Lustransformaties voor de geoptimaliseerde generatie van herconfigureerbare hardware

Loop Transformations for the Optimized Generation of Reconfigurable Hardware

Harald Devos

Promotoren: prof. dr. ir. D. Stroobandt, prof. dr. ir. J. Van Campenhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Elektrotechniek

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen
Academiejaar 2007 - 2008

UNIVERSITEIT GENT

ISBN 978-90-8578-191-2
NUR 959
Wettelijk depot: D/2008/10.500/10

Promotoren

prof. dr. ir. D. Stroobandt
prof. dr. ir. J. Van Campenhout

Universiteit Gent
Faculteit Ingenieurswetenschappen
Vakgroep Elektronica en Informatiesystemen (ELIS)
Onderzoeksgroep Parallelle Informatiesystemen (PARIS)
Sint-Pietersnieuwstraat 41
B-9000 Gent
België

De auteur genoot tijdens zijn onderzoeksactiviteiten van een beurs als FWO-aspirant.

Erkenntnis is nicht aus Lehren zu vermitteln,
sondern kann nur durch eigene Erfahrung erworben werden.
Hermann Hesse

Dankwoord
Acknowledgements

Dit werk zou niet mogelijk geweest zijn zonder de de financiële ondersteuning van het FWO (Fonds voor Wetenschappelijk Onderzoek – Vlaanderen) en het IWT (Instituut voor de aanmoediging van Innovatie door Wetenschap en Technologie in Vlaanderen).

Verder wil ik mijn promotoren bedanken omdat ze me de kans gaven aan deze universiteit onderzoek te verrichten, me hierbij de vrijheid gaven om mijn eigen weg in de onderzoekswereld te zoeken en richting gaven wanneer ik dreigde verloren te lopen.

Voor de dagelijkse werking van een onderzoeksgroep zijn postdocs minstens zo belangrijk als professoren. Mark, bedankt voor de voortreffelijke leiding van het RESUME-project, het van tijd tot tijd meebrengen van ovenexperimenten en voor het feit dat we altijd je bureau konden binnenwandelen met de meest uiteenlopende problemen of ideeën. Kristof Beyls introduceerde me in de wereld van lustransformaties en het polyedrische model. Hij nam me mee naar de doctoraatsverdedigingen van Sven Verdoolaege en Sylvain Girbal — de eerste na een onvergetelijke treinreis — die ik nog steeds als mijlpalen in mijn onderzoek beschouw.

Sven Verdoolaege wil ik bedanken voor de vruchtbare discussies omtrent het benaderen van quasi-veeltermen en het implementeren van een deel van de technieken beschreven in hoofdstuk 4.

Many thanks go to Sylvain Girbal, Albert Cohen and Nicolas Vasilache for providing the WRaP-IT/URUK tool suite and giving support. I also would like to thank Cédric Bastoul for writing CLooG. I hope we will meet soon.

Dank aan alle collega's en oud-collega's van PARIS voor het nalezen

van allerhande schrijfsels, het ontwikkelen van software-drivers en regressietesten, het uitvoeren van systeemintegratie en vermogensmetingen, het aan de praat krijgen van nieuwe hardware-platformen, het animeren van review- en andere vergaderingen, het uitwisselen van ideeën en het brainstormen over uiteenlopende onderwerpen, het doorgeven van scriptjes, artikels, templates, . . . , en bovenal de aangename werksfeer. Hendrik en Philippe wens ik een succesvolle toekomst met Sigasi. Ook aan alle anderen aangename en vruchtbare werk- en onderzoekservaringen toegewenst.

Ronny, Michiel en Wim verdienen een bedankje voor het onderhouden van de hardware-, software- en netwerkinfrastructuur. Jeroen, Marnix, Rita en Wouter voor de hulp bij administratieve en logistieke beslommeringen en mijn speurtochten door de bibliotheek.

De leden van de OAP-vergadering en het organiserend comité van het doctoraatssymposium lieten me toe mijn horizon te verbreden buiten het technisch-wetenschappelijke domein. Dankzij het GUK (Gents Universitair Koor) waren de voorbije jaren aan de universiteit ook een culturele verrijking. Onnodig te vermelden dat dit ook een uitbreiding van de vriendenkring met zich meebracht.

Vrienden en familie wil ik bedanken voor hun interesse en steun op alle vlakken. Mijn ouders gaven me de kans zonder te veel andere bekommernissen aan dit doctoraat te werken.

Ten slotte wil ik Krista bedanken omdat ik alles met haar mocht en mag delen.

Harald Devos
Gent, 11 februari 2008

Examencommissie

prof. Ronny Verhoeven, voorzitter
Onderwijsdirecteur Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Erik D'Hollander, secretaris
Vakgroep Elektronica en Informatiesystemen (ELIS)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Albert Cohen
ALCHEMY Group, INRIA Saclay Île de France
Parc Club Orsay Université
France

prof. Leo Storme
Vakgroep Zuivere Wiskunde en Computeralgebra
Faculteit Wetenschappen
Universiteit Gent

prof. Dirk Stroobandt, promotor
Vakgroep Elektronica en Informatiesystemen (ELIS)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Jan Van Campenhout, promotor
Vakgroep Elektronica en Informatiesystemen (ELIS)
Faculteit Ingenieurswetenschappen
Universiteit Gent

prof. Ingrid Verbauwhede
Departement Elektrotechniek (ESAT)
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

Samenvatting

FPGA's De FPGA (Field Programmable Gate Array, in het veld programmeerbare poortenmatrix) is de laatste twintig jaar uitgegroeid tot een standaardcomponent in het ontwerp van digitale schakelingen. Doordat FPGA's herconfigureerbaar zijn is het mogelijk een ontwerp nog na de productie te wijzigen en kan de hoge kost voor de fabricage van maskers vermeden worden. Dit verkort ook de ontwerp- en productiecyclus. De vaste prijs per chip maakt een implementatie op een FPGA goedkoper dan in een ASIC (Application Specific Integrated Circuit, toepassingsspecifieke geïntegreerde schakeling) voor lage en middelgrote productievolumes. Voor de herconfigureerbaarheid moet echter een prijs betaald worden: de grote hoeveelheid programmeerbare interconnecties en logica brengt een lagere kloksnelheid en een minder efficiënt gebruik van de beschikbare chipoppervlakte met zich mee.

Het geheugenknelpunt De hoeveelheid geheugen op een FPGA volstaat in vele gevallen niet om te voldoen aan de behoeften van moderne applicaties. De meeste multimediatoepassingen zijn voorbeelden van dergelijke data-intensieve toepassingen. Een extern geheugen is nodig om ze te kunnen implementeren. De toegangen naar een extern geheugen verlopen echter trager dan de transacties binnen de FPGA, zowel op het gebied van latentie als van bandbreedte. Hierdoor dreigt dit externe geheugen een knelpunt te worden van het ontwerp: als tijd verloren gaat door het wachten op datatransfers van en naar dit geheugen kan de grote rekenkracht van een FPGA niet benut worden.

Een oplossing om de datatrafiek tussen de FPGA en het externe geheugen te verminderen is het gebruik van buffers op de FPGA. Gegevens die meermaals gebruikt worden hoeven dan slechts eenmaal gekopieerd te worden naar de buffer en kunnen daarna uit dit buffergeheugen gelezen worden. Op elk moment kan slechts een beperkte hoeveelheid data in de buffer aanwezig zijn. Daarom moet getracht

worden alle bewerkingen op een bepaald data-element te groeperen in de tijd, het zogenaamde verbeteren van de temporele lokaliteit. Hiervoor kunnen lustransformaties gebruikt worden die de volgorde van de berekeningen veranderen. Lustransformaties worden veelvuldig gebruikt voor de optimalisatie van software, meer bepaald het cachegedrag. In hardware-ontwerpomgevingen, bijvoorbeeld hoog-niveau-synthesetools, beperkt het toepassen van lustransformaties zich voornamelijk tot het verhogen van de hoeveelheid parallellisme. Voor lokaliteit is er tot nog toe (te) weinig aandacht.

