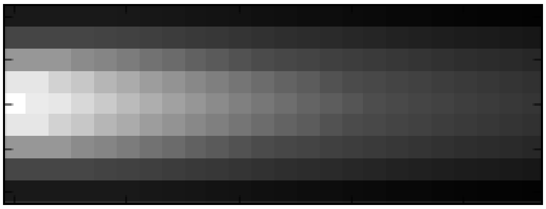


Figure 6.9 – One-dimensional heatflow simulation

Figure 6.9 shows the propagation of heat through a one-dimensional medium where the vertical direction represent the position and the horizontal time. Over time, the hotspot in the middle evens out over the whole height due to the cold (black) temperatures at the top and bottom.

Similarly, the function for the two-dimensional heatflow problem is simulated using the initial temperatures of $t = 0$. The initial temperatures has a hot spot in the top left part. Over time, the heat in this spot evens out ($t = 32$) and eventually disappears ($t = 256$). Different temperatures are supplied to the borders. The upper and right border are heated while the others kept at a low temperature. Slowly, the heat from the upper and right border propagates deeper into the material ($t = 256$).

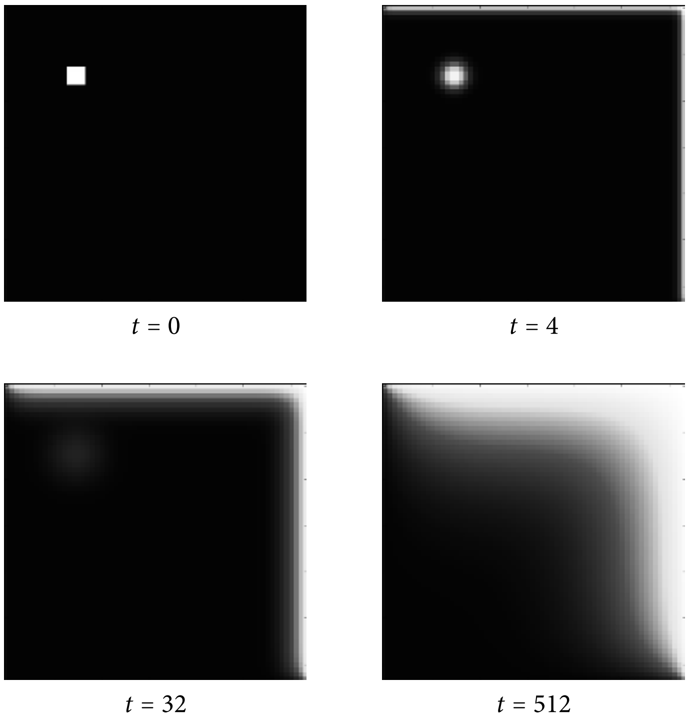


Figure 6.10 – Two-dimensional heatflow simulation

6.5 Hardware Results

For both the one-dimensional and two-dimensional heatflow kernel, hardware has been generated using the CλaSH compiler These have been instantiated with different parallelization factors (P) to show the scaling on FPGA hardware. All designs have been synthesized for a Xilinx XC5VLX110T FPGA and were all capable of running at 200 MHz. The performance is directly determined by the parallelization factor, i.e., the number of stencil computations performed in a single clock cycle is equal to the parallelization factor P. Table 6.1 shows the amount of resources required for computation (LUTs and DSP48E multipliers) and storage (REGs). The number of clock-cycles to compute a stencil computation for a list with 256 elements and an image of 256×256 is shown in the rightmost column.

Both the one-dimensional and two-dimensional architectures scale linearly with parallelization factor P in terms of LUTs. The amount of registers for the one-dimensional architecture scales linearly as well but for the two-dimensional architecture the amount of registers is practically constant. This is because most registers are used for the line buffers. Compared to related work of [102], a similar 2D architecture is derived with very similar resource consumption even though a different

Table 6.1 – Area of stencil computation architectures.

	P	LUTs	REGs	DSP48E	clock-cycles
1D	1	72	72	1	256
	2	143	96	2	128
	4	285	144	4	64
	8	569	240	8	32
	16	1137	432	16	16
	32	2273	816	32	8
2D	1	116	12336	1	65536
	2	231	12360	2	32768
	4	461	12408	4	16384
	8	961	12504	8	8192
	16	1921	12696	16	4096
	32	3841	13080	32	2048

kernel is used. Both the amount of LUTs required and clock frequency is comparable to the design in [102].

6.6 Conclusion

The set of transformation rules presented in chapter 4 have been extended and generalized to include stencil computations in which two-dimensional data structures are processed with overlapping data dependencies. Using the higher-order functions *stencil* and *stencil2d*, stencil computations can be expressed using a single higher-order function without the use of for loops with the possible accompanying off-by-one errors. Using the transformation rules the amount of parallelism can be controlled using a single parameter.

Several applications have been implemented using this approach including heat flow and cellular automata. These applications have been simulated to verify their functionality. For the one-dimensional and two-dimensional heatflow applications, real hardware has been designed using CλaSH. This hardware has shown to scale linearly with the parallelization parameter. Also resource consumption is comparable with related work.

Currently, only one and two-dimensional stencil applications have been considered. Higher-dimensional stencil computations are an interesting direction for future work since this puts more stress on resource usage and communication. The transformation rules should therefore be altered taking these constraints into account.

7 Conclusions and Recommendations

The main trend that can be observed in the last thirty years of developments of FPGA technology, is the enormous increase in the amount of programmable resources. To be able to utilize all these resources, a lot of parallelism is required. In this thesis, this problem is addressed by utilizing abstractions from the functional language Haskell in the context of hardware design. Higher-order functions in particular, are a natural abstraction to express structure and parallelism in hardware since only mathematical dependencies are described. By starting with a structural description of hardware and applying meaning-preserving transformations to it, additional dependencies that prevent possible parallelization are avoided.