Dit proefschrift Dit proefschrift behandelt lustransformaties voor het verbeteren van de datalokaliteit en het integreren ervan in het hardware-ontwerpproces. Drie aspecten worden hierbij nader onderzocht: (1) het samenstellen van lange reeksen lustransformatiestappen, (2) het zoeken naar bovengrenzen op (quasi-)veeltermfuncties over discrete domeinen voor het statisch evalueren van implementatie-eigenschappen, en ten slotte (3) het genereren van hardware-beschrijvingen uit de polyedrische voorstelling die in dit werk gebruikt wordt om lustransformaties te beschrijven en toe te passen.

Het polyedrisch model Het iteratiedomein van een programmastatement is de verzameling van alle waarden die de iteratievector, de vector samengesteld uit de iteratoren van de lussen omheen een statement, aanneemt tijdens de uitvoering. Wanneer we ons beperken tot lusnesten waarin elke lusgrens een lineaire combinatie is van de iteratoren van de omhullende lussen en van gehele parameters dan komt elk iteratiedomein overeen met een $\mathbb{Z}$-polyeder. Dit is een verzameling bestaande uit de geheeltallige punten in een polyeder of meer algemeen de doorsnede van een geheeltallig rooster met een polyeder. De programmavoorstelling die hierop gebaseerd is heet het polyedrisch (of geometrisch) model.

Sequenties van lustransformaties In een polyedrische voorstelling worden lustransformaties beschreven als bewerkingen op matrices en vectoren. Dit maakt het samenstellen van transformaties eenvoudiger dan wanneer een tekstuele of abstracte-syntax-boomvoorstelling gebruikt wordt. In dit proefschrift wordt onderzocht wat de invloed van de volgorde van transformatiestappen is en hoe korte aaneenschakelingen van transformatiestappen samengesteld kunnen worden tot langere reeksen.

Afbakenen van het bereik van quasi-veeltermen Om de keuze van lustransformaties te sturen is het nodig om eigenschappen te kunnen evalueren die het resultaat zijn van deze transformaties. Het schatten of berekenen van dergelijke eigenschappen zonder het uitvoeren van een implementatie is mogelijk door statische programma-analyse. Voor de programma's die kunnen voorgesteld worden in het polyedrisch model, kunnen veel analyseproblemen herleid worden tot het tellen van het aantal geheeltallige punten in een polyeder of het tellen van het aantal geheeltallige punten in verzamelingen beschreven door zogenaamde Presburger-formules. Dit zijn stelsels opgebouwd uit lineaire ongelijkheden samengesteld met logische operatoren en kwantoren. De oplossing van een dergelijk telprobleem kan beschreven worden als een (stuksgewijze) Ehrhart quasi-veelterm in functie van de parameters van het telprobleem. In sommige analyseproblemen is het nodig een bovengrens te vinden voor de oplossing van een telprobleem over een bepaald discreet domein van parameterwaarden (de variabelen in de quasi-veelterm). Als bijvoorbeeld de hoeveelheid levende data in een programma uitgedrukt wordt in functie van het tijdstip in de uitvoering (de iteratorwaarden), dan is de minimale vereiste hoeveelheid geheugen om een uitvoering mogelijk te maken het maximum hiervan over alle mogelijke uitvoeringstijdstippen.

In dit werk worden verschillende manieren voorgesteld om grenzen op het bereik van quasi-veeltermfuncties te bepalen. Eerst wordt aangetoond dat de extrema van veeltermfuncties over een continu domein een goede benadering zijn voor de extrema over de geheeltallige punten binnen dit domein, op voorwaarde dat het domein voldoende groot is en de graad van de veelterm voldoende laag. Partiële evaluatie voor een deel van de veranderlijken kan gebruikt worden om de nauwkeurigheid van deze benadering te verhogen. Door het toevoegen van extra veranderlijken kunnen quasi-veeltermen omgezet worden naar veeltermen zodat deze benaderingsmethode ook hier bruikbaar wordt. Voor kleine domeinen is de exacte methode, namelijk het evalueren van de quasi-veelterm in alle discrete punten van het domein, sneller dan de voorgestelde methoden die een continue benadering gebruiken. Daarom werden hybride methoden onderzocht die voor kleine domeinen de exacte methode gebruiken en voor grotere domeinen een van de continue benaderingen hierboven beschreven. Dit blijkt een goede combinatie van de verschillende methoden te zijn (op het gebied van nauwkeurigheid en rekentijd) met slechts een kleine kost voor de selectie van de methode voor elk domein.

Hardware-generatie vanuit een polyedrische voorstelling Lustransformaties beïnvloeden niet alleen de datatransfers maar hebben ook een invloed op de controlecomplexiteit van een implementatie. Dit manifesteert zich meestal slechts na het verfijnen van een ontwerp tot op een synthetiseerbaar niveau, hetgeen het verkennen van de ontwerpruimte van mogelijke lustransformaties bemoeilijkt. Daarom is het nuttig lustransformaties te integreren in hoog-niveau-synthesetools. In dit proefschrift wordt een hardware-architectuur voorgesteld die toelaat om een hardwarebeschrijving te genereren vanuit een polyedrische voorstelling. Verschillende afwegingen tussen oppervlakte en kloksnelheid zijn mogelijk. De code generator, CLooGVHDL, werd getest aan de hand van het genereren van varianten van een inverse discrete wavelet-transformatie. De syntheseresultaten zijn beter dan die van de commerciële hoog-niveau-synthesetool Impulse C en competitief met die van de Celoxica Handel-C compiler.

Besluit In dit proefschrift worden verschillende technieken voorgesteld die bruikbaar zijn voor het vermijden van geheugenknelpunten in hardware-ontwerp. Het opstellen van lustransformatiesequenties om de lokaliteit te verbeteren, het zoeken naar grenzen op het bereik van quasi-veeltermen om het resultaat van dergelijke transformaties te evalueren en ten slotte het genereren van hardware vanuit de polyedrische voorstelling die voor deze transformaties gebruikt wordt.

Summary

FPGAs In the last two decades, the Field Programmable Gate Array (FPGA) has become a standard component in digital design. Its reconfigurability allows to update a design after production and avoids the cost of producing masks for wafer processing. As a result FPGAs come at a fixed cost per chip, which makes FPGA implementations cheaper than ASICs (Application Specific Integrated Circuits) for small and moderate production volumes. The penalty paid for the reconfigurability is a lower clocking frequency and a lower chip density. Still, FPGAs can outperform instruction set processors when the available parallelism can be exploited.

The memory bottleneck Multimedia applications have emerged on all kind of devices, from desktop computers up to PDAs and cellphones. These applications are not only computation-intensive but also data-intensive, which means a large amount of memory is needed. To implement data-intensive applications on FPGAs off-chip memory is needed. The accesses to this memory are slower (bandwidth and latency) than accesses to on-chip memory and are a potential bottleneck. The high computational power of a FPGA can not be exploited if a design suffers a memory bottleneck.

A memory hierarchy should be constructed to decrease the number of off-chip transactions. If a data element is copied to an on-chip buffer this data element can be used several times with only one access to the external memory. The buffers have a limited size and cannot contain a copy of all data at the same time. Therefore, the different accesses to a data element should be close together in time, i.e. exhibit a good temporal locality. Loop transformations are a means to improve the data locality by changing the execution order of computations and data accesses. This technique is commonly used for software optimizations, in particular optimization of the cache behavior. Current high-level

synthesis environments for hardware design lack support to implement data-intensive applications on heterogeneous memory systems. They focus rather on parallelism than on locality.

This thesis This thesis addresses the memory hierarchy problem to high-level transformations of loop structures and the integration of these transformations in the hardware design flow. This work focuses on three aspects: (1) the composition of long transformation sequences, (2) the search for bounds on polynomials and quasi-polynomials over discrete domains for static program analysis and finally (3) the generation of hardware descriptions from a polyhedral representation, used to describe and apply loop transformations.

The polyhedral model We restrict ourselves to loop nests in which all loop bounds are linear expressions in some integer parameters and the iterators of surrounding loops. As a result, the iteration domain of a program statement, i.e. the set of all possible values of the iteration vector (the vector composed of the iterators of all surrounding loops), can be described as a $\mathbb{Z}$-polytope, i.e. the set of integer points, or more generally the set of points in an integer lattice, that lie inside a rational polytope. The program representation based on this property is called the polyhedral model.

Sequences of loop transformations In a polyhedral representation loop transformations are described as operations on matrices and vectors. This eases the composition of transformation sequences in comparison with a textual or abstract syntax tree representation. In this thesis the influence of the order of transformation steps is studied together with the possibility to build long transformation sequences by combining short sequences.

Bounds on quasi-polynomials Static program analysis involves the estimation or computation of program properties without execution. This can be used to guide the choice of loop transformations to apply. For programs that can be represented in the polyhedral model, many analysis problems can be reduced to counting the number of integer points in a polytope or the number of integer solutions to so-called Presburger formulas, i.e. systems of inequalities connected with logical operators and quantifiers. The solution of such a counting problem can

be expressed as a (piecewise) Ehrhart quasi-polynomial that is a function of the parameters of the problem. In some cases an upper bound on a quasi-polynomial over a discrete domain is needed. If, for example, the number of live data elements is expressed as a function of the point of execution of a program, then the minimal memory requirement of that program is the maximum over all points of execution.

This thesis presents several methods to find bounds on the range of quasi-polynomials. First, it is proved that the extrema of a polynomial over a continuous domain are a good approximation of the extrema over the integer points in that domain, provided that the degree of the polynomial is sufficiently low and the size of the domain is sufficiently large. Partial evaluation for a selection of the variables allows to improve the accuracy of this approximation. By introducing new variables quasi-polynomials can be transformed into polynomials to make these approximations useful to find bounds on quasi-polynomials. For small domains the exact method that evaluates the quasi-polynomial in all discrete points of the domain is faster than the presented methods. Therefore, hybrid methods have been constructed that apply the exact method on small domains and apply one of the presented methods that use continuous domain approximations on larger domains. These hybrid methods appear to combine the strengths of the different methods with only a small overhead introduced by the selection mechanism that selects the method to apply in a domain.

Hardware generation from the polyhedral model Loop transformations not only influence the data transfers but also the control complexity of an implementation. The impact on the hardware performance can typically only be quantified after refinement to a synthesizable level. This hinders an exploration of the space of loop transformations. Therefore, it would be beneficial to integrate loop transformations in high-level synthesis tools. A hardware architecture is presented with close correspondence to the polyhedral representation, which allows to generate hardware starting from a polyhedral representation. Parallelism is supported by the architecture but not implemented yet in the current code generator, called CLooGVHDL. Different trade-offs between area and clock speed are investigated. CLooGVHDL has been tested by generating variants of an inverse discrete wavelet transform. The results outperform those of the commercial high-level synthesis tool Impulse C and are competitive to those of the Celoxica Handel-C compiler.

Conclusion This thesis presents several techniques that are useful in hardware design to avoid a memory bottleneck. The composition of loop transformation sequences to improve the data locality, the search for bounds on quasi-polynomials to evaluate the result of such transformations and finally, the generation of hardware from a polyhedral representation used to apply the loop transformations.

Contents

List of abbreviations

1-D	1-Dimensional
2-D	2-Dimensional
AD	Arithmetic Decoder
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
AST	Abstract Syntax Tree
BRD	Backward Reuse Distance
CIF	Common Intermediate Format (288×352 pixels, 30000/1001 (≈ 29.97) frames/s, YCbCr 4:2:0)
CLooG	Chunky Loop Generator
CPU	Central Processing Unit
DDR	Dual/Double Data Rate
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
DTSE	Data Transfer and Storage Exploration
DWT	Discrete Wavelet Transform
EDIF	Electronic Design Interchange Format
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
gcd	Greatest Common Divider
GOP	Group Of Pictures
GPL	GNU General Public License
GPP	General Purpose Processor
GPU	Graphics Processing Unit

HDL	Hardware Description Language
HW	HardWare
IC	Integrated Circuit
IDWT	Inverse Discrete Wavelet Transform
IF	InterFace
iff	If and only if
ILP	Instruction Level Parallelism
ILP	Integer Linear Programming
IMEC	Interuniversitair Micro-Elektronica Centrum / Interuniversity Microelectronics Center
I/O, IO	Input Output
IOE	Input Output Element
IP	Intellectual Property
KPN	Kahn Process Network
LAB	Logic Array Block
LB	Line-Based
LBT	Line-Based and Tiled
LE	Logic Element
LE	Set of Living Elements
LHS	Left-Hand Side
LRU	Least Recently Used
LUT	Look-Up Table
MAC	Multiply ACcumulate
MRE	Maximal Relative Error
NPA	Non-Programmable Accelerator
ORC	Open Research Compiler
PSNR	Peak Signal to Noise Ratio
QCIF	Quarter CIF
QoS	Quality of Service
RAM	Random Access Memory
RaR	Read after Read
RaW	Read after Write
RC	Row-Column(-wise)
RE	Relative Error
RHS	Right-Hand Side
RTL	Register Transfer Level
RMSE	Root Mean Square Error

SCoP	Static Control Part
SDRAM	Synchronous Dynamic Random Access Memory
SLO	Suggestions for Locality Optimizations
SNR	Signal to Noise Ratio
SoC	System on Chip
SOPC	System On Programmable Chip
SUIF	Stanford University Intermediate Format
SW	SoftWare
Th	Threshold
TLM	Transaction-Level Modeling
URUK	Unified Representation Universal Kernel
VGA	Video Graphics Array (computer display standard)
VHDL	Very high speed integrated circuits Hardware Description Language
VLIW	Very Long Instruction Word (processor)
WaR	Write after Read
WaW	Write after Write
WED	Wavelet Entropy Decoder
WHIRL	Winning Hierarchical Intermediate Representation Language
WRaP-IT	WHIRL Represented as Polyhedra - Interface Tool

List of notations

$\mathbb{N}_0,\ \mathbb{Z}_0$	$\mathbb{N} \setminus \{0\},\ \mathbb{Z} \setminus \{0\}$
$a\mathbb{Z}$	The set of multiples of a: $a\mathbb{Z} = \{m \in \mathbb{Z} \mid m = aq \text{ for some } q \in \mathbb{Z}\}$
$\lfloor \cdot \rfloor$	Integer part: $\lfloor x \rfloor = y \Leftrightarrow y \in \mathbb{Z} \wedge x - 1 < y \leq x$
$\{\cdot\}$	Fractional part: $\{x\} = x - \lfloor x \rfloor$
$\lvert \cdot \rvert$	Absolute value
$\lvert S \rvert$	Number of elements of the set S (cardinality)
$(cond\ ?\ a : b)$	$(cond\ ?\ a : b) = \begin{cases} a & \text{if } cond, \\ b & \text{otherwise.} \end{cases}$
%	$\bmod$, modulo operator
$a \equiv b \pmod{d}$	$a \bmod d = b \bmod d$
δ_{ij}	Kronecker delta function: $\delta_{ij} = \begin{cases} 1 & \text{if } i = j \ , \\ 0 & \text{if } i \neq j \ . \end{cases}$
`S1`	Statement `S1`
$D_{\texttt{S1}}$	Depth of statement `S1`
$\mathcal{D}_{\texttt{S1}}$	Iteration domain of `S1`
$\mathcal{I}_{\texttt{S1}}$	Iteration vector of `S1`
$\mathcal{D}_{\texttt{S1}\delta\texttt{S2}}$	Dependence Domain of dependences from statement `S1` to `S2`
$LE_A(t)$	Set of Live Elements of array A at time t
$\theta_{\texttt{S1}}$	Schedule function of statement `S1`
$\overline{F}_{\mathrm{b}}, \overline{G}_{\mathrm{b}}, \ldots$	Upper bound on $f, g, \ldots$ using Bernstein expansion
$\underline{F}_{\mathrm{b}}, \underline{G}_{\mathrm{b}}, \ldots$	Lower bound on $f, g, \ldots$ using Bernstein expansion
$\overline{F}_{\mathrm{c}}, \overline{G}_{\mathrm{c}}, \ldots$	Continuous-domain maximum of $f, g, \ldots$
$\underline{F}_{\mathrm{c}}, \underline{G}_{\mathrm{c}}, \ldots$	Continuous-domain minimum of $f, g, \ldots$
$\overline{F}_{\mathrm{d}}, \overline{G}_{\mathrm{d}}, \ldots$	Discrete-domain maximum of $f, g, \ldots$
$\underline{F}_{\mathrm{d}}, \underline{G}_{\mathrm{d}}, \ldots$	Discrete-domain minimum of $f, g, \ldots$