In chapter 2, background information on several hardware design trends and tools is given. A popular approach to increase the abstraction level and design productivity is by using high-level synthesis. In the traditional high-level synthesis approach, often an imperative language is parallelized and translated to hardware. A more formal and transformational approach can be found in the SPIRAL project where DFT-like algorithms are implemented and optimized using iterative rewriting of DFT formulas. In this thesis, similar to the SPIRAL project, rewrite rules are used to derive mathematical equivalent description of circuits. However, a higher design productivity is pursued by using abstractions from functional languages instead of the imperative approach taken by most HLS projects. For the implementation of actual hardware, CλaSH is used as it incorporates a lot of abstractions that are specifically useful for hardware design. Especially useful is the support for higher-order functions to express structure in hardware.

In chapter 3, the connection between discrete mathematics and the use of higher-order functions is explored. A particle filter is implemented by slight alterations to the Haskell model that is derived from the mathematical definition. Particle filters are challenging applications to implement in hardware due to the available parallelism, data-dependent processing and a feedback loop. The amount of parallelism is maximized by mapping every operation to hardware. Although hardware could be derived using this approach, the scalability of this approach is limited since the unbounded amount of parallelism in these applications require a lot of resources. A trade-off between resource consumption and execution time is therefore required.

In chapter 4, a method to facilitate in the trade-off between resource consumption and execution time is explored. In order to distribute computations over space and time, several transformation rules for higher-order functions have been derived. Using these transformations, data and computations are split into smaller slices. Computations that are performed on a slice are executed in parallel while all slices are processed sequentially. The size of the slice, a parameter introduced by the transformation, determines the amount of parallelism and resource consumption. Proofs are derived showing that the transformations are meaning-preserving. Additionally, an embedded language is proposed to capture the mapping over time and space in types to prevent erroneous composition.

In chapter 5, the transformation rules proposed in chapter 4 are applied to the particle filter. All higher-order functions are transformed, resulting in parameterized components where the amount of parallelism and thereby the performance and the resource consumption can be controlled. For composition and scheduling of operations, dataflow principles are used. Using these techniques, the clock frequency of the particle filter has been increased to 25 MHz while the area has been reduced by a factor 5.7.

The set of transformation rules presented in chapter 4 has been extended and generalized in chapter 6. Both one and two-dimensional stencil computations can now be expressed using a higher-order functions as well. Using transformation rules, specifically derived for stencil computations, the amount of parallelism can be controlled using a single parameter. Several applications have been implemented using this approach including heat flow and cellular automata.

As formulated in section 1.2, the research question addressed in this thesis is:

» *How can a designer make a transparent trade-off between resource usage (chip area) and execution time?*

In this thesis, this research question is answered by investigating the use of higher-order functions. Higher-order functions can be used effectively to express structure and parallelism in DSP applications. Using transformation rules, computations expressed using higher-order functions are distributed over space and time, effectively making a trade-off between resource usage (chip area) and execution time. The amount of parallelism is controlled by a parameter that is introduced during the application of the transformation rule giving the designer full control of resource consumption and execution time.

7.1 Contributions

The three main contributions of this thesis are:

» **A design methodology for hardware based on exploiting regularity of higher-order functions.** In this thesis, a design methodology is presented showing how hardware can be designed by using a commonly used abstraction in functional languages: higher-order functions. First, a mathematical formulation of a DSP algorithm is expressed using higher-order functions to capture the structure and dependencies among operations. The second step is the transformation of this expression using transformation rules such that efficient hardware can be derived using the CλaSH compiler.

» **Transformation rules to distribute computations, expressed using higher-order functions, over space and time.** For several commonly used higher-order functions like *zipWith* and *foldl*, transformation rules have been derived. Additionally, these transformation rules have been proven to be meaning-preserving. The transformation rule distributes the computations, expressed using a higher-order function, over space and time. The amount of parallelism and resource usage can be fully transparently controlled by the designer using a parameter that is introduced by the transformation.

» **Several case studies showing the applicability of the design methodology to a large range of DSP applications.** Among others, the design methodology has been applied to a FIR filter, a particle filter and several stencil computation applications. The connection between discrete mathematics, higher-order function and hardware is first explored in a dotproduct example after which the methodology is applied to a particle filter. Stencil computations are explored to extend the set of transformation rules such that the design methodology can also be applied to two-dimensionally structured applications.

7.2 Recommendations

Although transformation rules for several higher-order functions have been derived and shown to be sufficient for several applications, a lot more applications can be supported by developing more higher-order functions with corresponding transformation rules. In this thesis, the focus has been on higher-order functions with regular structure where the dimensions are known at compile time. A possible direction for future work would be the derivation of rules for higher-order functions where structure is data-dependent. These higher-order functions can be supported by implementing recursion on hardware.

Currently, all higher-order functions to which the transformation rules of chapter 4 have been applied can be used by instantiating CλaSH components. In the future, an integrated approach that combines the transformation rules with the typed embedded language can be developed bringing all steps together in a single environment. In this environment, a design can be formulated and simulated after which the transformation rules are applied resulting in a parameterized model where the amount of parallelism can be selected by the designer. Once the trade-off between time and space is performed by selecting the proper parameters, hardware is generated. A well known method to achieve this level of integration is by developing an embedded language. In this embedded language, the applications can be defined, transformed and parameterized and simulated to determine performance metrics.

Instead of letting the designer choose the optimal parameters of parallelism, this process could also be automated. To achieve this, an optimization problem has to be defined taking into account the scaling of resource consumption given a certain higher-order function. This does require an FPGA tool to be in the optimization loop in order to give feedback regarding costs and constraints of the chosen FPGA. The optimization process then selects a candidate value for a parameter, determines the performance metrics by synthesizing a part of the application and uses this as feedback to change the parameter to minimize costs.

A

Shallow embedded language for space and time types

In section 4.3, a shallow embedded language for space and time types has been introduced. Using this embedded language, an applications to which the transformation rules of chapter 4 have been applied can be modeled. In this appendix, the implementation of this embedded language is presented.

Listing A.1 shows the implementation of the embedded language including some examples. Every type is based on the types for space and time (*LstSpace* and *LstTime*) as shown on line 8 and 9. When the elements of a list are distributed over space and time, their respective types are also composed (line 14). Furthermore, the higher-order functions *map*, *zipWith* and *foldl* are implemented in two versions: distributed over time and over space respectively. Finally, line 92 and further show how the space and time specific functions are used in a dot product example.

```haskell
1  module SpaceTime where
2
3  import Data.List
4  import Data.List.Split
5
6  -- The types for expressing list that are distrubuted
7  -- over space or time:
8  type LstSpace a = [a]
9  type LstTime a = [a]
10
11 -- Split list into sublists where the elements in the
12 --  sublists are distributed over space while the
13 --  sublists are distributed over time
14 split_st :: Int -> [a] -> LstTime (LstSpace a)
15 split_st p xs = chunksOf p xs
16
17 -- Split list into sublists where the elements in the
18 --  sublists are distributed over time while the
```

```haskell
--  sublists are distributed over space
split_ts :: Int -> [a] -> LstSpace (LstTime a)
split_ts p xs = chunksOf p xs

-- Merge list of list into single list over space
concat_s :: [[a]] -> LstSpace a
concat_s xss = concat xss

-- Merge list of list into single list over time
concat_t :: [[a]] -> LstTime a
concat_t xss = concat xss

-- Apply function f to all elements in xs over time
map_t :: (a -> b) -> LstTime a -> LstTime b
map_t f xs = map f xs

-- Apply function f to all elements in xs in parallel
map_s :: (a -> b) -> LstSpace a -> LstSpace b
map_s f xs = map f xs

-- Tradeoff transformation rule applied to map
map_st :: Int -> (a -> b) -> [a] -> [b]
map_st p f xs = concat_s yss
    where
        xss = split_st p xs
        yss = map_t (map_s f) xss

-- Result of map_st and map are equal
resmap = map_st 2 (+1) [1..6] == map (+1) [1..6]

-- Fold function f over all elements in xs over time
foldl_t :: (a -> b -> a) -> a -> LstTime b -> a
foldl_t f x ys = foldl f x ys

-- Fold function f over all elements in xs in parallel
foldl_s :: (a -> b -> a) -> a -> LstSpace b -> a
foldl_s f x ys = foldl f x ys

-- Tradeoff transformation rule applied to foldl
foldl_st :: Int -> (a -> b -> a) -> a -> [b] -> a
foldl_st p f x ys = res
    where
        yss = split_st p ys
        res = foldl_t (foldl_s f) x yss

```

```
-- Result of foldl_st and foldl are equal
resfoldl = foldl_st 2 (+) 0 [1..6] == foldl (+) 0 [1..6]

-- ZipWith with operations executed over time
zipWith_t :: (a -> b -> c) -> LstTime a -> LstTime b ->
    LstTime c
zipWith_t f xs ys = zipWith f xs ys

-- ZipWith with operations executed in parallel
zipWith_s :: (a -> b -> c) -> LstSpace a -> LstSpace b ->
    LstSpace c
zipWith_s f xs ys = zipWith f xs ys

-- Tradeoff transformation rule applied to zipWith
zipWith_st :: Int -> (a -> b -> c) -> [a] -> [b] -> [c]
zipWith_st p f xs ys = concat_s zss
    where
        xss = split_st p xs
        yss = split_st p ys
        zss = zipWith_t (zipWith_s f) xss yss

-- Result of zipWith_st and zipWith are equal
reszipWith = zipWith_st 2 (+) [1..6] [2..7] == zipWith (+)
    [1..6] [2..7]

-- Plain definition of dotproduct
dotpr xs ys = foldl (+) 0 ws
    where
        ws = zipWith (*) xs ys

-- Computations of dotproduct distributed over space and time
dotpr_st p xs ys = z
    where
        xss = split_st p xs
        yss = split_st p ys
        wss = zipWith_t (zipWith_s (*)) xss yss
        z = foldl_t (foldl_s (+)) 0 wss

-- Result of dotpr_st and dotpr are equal
resdotpr = dotpr [1..6] [3..8] == dotpr_st 2 [1..6] [3..8]
```

Listing A.1 – Space and time types with examples

Acronyms

A	ASIC	application specific integrated circuit
B	BMF	Bird-Meertens formalism
	BRAM	block RAM
C	CFD	computational fluid dynamics
	CLB	configurable logic block
	CPU	central processing unit
D	DCT	discrete cosine transform
	DFT	discrete Fourier transform
	DSL	domain specific language
	DSP	digital signal processing
E	EDSL	embedded DSL
F	FF	flip flop
	FFT	fast Fourier transform
	FIFO	first-in-first-out buffer
	FIR	finite impulse response
	FOF	first-order function
	ForSyDe	formal system design
	FPGA	field-programmable gate array
	FPLA	field-programmable logic array
G	GHC	Glasgow Haskell compiler
	GHCI	interactive GHC
H	HDL	hardware description language
	HLS	high-level synthesis
	HOF	higher-order function
I	IP	intellectual property
	ISPL	instruction set processor language
L	LFSR	linear feedback shift register
	LUT	look-up table
M	MAC	multiply accumulate
P	PDF	probability density function
	PF	particle filter

R	RAM	random-access memory
	ROCCC	Riverside optimizing compiler for configurable circuits
	RSR	residual systematic resampling
	RTL	register-transfer level
S	SIMD	single instruction multiple data
	SIRF	sequential importance resampling filter
	SoC	system on chip
	SPL	signal processing language
	SRAM	static RAM
U	UDT	utility directed transformations
V	VHDL	very high speed integrated circuit HDL
	VLIW	very large instruction word
W	WHT	Walsh-Hadamard transform

Bibliography

[1] Virtex-6 FPGA ML605 Evaluation Kit, 2014. URL `http://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html`. (Cited on page 78).