RE	Relative error of continuous- versus discrete-domain range of a polynomial
RE_x	RE after partial evaluation with x kept as free variable
RE_{rivl}	Maximal RE of a polynomial computed with Rivlin's method
MRE	Maximal possible value of RE for any polynomial of a given degree for a given domain size
$\mathcal{S}_N$	$\mathcal{S}_N = \{S \subset \{0, 1, \ldots, N\}, \lvert S\rvert = g + 1\}$
$L_{S,i}(x)$	Lagrange interpolant, defined in Equation (4.11) : $L_{S,i}(x) = \prod_{0 \leq j \leq g, j \neq i} \frac{x - x_j}{x_i - x_j}$, with $i \in \{0, 1, \ldots, g\}$, $S = \{x_0, \ldots, x_g\} \in \mathcal{S}_N$
$M_N(x)$	$M_N(x) = \min_{S \in \mathcal{S}_N} \sum_{i=0}^{g} \lvert L_{S,i}(x)\rvert$
K, K_N, $K(g, N)$	$K(g, N) = \max_{x \in [0,N]} M_N(x)$

Chapter 1

Introduction

The Field Programmable Gate Array (FPGA) has become a standard component in digital design. Its reconfigurability allows to update a design after production and avoids the cost of producing masks for wafer processing. This makes FPGA implementations cheaper than ASICs (Application Specific Integrated Circuits) for small and moderate production volumes. The penalty paid for the reconfigurability is a lower clocking frequency and a lower chip density. Still, FPGAs can outperform processors when the massive available parallelism can be exploited.
Multimedia applications have emerged on all kind of devices, from desktop computers up to PDAs and cellphones. These applications are not only computation-intensive but also data-intensive, which means a large amount of memory is needed. To implement data-intensive applications on FPGAs off-chip memory is needed. The accesses to this memory are a potential bottleneck. To avoid a memory bottleneck, a memory hierarchy should be constructed and loop transformations are needed to improve the data locality. Current high-level synthesis environments lack support to implement data-intensive applications on heterogeneous-memory systems. They rather focus on parallelism.

1.1 Reconfigurable hardware

1.1.1 The benefit of reconfigurable hardware

General Purpose Processors (GPP) and their low-end variants, microcontrollers, offer a standard platform for implementing an algorithm. Thanks to mass production they are relatively cheap and they can eas-

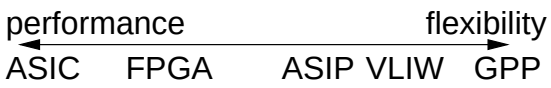


Figure 1.1: Trade-off between flexibility and performance.

ily be (re-)programmed by compiling (high-level) software using an appropriate (standard) compiler. Only when the desired performance cannot be reached (speed, power, ...) another platform will be chosen. Very Long Instruction Word processors (VLIW), Digital Signal Processors (DSP) and Application Specific Instruction set Processors (ASIP), e.g., Graphics Processing Units (GPU), are more specialized towards a specific application (domain), and can thus offer a higher performance (within the application domain) at the cost of a lower performance for code that does not match the specialized architecture. Also the writing of software becomes more complex.

Application Specific Integrated Circuits (ASICs) give the designer total freedom on how to use the available silicon real estate. The hardware architecture can now be built targeted to the application, which may lead to the highest possible performance. However, the cost of ASIC design and manufacturing drastically increases as the technology evolves to lower feature sizes. Only for very large production volumes the cost of producing masks is affordable. The long production time between the tape-out of a design and the delivery of test chips makes iterating over a design slow and expensive. Next to the large production cost and time, there is a high non-recurring engineering cost. The low-level design is very labor intensive and thus also increases the cost of iterating over a design.

Reconfigurable hardware offers a compromise between the flexibility of processors and the performance of ASICs (Figure 1.1). Reconfigurable devices are (mass produced) standard components, which have a fixed price per chip. They consist of an array of elementary hardware blocks connected by a network of programmable interconnections. All hardware blocks can work in parallel and by this offer a huge potential performance. Both the hardware blocks and the interconnections can be reconfigured. When done after power-up this is called *static reconfiguration*. Many systems use static reconfiguration for upgrading or debugging a system. This possibility is a large advantage compared to ASICs, which cannot be changed after production. *Dynamic reconfiguration* means switching between configurations at run-time depending

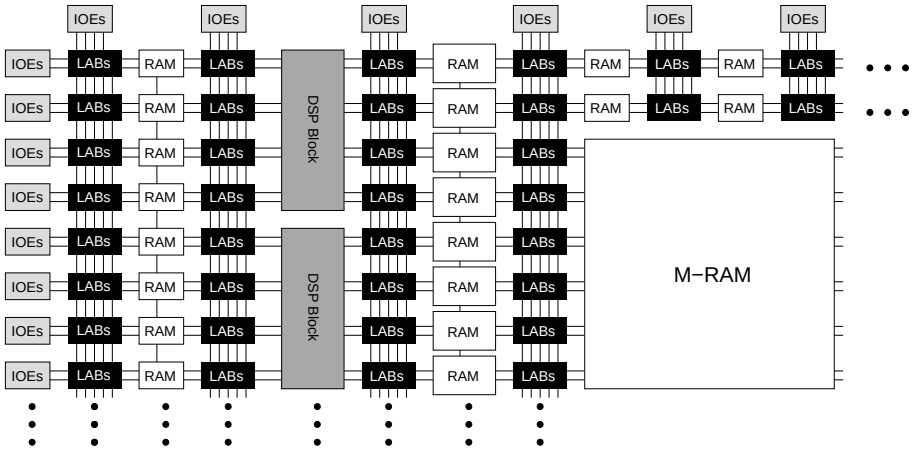


Figure 1.2: Example FPGA structure, inspired by the Altera Stratix device [13]. IOE: Input Output Element, LAB: Logic Array Block, DSP Block: Digital Signal Processing Block.

on the tasks that have to be executed at a certain point in time, or to time-multiplex the available hardware if the available size is not large enough to contain the entire design. When talking about dynamic reconfiguration one can distinguish between full and partial reconfiguration. In the latter case only part of the device is reconfigured .

1.1.2 Field programmable gate arrays

Field Programmable Gate Arrays (FPGAs) are the most popular example of reconfigurable devices. Figure 1.2 shows the architecture of a typical FPGA. The hardware blocks can be (re)configured to specify their function, e.g., the logic function of a Look-Up Table (LUT), the word and address size of memory blocks, the voltage levels of I/O pins. Together with the configuration of the interconnections this allows to implement any digital design (within size constraints) on a FPGA. The device can be reconfigured without limit. This shortens the production-testing-cycle, compared to an ASIC and allows to upgrade a system after production. However, a cost has to be paid for this flexibility. The reconfigurable interconnection network occupies a large part of the chip area and introduces long delays. As a result, the achievable clock frequency will be relatively low. The computational density [64] and performance are lower than on an ASIC but can still outperform those of processors [97, 36]. FPGAs can be more efficient

than instruction set architectures when the algorithm has a high degree of regularity and exploits high levels of parallelism.

FPGAs have evolved from very regular, fine-grain reconfigurable devices [58], which were mainly used to contain the glue logic or parts of a system, towards large heterogeneous devices with coarser blocks, such as block RAMs, embedded multipliers and even processors. They can contain an entire system on a chip (SoC) [1, 8]. Next to FPGAs, other reconfigurable architectures have been developed [145, 132, 103]. Many of them are composed of an instruction set processor and a reconfigurable array, which serves as an accelerator for custom instructions [125, 50].

A disadvantage of the fine-grain reconfigurability of FPGAs is the large area and delay cost of the interconnections, and the large reconfiguration time, typically some tens of milliseconds. Coarse-grain architectures restrict the reconfigurability, e.g., by fixing the word size of operations, and can therefore run at higher frequencies and have a lower reconfiguration time. This was also the reason for the introduction of coarser blocks inside FPGAs. Kuon and Rose [114] have measured the difference between FPGAs and ASICs in terms of logic density, circuit speed and power consumption for core logic, i.e. without considering I/O, on a set of benchmark designs. They found that for a 90-nm CMOS technology the ratio of silicon area required in FPGAs vs. ASICs is on average 35 when only using LUTs and flip-flops. When using hard blocks (multipliers and block memories) this average area gap drops to 18.

1.1.3 Design flow

Static configuration

In most of the systems, FPGAs are only configured at power-up, followed by a system reset (static reconfiguration). In this case the design trajectory is very similar to the higher levels of a VLSI design trajectory (Figure 1.3). FPGA vendors offer tools that can generate a configuration bitstream starting from Register Transfer Level (RTL) VHDL or Verilog. These tools take care of the technology mapping (synthesis) and placing and routing of the structural elements on the FPGA. After placing and routing, the number of hardware blocks used, the achievable clock frequencies, critical paths, ... are reported. The path from an algorithm description or system specification to synthesizable RTL code is less

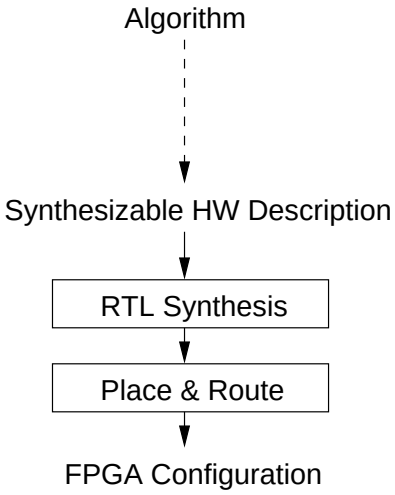


Figure 1.3: Simplified FPGA design flow.

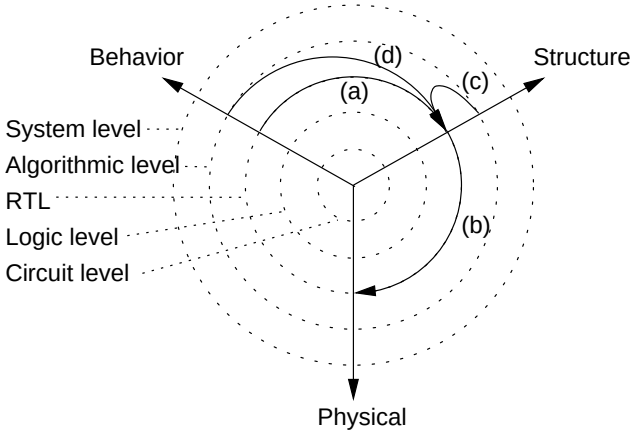


Figure 1.4: Gajski and Kuhn's Y-chart [87] positioning tool support for FPGA design. (a) RTL synthesis. (b) Place & Route. (c) System builder. (d) High-level synthesis.

automated and involves a lot of manual design effort. Not only has behavior to be specified and refined, but also structure has to be taken in mind. This differs a lot from writing software code and requires to make choices in a huge design space as presented by [63].

The design flow can be visualized on a Y-chart as introduced by Gajski and Kuhn in [87] (Figure 1.4). The concentric rings represent levels of abstraction, from *system level* at the outer to *circuit level* at the

inner. The three axes represent three different representations of a design: the *behavioral* or functional representation, the *structural* representation and the *physical* (geometrical) representation. The exact number of levels and their names may vary between variants of this chart, depending on the context.

Multiple tools, such as Altera SOPC builder [15] or Matlab Simulink allow to easily build a system by connecting modules from a library or user-made modules that meet a certain interface. The connections specified between modules at a high level are automatically refined to RTL (Figure 1.4(c)[1]). This alleviates the task of gluing a system together but leaves the task of designing the modules and thinking up the system architecture.

Recently, several high-level synthesis tools have emerged that claim to raise the synthesizable level from RTL to algorithmic level (Figure 1.4(d)) [153]. In reality, they offer a trade-off between abstraction level, i.e. design effort, and quality of the resulting hardware. In most cases, the coding style of an algorithm has a large influence on the synthesis results and the code has to be written with the target architecture in mind.

Dynamic reconfiguration

Dynamic (partial) reconfiguration asks for an alternative design flow [126, 59, 118] and will not be considered in this work. In the Gajski Y-chart, dynamic reconfiguration can be represented by adding a *reconfiguration level* below the algorithmic level [74], or by interpreting structure not only as structure in *space*, but also as structure in the combined *space-time*.

[1] In this figure the (non-standard) term *System Builder* is used. In [86] both (c) and (d) would be called high-level synthesis, the first in a bottom-up design methodology, the second in a top-down methodology. Nowadays, the widely used term high-level synthesis denotes the transfer from a high-level behavioral description to a low-level structural description. This convention is followed in this work.

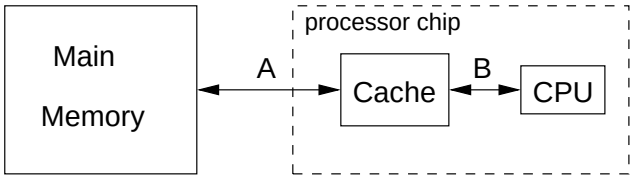


Figure 1.5: Simplified representation of the memory hierarchy in a processor-based system (Von Neumann architecture).

1.2 The need for loop transformations

1.2.1 The memory bottleneck

In the last decade multimedia applications have emerged on many places. Image and video processing are an example of applications that not only demand a high computational power, but also are data-intensive. They work on huge sets of data. Not all this data can be stored on a processor chip or FPGA. Therefore, an external memory has to be added (*Main Memory* in Figure 1.5 and 1.6).

The processing power of processors and FPGAs has increased exponentially during the last decades, not only by increasing the clock frequency but also by increasing the parallelism in the execution. The performance of memory has also increased exponentially, however with a smaller exponent. As a result, there is a widening *processor-memory gap*. The processing speed is often not limited by the processing of the data but rather by the time to fetch and store the needed data (a so-called *memory bottleneck*). The memory inside a processor core (register banks) or FPGA (registers, memory blocks) of course works at the same speed as the computational units, but the used technology, SRAM (Static RAM), is too expensive (area and power) to build the large main memory of a system.

To bridge this gap a memory hierarchy is built by inserting on-chip buffer memories between the processor and the main memory, e.g., one or multiple *caches*. Figure 1.5 shows the simple case of a single cache. The cache memory size is much larger than the register files in the processor but much smaller than the main memory. The data transfers on link B are slower (latency and bandwidth) than inside the CPU (Central Processing Unit), but much faster than on link A to the main memory. The memory blocks inside a FPGA can also be used to construct a memory hierarchy. In Figure 1.6 the general system architecture that will be

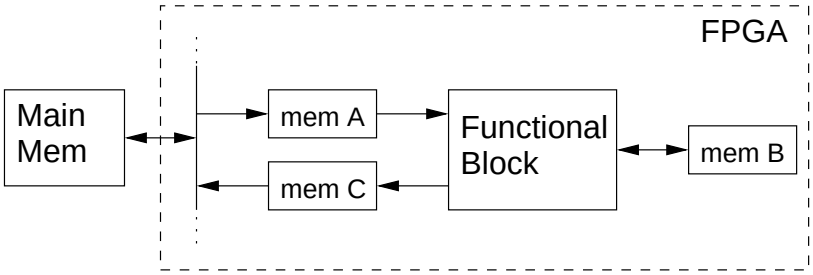


Figure 1.6: General (FPGA) memory hierarchy used within this work. Several of the memories might coincide or consist of multiple memories.

used in this work is defined. Each *functional block* has a memory to store intermediate results (mem B), and communicates with the main memory and other blocks through buffers (mem A and mem C). Some of these memories may consist of smaller memories that can be accessed in parallel or might be omitted or combined with other memories. The infrastructure that connects the blocks may be a bus, as suggested by the figure, but can also consist of, e.g., a switch fabric [15] or point-to-point connections.