[2] Particle filter, python cookbook, 2012. URL `http://www.scipy.org/Cookbook/ParticleFilter`. (Cited on page 33).

[3] ROCCC manual, june 1, 2012. URL `http://roccc.cs.ucr.edu/UserManual.pdf`. (Cited on pages 16 and 17).

[4] *A New Kind of Science*. Wolfram Media Inc., Champaign, Ilinois, US, United States, 2002. ISBN 1-57955-008-8. (Cited on page 93).

[5] IEEE Standard SystemC ® Language Reference Manual, 2005. (Cited on page 14).

[6] J. Alarcon, R. Salvador, F. Moreno, P. Cobos, and I. Lopez. A new Real-Time Hardware Architecture for Road Line Tracking Using a Particle Filter. In *IEEE Industrial Electronics, IECON 2006 - 32nd Annual Conference on*, pages 736–741, Nov 2006. doi: 10.1109/IECON.2006.347566. (Cited on pages 34 and 35).

[7] Altera. Altera multicore ARM in SoC, . URL `https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html`. (Cited on page 4).

[8] Altera. Altera SoCs, . URL `https://www.altera.com/products/soc/overview.html`. (Cited on page 4).

[9] N. M. M. Alves and S. de Mello Schneider. Implementation of an embedded hardware description language using haskell. 9(8):795–812, aug 2003. URL `http://www.jucs.org/jucs_9_8/implementation_of_an_embedded`. (Cited on page 25).

[10] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *Signal Processing, IEEE Transactions on*, 50(2):174 –188, feb 2002. ISSN 1053-587X. doi: 10.1109/78.978374. (Cited on page 28).

[11] P. J. Ashenden. *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon) (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2008. ISBN 0120887851, 9780120887859. (Cited on page 9).

[12] Astron. Unioard. URL `http://www.astron.nl/r-d-laboratory/uniboard/uniboard-i-and-ii`. (Cited on page 1).

[13] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić. Generic Hardware Architectures for Sampling and Resampling in Particle Filters. *EURASIP J. Appl. Signal Process.*, pages 2888–2902, 2005. doi: http://dx.doi.org/10.1155/ASP.2005.2888. (Cited on pages 34 and 35).

[14] C. P. Baaij. *Digital Circuits in CλaSH – Functional Specification and Type-Directed Synthesis*. PhD thesis, Universiteit Twente, PO Box 217, 7500AE Enschede, The Netherlands, jan 2015. (Cited on pages 10, 11, 14, 24, and 25).

[15] C. P. R. Baaij, M. Kooijman, J. Kuper, M. E. T. Gerards, and E. Molenkamp. Tool demonstration: CLasH - from Haskell to hardware. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell, Edinburgh, Scotland*, pages 3–3, New York, 2009. ACM. (Cited on page 10).

[16] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France*, pages 714–721, USA, September 2010. IEEE Computer Society. (Cited on pages 10 and 87).

[17] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4): 487–504, 1984. (Cited on page 45).

[18] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computer Science*, pages 151–218. Springer-Verlag, 1988. NATO ASI Series F Volume 55. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University. (Cited on pages 21 and 45).

[19] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: 10.1145/289423.289440. URL `http://doi.acm.org/10.1145/289423.289440`. (Cited on pages 23, 24, and 87).

[20] W. A. Boeijink, P. K. F. Hölzenspies, and J. Kuper. Introducing the PilGRIM: A processor for executing lazy functional languages. In J. Hagen and M. T. Morazan, editors, *22nd International Symposium on Implementation and Application of Functional Languages, IFL 2010, Alphen a/d Rijn, The Netherlands*, volume 6647 of *Lecture Notes in Computer Science*, pages 54–71, Berlin, 2011. Springer Verlag. (Cited on page 25).

[21] M. Bolic. *Architectures for efficient implementation of particle filters*. PhD thesis, Stony Brook University, 2004. (Cited on pages 34 and 35).

[22] M. Bolić, P. M. Djurić, and S. Hong. Resampling algorithms for particle filters: a computational complexity perspective. *EURASIP J. Appl. Signal Process.*, 2004:2267–2277, January 2004. ISSN 1110-8657. doi: http://dx.doi.org/10.1155/S1110865704405149. URL `http://dx.doi.org/10.1155/S1110865704405149`. (Cited on page 31).

[23] M. Bolic, P. Djuric, and S. Hong. Resampling algorithms and architectures for distributed particle filters. *Signal Processing, IEEE Transactions on*, 53(7):2442–2450, July 2005. ISSN 1053-587X. doi: 10.1109/TSP.2005.849185. (Cited on pages 34 and 35).

[24] J. Bos. Synthesizable specification of a VLIW processor in the functional hardware description language CλaSH, September 2014. URL `http://essay.utwente.nl/66086/`. (Cited on page 14).

[25] M. Bowen. Handel-C Language Reference Manual. *Embedded Solutions Ltd*, 2, 1998. (Cited on page 68).

[26] T. Bronkhorst. Hardware design of a cooperative adaptive cruise control system using a functional programming language, August 2014. URL `http://essay.utwente.nl/65686/`. (Cited on page 10).

[27] O. Cappe, S. Godsill, and E. Moulines. An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo. *Proceedings of the IEEE*, 95(5):899 –924, may 2007. ISSN 0018-9219. doi: 10.1109/JPROC.2007.893250. (Cited on pages 29 and 31).