To benefit from such a memory hierarchy the data in the buffers has to be reused as much as possible to decrease the accesses to the external memory. Therefore, the different accesses to a single data element should be close together in time, i.e. exhibit a good temporal locality, and consecutive accesses in time should access elements that are close to each other in the memory, i.e. exhibit a good spatial locality. The latter allows to transfer data to and from the external memory in bursts, which is more speed and power efficient.

A major point of difference between GPPs and FPGAs is that the memory hierarchy on a FPGA is made application specific and the data transfers are not controlled by a cache policy but are explicit, i.e. under full control of the designer, similar to scratch pad memories [23, 111]. It is the task of the designer to build a memory hierarchy that is appropriate for the application and avoid a memory bottleneck by ensuring that the data locality is sufficient.

Until now, high-level synthesis tools offer little support for the construction of memory hierarchies.

```
for (i=0;i<=N;i++){
  for (j=0;j<=N-i;j++){
    if (i==0)
      B[j]=1;                    // S1(i,j)
    B[i+j]=B[i+j]*A[i][j];       // S2(i,j)
  }
}
for (k=0;k<=N-1;k++){
  C[k]=B[k]+B[k+1];              // S3(k)
}
```

Figure 1.7: Small program example. The accesses to array B have a bad temporal locality.

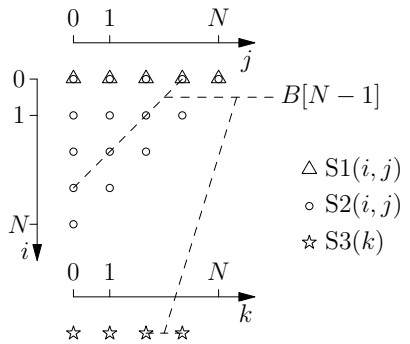


Figure 1.8: Polyhedral representation of the domains of the statements in Figure 1.7. The execution order is from left to right and from top to bottom. The accesses to the element $B[N-1]$, indicated by dashed lines, illustrate the bad temporal locality.

1.2.2 Loop transformations

As mentioned in the previous section, the data locality has a large influence on the performance of a system. Loop transformations can serve as a means to improve the data locality. They alter the execution order of computation steps, which can, e.g., bring accesses to the same data element closer together to improve the temporal locality. This will be illustrated by the following example. The definition of some terms used here is delayed until Chapter 3.

Consider the small loop nest containing three statements in Figure 1.7. A graphical (geometrical or polyhedral) representation of the iteration domains is shown in Figure 1.8. All the elements of array A and C are only accessed once. The elements of array B are accessed mul-

tiple times with a bad temporal locality, as illustrated for the element $B[N-1]$. New iterators i' and j' are defined by the following mapping[2] of statement invocations between the i, j, k space and the i', j' space

$$\begin{aligned} \mathrm{S1}(0, j) &\mapsto \mathrm{S1}(0, i') \\ \mathrm{S2}(i, j) &\mapsto \mathrm{S2}(j', i' - j') \\ \mathrm{S3}(k) &\mapsto \mathrm{S3}(i' - 1)\ . \end{aligned}$$

The mappings define the iteration domains in the i', j' space (by the condition that the same statement invocations occur). The execution order, now defined by (i', j'), is altered as shown in Figure 1.9 and 1.10. The temporal locality has improved. Within the innermost loop (iterator j') all statement invocations access the same element of array B. At each moment at most two elements of B contain data that is needed later on, i.e. are *alive*. As a result, the array can be reduced to two elements. This is done by the introduction of a modulo operation $(\bmod 2)$ in the index expressions.

This example shows how a loop transformation can improve the temporal locality of a program. Also the spatial locality and exploitable parallelism can be improved by loop transformations. Tools exist that apply user specified loop transformations on a program [29]. Finding loop transformations that optimize a certain property is much harder to automate.

Often, naively applying transformations, such as loop fusion, is prohibited since this would violate data dependences. In order to make them valid, a sequence of enabling transformations is needed. Typically, the data locality has to be made worse first, in order to make subsequent optimizations possible. Because of this, optimizing compilers often fail to automatically find the best sequence of transformations [90]. Iterative compilation has been proposed to automate the search for such sequences. The strategy is to iteratively compile a program with different sequences of optimizations and to evaluate the result by profiling the resulting binary. Based on the results, new sequences of transformations are explored. Alternatively, Cohen et al. [57] propose to use a processor simulator to analyze the main performance bottlenecks in a manual iterative process.

The process of optimizing the locality of a program by loop transformations can be split into three steps:

[2] How this mapping is found will be explained in Chapter 3.

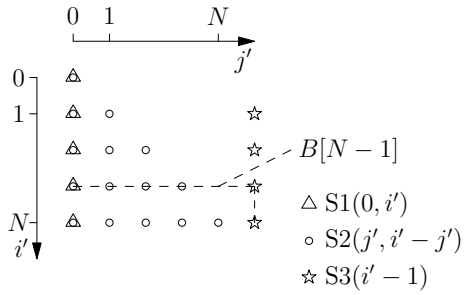


Figure 1.9: Domain after transformation. Execution order from left to right and from top to bottom. The locality of array B has improved. Maximum two elements are alive at the same time. The accesses to $B[i']$ and $B[i'-1]$ can be replaced with accesses to $B[i' \bmod 2]$ and $B[(i'-1) \bmod 2]$.

```
for (i'=0;i'<=N;i'++){
  B[i']=1;                      // S1(0,i')
  for (j'=0;j'<=i';j'++)
    B[i']=B[i']*A[j'][i'-j']; // S2(j',i'-j')
  if (i'>=1)
    C[i'-1]=B[i'-1]+B[i'];     // S3(i'-1)
}
```

Figure 1.10: Code after loop transformations, corresponding to Figure 1.9

- First, the parts of the code that cause bad locality have to be identified. This can be done by manual analysis (only feasible for small programs) or assisted by tools. The techniques can be divided into static code analysis and profiling of program runs.
- Once the *bad* code is identified, one has to seek loop transformations that improve the locality.
- The last step is the application of the loop transformations on the code. For this step multiple tools and/or environments have been developed, such as SUIF [6, 117, 100] and WRaP-IT/URUK [57, 90].

Iterating over these steps may be needed to get a satisfying result. Heuristics are needed, as the search space of possible transformations is too large to explore exhaustively. When designing hardware, some of the impact of a transformation might only become visible after descending to a synthesizable level. In this case iterating has a large cost, which augments the need for design automation.

To exploit the locality created by loop transformations further steps are needed, such as data reuse exploration, memory data layout and address optimizations [135, 131]. The DTSE methodology (Data Transfer and Storage Exploration) [52], developed at IMEC, offers a framework in which all these optimization steps have their place. Applying loop transformations is one of the first steps in this methodology.

1.3 Contributions of this work

Current high-level design environments offer little support to implement data-intensive applications on heterogeneous-memory systems; they rather focus on parallelism. This thesis addresses the memory hierarchy problem to high-level transformations of loop structures.

This work presents contributions to three different aspects in the flow from loop transformations to synthesizable hardware descriptions.

1.3.1 Loop transformations

Girbal illustrates in [90, 57] that optimizations directed by the human intelligence of a programmer can still outperform iterative compilation. Therefore, tools have been developed that automate the application of loop transformations but leave the decision of which transformations to apply to the programmer. These are specified as a sequence of elementary transformation steps.

In Chapter 3, we propose to raise the abstraction level from combining elementary transformation steps to combining application-specific *primary* (sub)sequences. By doing so, long transformation sequences can be constructed in a shorter time and more promising transformation variants can be generated. In this context, also the influence of the order in which transformation steps are applied is studied.