[28] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: 10.1145/1248648.1248652. URL `http://doi.acm.org/10.1145/1248648.1248652`. (Cited on pages 5 and 22).

[29] J. U. Cho, S. H. Jin, X. D. Pham, J. W. Jeon, J. E. Byun, and H. Kang. A Real-Time Object Tracking System Using a Particle Filter. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2822–2827, Oct 2006. doi: 10.1109/IROS.2006.282066. (Cited on pages 34, 35, and 80).

[30] P. P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006. ISBN 0471720925. (Cited on pages 9 and 36).

[31] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011. ISSN 0278-0070. doi: 10.1109/TCAD.2011.2110592. (Cited on page 15).

[32] A. Corporation. Altera Megafunctions. URL `http://www.altera.com/products/ip/altera/mega.html`. (Cited on page 9).

[33] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion. From Lists to Streams to Nothing at All. In *ICFP'07*, 2007. (Cited on page 22).

[34] D. Crisan and A. Doucet. A survey of convergence results on particle filtering methods for practitioners. *Signal Processing, IEEE Transactions on*, 50(3):736–746, Mar 2002. ISSN 1053-587X. doi: 10.1109/78.984773. (Cited on page 29).

[35] P. D'Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. F. Moura, M. Püschel, and J. Johnson. Generating FPGA accelerated DFT libraries. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 173–184, 2007. (Cited on page 20).

[36] N. Dave, M. Pellauer, S. Gerding, and Arvind. 802.11a transmitter: A case study in microarchitectural exploration. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, MEMOCODE '06, pages 59–68, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0421-5. doi: 10.1109/MEMCOD.2006.1695901. URL `http://dx.doi.org/10.1109/MEMCOD.2006.1695901`. (Cited on page 25).

[37] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `http://doi.acm.org/10.1145/1327452.1327492`. (Cited on page 53).

[38] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 5B.4.1–5B.4.7, April 2011. doi: 10.1109/IRPS.2011.5784522. (Cited on page 4).

[39] Y. Dong, Y. Dou, and J. Zhou. Optimized Generation of Memory Structure in Compiling Window Operations Onto Reconfigurable Hardware. In *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 110–121, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71430-9. URL `http://dl.acm.org/citation.cfm?id=1764631.1764645`. (Cited on pages 86 and 89).

[40] B. Draper, J. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe. Accelerated image processing on FPGAs. *Image Processing, IEEE Transactions on*, 12(12):1543–1551, Dec 2003. ISSN 1057-7149. doi: 10.1109/TIP.2003.819226. (Cited on page 86).

[41] J. P. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 079238542X. (Cited on page 15).

[42] J. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley, 2014. ISBN 9781118730966. URL `https://books.google.nl/books?id=m0LAAgAAQBAJ`. (Cited on page 95).

[43] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009. (Cited on page 20).

[44] P. Gammie. Synchronous Digital Circuits As Functional Programs. *ACM Comput. Surv.*, 46(2):21:1–21:27, Nov. 2013. ISSN 0360-0300. doi: 10.1145/2543581.2543588. URL `http://doi.acm.org/10.1145/2543581.2543588`. (Cited on pages 23 and 25).

[45] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970. (Cited on page 93).

[46] M. E. T. Gerards, C. P. R. Baaij, J. Kuper, and M. Kooijman. Hiding state in CλaSH hardware descriptions. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010), Alphen aan den Rijn, the Netherlands*, pages 107–119, Utrecht, August 2010. Utrecht University. (Cited on page 10).

[47] M. E. T. Gerards, C. P. R. Baaij, J. Kuper, and M. Kooijman. Higher-order abstraction in hardware descriptions with CλaSH. In P. Kitsos, editor, *Proceedings of the 14th EUROMICRO Conference on Digital System Design, DSD 2011, Oulu, Finland*, pages 495–502, USA, August 2011. IEEE Computer Society. (Cited on pages 10 and 68).

[48] J. Gibbons. An introduction to the bird-meertens formalism. In S. Reeves, editor, *Proceedings of the First New Zealand Formal Program Development Colloquium*, pages 1–12, Hamilton, nov 1994. URL `http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/nzfpdc-squiggol.ps.gz`. (Cited on page 21).

[49] A. Gill. Type-safe Observable Sharing in Haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi: 10.1145/1596638.1596653. URL `http://doi.acm.org/10.1145/1596638.1596653`. (Cited on page 25).

[50] A. Gill and A. Farmer. Deriving an Efficient FPGA Implementation of a Low Density Parity Check Forward Error Corrector. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 209–220, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: http://doi.acm.org/10.1145/2034773.2034804. (Cited on page 87).

[51] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava. In M. Morazán and S.-B. Scholz, editors, *Implementation and Application of Functional Languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-16477-4. doi: 10.1007/978-3-642-16478-1_2. URL `http://dx.doi.org/10.1007/978-3-642-16478-1_2`. (Cited on page 25).

[52] N. J. Gordon, D. J. Salmond, and A. F. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE Proceedings F (Radar and Signal Processing)*, volume 140, pages 107–113. IET, 1993. (Cited on page 27).

[53] S. Guo and W. Luk. Compiling Ruby into FPGAs. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, volume 975 of *Lecture Notes in Computer Science*, pages 188–197. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-60294-1. doi: 10.1007/3-540-60294-1_112. URL `http://dx.doi.org/10.1007/3-540-60294-1_112`. (Cited on page 23).

[54] Z. Guo, B. Buyukkurt, and W. Najjar. Input Data Reuse in Compiling Window Operations Onto Reconfigurable Hardware. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 249–256, New York, NY, USA, 2004. ACM. ISBN 1-58113-806-7. doi: 10.1145/997163.997199. URL `http://doi.acm.org/10.1145/997163.997199`. (Cited on page 86).

[55] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient Hardware Code Generation for FPGAs. *ACM Trans. Archit. Code Optim.*, 5(1):6:1–6:26, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1369402. URL `http://doi.acm.org/10.1145/1369396.1369402`. (Cited on page 16).

[56] R. Gupta and F. Brewer. High-Level Synthesis: A Retrospective. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 13–28. Springer Netherlands, 2008. ISBN 978-1-4020-8588-8. URL `http://dx.doi.org/10.1007/978-1-4020-8588-8_2`. 10.1007/978-1-4020-8588-8_2. (Cited on page 15).

[57] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. (Cited on page 54).

[58] J. D. Hol, T. B. Schon, and F. Gustafsson. On Resampling Algorithms for Particle Filters. In *Nonlinear Statistical Signal Processing Workshop, 2006 IEEE*, pages 79 –82, sept. 2006. doi: 10.1109/NSSPW.2006.4378824. (Cited on page 31).

[59] S. Hong, X. Liang, and P. Djuric. Reconfigurable particle filter design using dataflow structure translation. In *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*, pages 325–330, Oct 2004. doi: 10.1109/SIPS.2004.1363071. (Cited on page 68).

[60] C. Huijs and T. Krol. A formal semantic model to fit SIL for transformational design. In *EUROMICRO 94. System Architecture and Integration. Proceedings of the 20th EUROMICRO Conference.*, pages 100–107, Sep 1994. doi: 10.1109/EURMIC.1994.390401. (Cited on page 21).

[61] *IM5200 Field Programmable Logic Array (FPLA)*. Intersil, 7 1975. (Cited on page 1).

[62] X. Jin. Implementation of the MUSIC Algorithm in CλaSH, June 2014. URL `http://essay.utwente.nl/65225/`. (Cited on page 14).

[63] C. T. Johnston, K. T. Gribbon, and D. G. Bailey. Implementing Image Processing Algorithms on FPGAs. In *Proceedings of the Eleventh Electronics New Zealand Conference, ENZCon'04, Palmerston North*, pages 118–123, 2004. (Cited on page 86).

[64] S. P. Jones, editor. *Haskell 98 Language and Libraries*, volume 13 of *Journal of Functional Programming*. 2003. (Cited on page 10).

[65] L. Jówiak, N. Nedjah, and M. Figueroa. Modern development methods and tools for embedded reconfigurable systems: A survey. *Integr. VLSI J.*, 43(1):1–33, Jan. 2010. ISSN 0167-9260. doi: 10.1016/j.vlsi.2009.06.002. URL `http://dx.doi.org/10.1016/j.vlsi.2009.06.002`. (Cited on page 4).

[66] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standards, 1999. (Cited on page 14).

[67] D. W. Knapp. *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. ISBN 0-13-569252-0. (Cited on page 15).

[68] T. Krol, J. van Meerbergen, C. Niessen, W. Smits, and J. Huisken. The Sprite Input Language – an intermediate format for high level synthesis. In *Design Automation, 1992. Proceedings., [3rd] European Conference on*, pages 186–192, Mar 1992. doi: 10.1109/EDAC.1992.205920. (Cited on page 21).

[69] J. Kuper, C. P. R. Baaij, M. Kooijman, and M. E. T. Gerards. Exercises in architecture specification using CλaSH. In *Proceedings of Forum on Specification and Design Languages, FDL 2010, Southampton, England*, pages 178–183, Gières, France, September 2010. ECSI Electronic Chips & Systems design Initiative. (Cited on page 10).

[70] J. Kuper, C. P. R. Baaij, M. Kooijman, and M. E. T. Gerards. Architecture specifications in CλaSH. In T. J. Kamierski and A. Morawiec, editors, *System Specification and Design Languages*, volume 106 of *Lecture Notes in Electrical Engineering*, pages 191–206. Springer Verlag, New York, December 2011. (Cited on page 10).

[71] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. (Cited on page 54).

[72] M. Lesniak. PASTHA: Parallelizing Stencil Calculations in Haskell. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 5–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-859-9. doi: 10.1145/1708046.1708052. URL `http://doi.acm.org/10.1145/1708046.1708052`. (Cited on page 85).

[73] S.-A. Li, C.-C. Hsu, W.-L. Lin, and J.-P. Wang. Hardware/software co-design of particle filter and its application in object tracking. In *System Science and Engineering (ICSSE), 2011 International Conference on*, pages 87–91, June 2011. doi: 10.1109/ICSSE.2011.5961879. (Cited on pages 34 and 35).

[74] C.-C. Lin et al. *Implementation of H. 264 Decoder in Bluespec SystemVerilog*. PhD thesis, Massachusetts Institute of Technology, 2007. (Cited on page 25).

[75] Q. Liu, T. Todman, W. Luk, and G. Constantinides. Optimizing Hardware Design by Composing Utility-Directed Transformations. *Computers, IEEE Transactions on*, 61(12):1800–1812, Dec 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.205. (Cited on pages 19 and 68).

[76] W. Luzhou, K. Sano, and S. Yamamoto. Domain-Specific Language and Compiler for Stencil Computation on FPGA-Based Systolic Computational-memory Array. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'12, pages 26–39, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28364-2. doi: 10.1007/978-3-642-28365-9_3. URL `http://dx.doi.org/10.1007/978-3-642-28365-9_3`. (Cited on pages 14, 86, and 89).

[77] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *Design Test of Computers, IEEE*, 26(4):18–25, July 2009. ISSN 0740-7475. doi: 10.1109/MDT.2009.83. (Cited on pages 4 and 15).

[78] MATLAB. *Version 8.4.0 (R2014b)*. The MathWorks Inc., Natick, Massachusetts, 2014. (Cited on page 14).