1.3.2 Bounds on quasi-polynomials over discrete domains

Bounds on (piecewise) quasi-polynomials over discrete domains are needed to statically evaluate properties, such as memory usage, of a program obtained after loop transformations. To this end, several novel methods to find such bounds are presented in Chapter 4 and compared

with existing methods. The presented methods take advantage of the fact that for large domains or small degrees the continuous-domain extrema of polynomials are a good approximation of the discrete domain extrema. For small domains the straightforward method of evaluating the (quasi-)polynomial in each point is still faster, but this solution does not scale for larger domains. We have proved that with a simple selection mechanism, which introduces almost no overhead, hybrid methods can be constructed that combine the strengths of different methods. With more complex selection mechanisms less additional benefit is expected.

1.3.3 Hardware generation

Since loop transformations not only influence the data access pattern but also the control complexity, integration of loop transformations and hardware generation is needed to speed up the design space exploration. Therefore, an architecture is presented in Chapter 5 with close correspondence to the polyhedral representation used to apply loop transformations. This architecture offers several options for area-speed trade-offs and supports parallelism and automation of VHDL code generation.

1.4 Structure of this thesis

The different chapters of this thesis are organized as follows. Chapter 2 gives an overview of related work on high-level synthesis and C-based hardware description languages, with special attention to the way memory is dealt with. Chapter 3 introduces the concepts of polyhedra and polytopes and how a polyhedral model can be used to apply loop transformations. Thereafter, the construction of long transformation sequences is discussed. In Chapter 4, the differences between the extrema of polynomials over discrete and continuous domains are studied. The results are used to construct methods to find bounds on polynomials and quasi-polynomials over discrete domains. The architecture that supports hardware generation starting from a polyhedral representation is presented in Chapter 5. Finally, conclusions and directions for future research are listed in Chapter 6.

Several appendices provide more information about implementation details, experimentation setup and results. For example, an

overview of the (inverse) discrete wavelet transform and several implementation aspects is found in Appendix A.

Chapter 2

High-level synthesis and C-based design

In recent years a lot of effort has been spent on raising the level of abstraction of hardware design. One approach is the development of tool suites that allow to build a system by selecting and connecting hardware blocks, custom made or drawn from a library of hardware cores. Another approach is the introduction of new design languages, typically C-based, that allow high-level synthesis or at least offer a smoother path from software to hardware or an appropriate means for HW/SW-codesign.
This chapter gives an overview of some high-level synthesis environments and investigates how they deal with memory related aspects.

2.1 Introduction

Everyone who has ever gone through the design process of a digital design, from high-level specification until synthesizable RTL code, understands the need for design automation. Manually refining high-level code involves decisions about architecture and timing. The impact of design choices is often only visible after low-level synthesis. This slows down the design iteration process and increases the development cost.

In recent years many high-level synthesis tools have emerged, each with their own application domain, input language, synthesis strategy, target architecture, ... They attempt to raise the abstraction level of hardware design, similar to the evolution in software from assembly to high-level languages.

Different approaches exist. Some tools offer little more than syntactic sugar, which requires hardware design knowledge of the user but gives much control over the implementation results. Other tools target software engineers with little hardware design knowledge and try to hide all implementation details from the developer. Some map applications on predefined architectures, while others need user directives to make architectural choices. Logically, there is a trade-off between the quality of synthesis results and the design effort and the class of applications that can be described.

This chapter gives an overview of several methods, with special attention to the way memory accesses are dealt with.

2.2 Overview of high-level synthesis environments

Several categories of high-level synthesis tools are distinguished depending on the target architecture and shape of the input description.

2.2.1 State-based description

Bluespec [2, 45] uses an operation-centric hardware abstraction [105], useful for describing systems with a high degree of concurrent (and thus complex) control. The behavior of a system is described as a collection of atomic operations on a set of state elements (terms). Each operation is specified as a set of simultaneous (hence atomic) state-element updates, and a predicate on the state values that captures the conditions under which it can occur. The effect of the state updates of an operation is atomic, i.e. the legal behaviors of the system constitute some sequential interleaving of the operations. One can describe each operation as if the rest of the system is frozen. However, the resulting implementation, generated by the Bluespec tools, carries out multiple operations per clock cycle if this does not conflict with the semantics of the atomic and sequential execution [104, 18]. This can, e.g., be used when a memory is shared by different computation blocks. Potential conflicts caused by concurrent accesses are made impossible by the synthesis methods.

The use of states and transitions to formally describe a system is very similar to Petri Nets [163]. That also Petri Nets can be used to describe a specification and refine, transform and translate it towards an implementation in hardware is demonstrated in [148].

These methods offer the designer a means to deal with the large complexity of concurrent control operations. However, they do not alleviate the task of developing an architectural structure and defining the *states* of a design. As such, these methods are good for control dominated systems but do not aid in dealing with data-intensive problems. They deal with parallelism, but data locality is not an issue at this low level of abstraction.

2.2.2 Process networks

Kahn Process Networks (KPN) [110] are a popular means to describe parallel systems. A process network consists of processes that communicate with each other via First-In-First-Out (FIFO) queues, called channels. The processes represent computation entities than can read data from the incoming channels and write data to the outgoing channels. A process is stalled if it tries to read from an empty channel or write to a full channel (*blocking* read and write). The size of the FIFO buffers should be large enough to avoid dead-lock [11]. Several variants and subsets of process networks have been defined, such as Communicating Sequential Processes (CSP), Synchronous Data Flow (SDF) and Cyclo-Static Data Flow [44]. An overview of such *Models of Computation* is given in [108] and [44].

The advantage of using process networks is that no global control exists. Control signals exist inside a node and FIFO channels are the only means of communication between nodes. As a result, all nodes can work in parallel and synthesis techniques can optimize each node independently.

Streaming-based applications perfectly fit this kind of networks. Such applications typically have a good spatial and temporal locality. However, if a process node does not process data in the order it is produced by another node, the FIFO buffers between those nodes have to be extended with reorder memories [157], which may result in a large memory cost.

The Compaan/Laura tool suite translates polyhedral loop nests into KPNs, by eliminating global memory and global control [156]. Reorder memories are automatically inserted where data is not consumed in the order it is produced [157]. Laura translates the KPNs into VHDL [171].

The User-Guided High-level synthesis tool (UGH) [21, 76], takes C code and a Draft Data Path (DDP) of nodes in a KPN as input. After

coarse and fine grain scheduling, a data path and Finite State Machine (FSM) is generated. A mapping step is (optionally) used to compute propagation delays, which are used to optimize the clock speed. No loop transformations nor memory optimizations are performed.

Impulse C [4] uses a subset of C extended with IO-macros to build a network of Communicating Sequential Processes (CSP). For each process that is selected to run in hardware, a data path and one large finite automaton is constructed. Communication with the other processes, which stay in software, is done through FIFOs. The states of a FSM relate directly to the control flow graph of the Impulse C code of the corresponding process. This code may be annotated with pragmas for pipelining or loop-unrolling. An experimental evaluation of Impulse C is found in Section 5.6.2.

To have satisfying results with such environments the application should have data access patterns that fit the model of producers and consumers linked by FIFOs. If not, reorder buffers may result in a large overhead or it may be possible that the final KPN consists of a single node, which eliminates the advantages of using such a model of computation.

Loop transformations may be needed to transform an application into a form that fits the KPN model. However, support for such transformations is not offered yet by the mentioned design environments.

2.2.3 Arrays of processing elements

Several projects aim at mapping loop nests on arrays of processing elements, often called systolic arrays, though they form an extension to the concept of systolic arrays in the strict sense. The hardware generation process consists of generating functional units, which can execute the loop bodies (described in a high-level language), placed in an array with controllers, memory and I/O. The iterations are mapped in both space (processing elements) and time.

In MMAlpha [93], loop nests are represented in a functional, dynamic single assignment language, (a system of recurrence equations). The code is entirely mapped onto a systolic array. To hide the bus or memory latency, the input to the array should be put in order in advance [65]. The output is done in the order samples are produced. In [93] Guillou et al. take care of the way on-chip memory blocks can be used for intermediate results. However, the case in which external

memories are needed is left as future work. How scheduling should be done to avoid large intermediate data sets is not addressed either.