[79] L. Meertens. Algorithmics — Towards programming as a mathematical activity. In J. de Bakker, M. Hazewinkel, and J. Lenstra, editors, *Mathematics and Computer Science*, CWI Monographs Volume 1, pages 289–334. North-Holland Publishing Company, Amsterdam, 1986. (Cited on page 21).

[80] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3): 31–51, 2012. ISSN 0929-5585. doi: 10.1007/s10617-012-9096-8. URL `http://dx.doi.org/10.1007/s10617-012-9096-8`. (Cited on page 16).

[81] P. Middelhoek, G. Mekenkamp, B. Molenkamp, and T. Krol. A transformational approach to VHDL and CDFG based high-level synthesis: a case study. In *Custom Integrated Circuits Conference, 1995., Proceedings of the IEEE 1995*, pages 37–40, May 1995. doi: 10.1109/CICC.1995.518133. (Cited on page 21).

[82] E. Molenkamp, G. E. Mekenkamp, J. Hofstede, and T. Krol. SIL: an intermediate for syntax based VHDL synthesis. *Proc. of the VIUF Spring*, 95:5–1, 1995. (Cited on page 21).

[83] M. Naylor and C. Runciman. The Reduceron Reconfigured. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 75–86, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863556. URL `http://doi.acm.org/10.1145/1863543.1863556`. (Cited on page 25).

[84] A. Niedermeier, R. Wester, C. P. R. Baaij, J. Kuper, and G. J. M. Smit. Comparing CλaSH and VHDL by implementing a dataflow processor. In *Proceedings of the Workshop on PROGram for Research on Embedded Systems and Software (PROGRESS 2010), Veldhoven, The Netherlands*, pages 216–221, Utrecht, November 2010. Technology Foundation STW. (Cited on pages 10 and 42).

[85] R. Nikhil. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 129–146. Springer Netherlands, 2008. ISBN 978-1-4020-8587-1. doi: 10.1007/978-1-4020-8588-8_8. URL `http://dx.doi.org/10.1007/978-1-4020-8588-8_8`. (Cited on page 25).

[86] J. O'Donnell. Overview of Hydra: a concurrent language for synchronous digital circuit design. *International Journal of Information*, 9(2):249–264, March 2006. URL `http://eprints.gla.ac.uk/3461/`. (Cited on page 24).

[87] A. Oetken, S. Wildermann, J. Teich, and D. Koch. A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 234–239, Aug 2010. doi: 10.1109/FPL.2010.54. (Cited on pages 34 and 35).

[88] K. K. Parhi. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007. (Cited on page 87).

[89] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The CMU Design Automation System: An Example of Automated Data Path Design. In *Proceedings of the 16th Design Automation Conference*, DAC '79, pages 73–80, Piscataway, NJ, USA, 1979. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=800292.811694`. (Cited on page 15).

[90] D. Parker and K. Parhi. Low-area/power parallel fir digital filter implementations. *Journal of VLSI signal processing systems for signal, image and video technology*, 17(1): 75–92, 1997. ISSN 0922-5773. doi: 10.1023/A:1007901117408. (Cited on page 87).

[91] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):661–679, Jun 1989. ISSN 0278-0070. doi: 10.1109/43.31522. (Cited on page 15).

[92] D. Pellerin and M. Holley. *Practical Design Using Programmable Logic*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-723834-7. (Cited on page 2).

[93] A. Peymandoust and G. De Micheli. Using symbolic algebra in algorithmic level DSP synthesis. In *Design Automation Conference, 2001. Proceedings*, pages 277–282, 2001. doi: 10.1109/DAC.2001.156151. (Cited on page 19).

[94] J. Proakis and D. Manolakis. *Digital Signal Processing*. Prentice Hall international editions. Pearson Prentice Hall, 2007. ISBN 9780131873742. URL `http://books.google.nl/books?id=H_5SAAAAMAAJ`. (Cited on page 92).

[95] M. Puschel and J. Moura. Algebraic Signal Processing Theory: Cooley-Tukey Type Algorithms for DCTs and DSTs. *Signal Processing, IEEE Transactions on*, 56(4):1502–1521, April 2008. ISSN 1053-587X. doi: 10.1109/TSP.2007.907919. (Cited on page 20).

[96] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840306. (Cited on pages 19, 20, and 68).

[97] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *Journal of High Performance Computing and Applications*, 18:21–45, 2004. (Cited on page 19).

[98] J. Robelly, G. Cichon, H. Seidel, and G. Fettweis. Automatic code generation for SIMD DSP architectures: An algebraic approach. In *Parallel Computing in Electrical Engineering, 2004. PARELEC 2004. International Conference on*, pages 372–375, Sept 2004. doi: 10.1109/PCEE.2004.17. (Cited on page 20).

[99] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(1):17–32, Jan 2004. ISSN 0278-0070. doi: 10.1109/TCAD.2003.819898. (Cited on page 24).

[100] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth. *Parallel and Distributed Systems, IEEE Transactions on*, 25(3):695–705, March 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.51. (Cited on page 86).

[101] R. Sass and A. G. Schmidt. *Embedded Systems Design with Platform FPGAs: Principles and Practices*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010. ISBN 9780080921785, 9780123743336. (Cited on page 4).

[102] M. Schmidt, M. Reichenbach, and D. Fey. A Generic VHDL Template for 2D Stencil Code Applications on FPGAs. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*, pages 180–187, April 2012. doi: 10.1109/ISORCW.2012.39. (Cited on pages 86, 97, and 98).

[103] M. Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 104–112, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802026. URL `http://doi.acm.org/10.1145/800055.802026`. (Cited on page 23).

[104] M. Sheeran. Designing regular array architectures using higher order functions. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 220–237. Springer Berlin Heidelberg, 1985. ISBN 978-3-540-15975-9. doi: 10.1007/3-540-15975-4_39. URL `http://dx.doi.org/10.1007/3-540-15975-4_39`. (Cited on page 68).

[105] M. Sheeran. Hardware design and functional programming: a perfect match. 11(7): 1135–1158, jul 2005. URL `http://www.jucs.org/jucs_11_7/hardware_design_and_functional`. (Cited on pages 23 and 25).

[106] G. J. M. Smit, J. Kuper, and C. P. R. Baaij. A mathematical approach towards hardware design. In P. M. Athanas, J. Becker, J. Teich, and I. Verbauwhede, editors, *Dagstuhl Seminar on Dynamically Reconfigurable Architectures, Dagstuhl, Germany*, volume 10281 of *Dagstuhl Seminar Proceedings*, page 11, Dagstuhl, Germany, December 2010. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI). (Cited on page 10).

[107] M. Sonka, V. Hlavac, and R. Boyle. *Image Processing, Analysis, and Machine Vision*. Thomson-Engineering, 2007. ISBN 049508252X. (Cited on page 85).

[108] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI signal processing systems for signal, image and video technology*, 28(1-2):7–27, 2001. ISSN 0922-5773. doi: 10.1023/A:1008155020711. (Cited on page 3).

[109] The GHC Team. The Glasgow Haskell Compiler, 2012. URL `http://www.haskell.org/ghc/`. (Cited on page 10).

[110] F. Van Nee. To a new hardware design methodology: A case study of the cochlea model, March 2014. URL `http://essay.utwente.nl/64835/`. (Cited on page 10).

[111] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, May 2010. doi: 10.1109/FCCM.2010.28. (Cited on pages 16 and 17).

[112] J. Voeten. On the fundamental limitations of transformational design. *ACM Trans. Des. Autom. Electron. Syst.*, 6(4):533–552, Oct. 2001. ISSN 1084-4309. doi: 10.1145/502175.502181. URL `http://doi.acm.org/10.1145/502175.502181`. (Cited on page 19).

[113] Y. Voronenko and M. Puschel. Algebraic Signal Processing Theory: Cooley-Tukey Type Algorithms for Real DFTs. *Signal Processing, IEEE Transactions on*, 57(1):205–222, Jan 2009. ISSN 1053-587X. doi: 10.1109/TSP.2008.2006152. (Cited on page 20).

[114] M. H. Wiggers, M. J. Bekooij, and G. J. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 658–663. IEEE, 2007. (Cited on page 55).

[115] S. Woop, E. Brunvand, and P. Slusallek. HWML: RTL/Structural Hardware Description using ML. Technical report, Technical report, Computer Graphics Lab, Saarland University, 2006. (Cited on page 24).

[116] Xilinx. Zynq-7000 Silicon Devices, . URL `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices.html`. (Cited on page 4).

[117] Xilinx. Xilinx CORE generator, . URL `http://www.xilinx.com/tools/coregen.htm`. (Cited on page 9).

[118] Xilinx. Xilinx Virtex Ultrascale, . URL `http://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale.html`. (Cited on page 2).

[119] *XC2064/XC2018 Logic Cell Array– Product Specification*. Xilinx, 1985. (Cited on page 2).

[120] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001. (Cited on page 20).

[121] H. Yu and M. Leeser. Automatic Sliding Window Operation Optimization for FPGA-Based Computing Boards. In *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 76–88, April 2006. doi: 10.1109/FCCM.2006.29. (Cited on pages 86 and 89).

List of Publications

[RW:1] R. Wester, C. P. R. Baaij, and J. Kuper. A two step hardware design method using CλaSH. In *22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway*, pages 181–188, USA, August 2012. IEEE Computer Society.

[RW:2] R. Wester, D. Sarakiotis, E. Kooistra, and J. Kuper. Specification of APERTIF Polyphase Filter Bank in CλaSH. In *Communicating Process Architectures 2012, Scotland*, pages 53–64, United Kingdom, August 2012. Open Channel Publishing.

[RW:3] K. C. H. Blom, R. Wester, A. B. J. Kokkeler, and G. J. M. Smit. Low-cost multichannel underwater acoustic signal processing testbed. In *7th IEEE Sensor Array and Multichannel Signal Processing Workshop (SAM), 2012, Hoboken, NJ, USA*, pages 505–508, USA, July 2012. IEEE.

[RW:4] R. Wester and J. Kuper. A space/time tradeoff methodology using higher-order functions. In *22nd International Conference on Field Programmable Logic and Applications, FPL2013, Porto*, pages 1–2, USA, 2013. IEEE Computer Society.

[RW:5] R. Wester and J. Kuper. Design space exploration of a particle filter using higher-order functions. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture Notes in Computer Science*, pages 219–226. Springer Verlag, London, 2014.

[RW:6] R. Wester and J. Kuper. Deriving Stencil Hardware Accelerators from a Single Higher-Order Function. In *Communicating Process Architectures 2014, England*, United Kingdom, August 2014. Open Channel Publishing.

[RW:7] J. Kuper and R. Wester. N Queens on an FPGA: Mathematics, Programming, or both? In *Communicating Process Architectures 2014, England*, United Kingdom, August 2014. Open Channel Publishing.

[RW:8] A. Niedermeier, R. Wester, K. C. Rovers, C. P. R. Baaij, J. Kuper, and G. J. M. Smit. Designing a dataflow processor using CλaSH. In *28th Norchip Conference, NORCHIP 2010, Tampere, Finland*, page 69. IEEE Circuits and Systems Society, November 2010.

[RW:9] A. Niedermeier, R. Wester, C. P. R. Baaij, J. Kuper, and G. J. M. Smit. Comparing CλaSH and VHDL by implementing a dataflow processor. In *Proceedings of the Workshop on PROGram for Research on Embedded Systems and Software (PROGRESS 2010), Veldhoven, The Netherlands*, pages 216–221, Utrecht, November 2010. Technology Foundation STW.

ISBN 978-90-365-3887-9

9 789036 538879