The PICO project [140](Program In Chip Out) maps loop nests on systolic arrays which serve as *Non-Programmable Accelerators* (NPA) [147] for a specialized VLIW or EPIC processor (Explicitly Parallel Instruction Computing). The user may annotate an array with a pragma directing PICO to keep that array in a local memory. Register promotion is used for arrays with uniform dependences to eliminate load-/store operations. To limit the hardware cost for registers caused by this register promotion, the iteration space is tiled. This tiling reduces the iteration space that is implemented by the NPA (inner loops of the nest). The outer loops run on the host processor. The input that can be mapped on NPAs is limited to perfect loop nests.

PARO [101] constructs an array of identical processing elements and explores the possible space-time mappings [102]. A mapping specific interconnect topology [33] is built, and the control is not centralized but distributed over the elements [32]. In order to match resource constraints partitioning techniques are applied [155] to either sequentially execute tiles (local parallel, global sequential) or to sequentially execute operations within a tile (local sequential, global parallel) or to use an intermediate scheme. This is similar to the tiling done by PICO.

All these projects target systolic arrays, and thus are less appropriate for building non regular designs [94]. For this class of regular applications, it is typically not difficult to find schedules with a good locality.

2.2.4 C-based high-level synthesis

The most general input for high-level synthesis is a high-level software language, e.g., C. The following environments all have an input format that is based on a subset of C. In many cases special constructs are added to guide the synthesis process.

SPARK [98, 99] takes a behavioral description in ANSI-C as input and generates synthesizable VHDL. It performs compiler transformations, using Hierarchical Task Graphs (HTG), that overcome the effect of programming style on the quality of generated circuits. This includes speculative code motion transformations, dynamic renaming and heuristic scheduling. The methods work well to implement control-intensive functions of scalars, but it assumes parallel access to all its input data and is therefore not suitable to work with arrays or

memory accesses [166].

The Cameron Project has created a high-level algorithmic language, named Single Assignment C (SA-C) [46], for expressing image processing applications. It has special constructs for vector and sliding window operations (processing an input stream as a sequence of smaller overlapping blocks) and is functional in nature. Hierarchical Data Dependence and Control Flow (DDCF) graphs are used as an intermediate representation. Optimizations are implemented in the SA-C compiler, which may be controlled by user pragmas in the source code, e.g., to evaluate space-time trade-offs. Compilation to FPGAs is done by generating a structural VHDL description, using pre-existing parameterized library cores, e.g. corresponding to the vector and window operations. The problem of dealing with data locality is shifted to the programmer. He or she has to take care of it by using vector and window operations whenever possible. The compiler itself is not able to extract these constructs from a general program.

A similar concept, the use of hardware skeletons, is promoted by Benkrid [37], inspired by software skeletons, proved successful in parallel programming. This framework is based on a library of parameterized descriptions of task-specific architectures to which the user can supply parameters such as values, functions or other skeletons. Examples are again vector and window operations. Improving the memory behavior, e.g., by introducing line-buffers is done in a predefined way. Again, the user has to detect where input constructs with a pre-defined solution can be used.

ROCCC (Riverside Optimizing Configurable Computing Compiler), a successor of SA-C, also targets window operations, but is able to extract them from the input code, written in C with strong restrictions (no pointers, perfect loop nests with constant bounds, affine index functions, ...). The SUIF (Stanford University Intermediate Format) compiler [6] serves as front-end. Scalar replacement is used to exploit the data reuse within the loop nests [95] and after user directed loop unrolling [49] and other optimizations VHDL is generated [96]. The controllers and address generators are implemented using pre-existing parameterized FSMs from a VHDL library. For window operations a so-called *smart buffer* is generated, which stores live input data and exports data in parallel to the computation blocks. The instantiated smart buffer stores each data element in a fixed register. It does not shift data elements from one register to another. This leads to a com-

plex structure of multiplexers as criticized by Dong et al. in [75]. They demonstrate that the clock speed is increased by shifting the data elements instead of using multiplexers at the output of the smart buffer. Furthermore, it is not clear how ROCCC deals with code that does not fit the sliding-window model.

A similar compilation and synthesis system is DEFACTO [73]. It combines parallelizing compiler techniques with behavioral synthesis technology. Its strategy is to distribute the processed data over multiple memory banks according to the access pattern. An automatic design space exploration is performed to find the optimal unroll factor of the loop nests [151], where a balanced design, i.e. on the transition from *compute bound* to *memory bound*, is targeted. When the reuse distances are too large, loop tiling is applied to reduce the number of on-chip registers, similar to PICO and PARO. A large restriction is that only constant loop bounds, not even parameterized, are considered.

Handel-C [3] extends the C language to express hardware functionality, such as word lengths, explicit timing parameters and parallelism and limits the language to exclude C features, such as random pointers, that do not lend themselves to hardware translation. The Handel-C compiler can directly generate EDIF-netlists (Electronic Design Interchange Format) for a target FPGA instead of VHDL, which results in a higher performance (area and clock frequency). Due to the direct mapping from Handel-C constructs to hardware, the coding *style* has a large influence on the resulting hardware. Writing Handel-C code should be done with the resulting hardware in mind. An experimental evaluation of Handel-C is found in Section 5.6.2.

The C2H compiler of Altera generates coprocessors for the Nios II processor (soft core), that implement a C function [14]. Each element of the C syntax, including pointers, is translated to an equivalent hardware structure using straightforward mapping rules [10]. ASAP scheduling (As Soon As Possible) is used. The developer has to know the C-to-hardware mappings, to control the hardware structure of an accelerator, based on the structure of the C code [12]. Else a large area overhead may be introduced.

Since Handel-C and C2H have a straight-forward mapping from data structures to memories it is the full responsibility of the programmer to build memory structures, take care of the data access locality and have a good mapping of data to the memories.

2.3 Conclusions

Next to tools that increase a designer's productivity by offering methods to easily build a system by connecting IP-cores and other modules, a lot of high-level synthesis tools with corresponding languages have emerged. They offer a trade-off between manual design effort and the efficiency of a hardware implementation.

The high-level synthesis methods described in this chapter all focus more on exploiting parallelism than on improving data-access patterns. This may lead to bandwidth limited designs. The only loop transformations mentioned above are loop unrolling to increase the parallelism and loop tiling to reduce the number of on-chip memory used. It is the responsibility of the user of a high-level synthesis tool to study the memory access pattern and its locality and improve it if needed. For this optimization process many other loop transformations can be used. The next chapter deals with data locality and ways to improve it by using loop transformations.

Chapter 3

Loop transformations in the polyhedral model

The memory access pattern of a system typically has a large influence on its execution speed and energy dissipation. Therefore, loop transformations are needed to improve the spatial or temporal data locality. This chapter first describes how programs can be represented and transformed using a polyhedral representation. Then, the composition of sequences of transformations is lifted to a higher abstraction level. Application specific primary sequences, composed of elementary loop transformations, can be constructed and combined to more easily explore long transformation sequences.

3.1 The need for loop transformations

3.1.1 Data locality

The processor-memory gap

As mentioned in Section 1.2.1, the processing power of processors and the performance of memory chips both increase exponentially, but not with the same exponent. This causes a widening processor-memory gap. The large main memory is built with a slower technology than used for the small memories inside the CPU or FPGA. As a result, for many applications the performance is limited by the accesses to the main memory and not by the computational power.

Cache and scratch pad memory

To bridge this gap a memory hierarchy is built by inserting one or multiple buffer memories, e.g., *caches* and *scratch pad memories*, between the processor and the main memory. The buffer memories are much larger than the register files in the processor but much smaller than the main memory. The data transfers between processor and buffers are slower (latency and bandwidth) than inside the CPU (Central Processing Unit), but much faster than on the link to the main memory.

We will now have a closer look at an architecture composed of a CPU, a cache and a main memory (Figure 1.5). If the CPU needs data that is not present in the cache (a *cache miss*), that data will be copied from the main memory together with neighboring data elements, filling a *cache line*. A next access to an element of this cache line (a *cache hit*), does not need a transfer from the main memory and will be faster and more energy efficient.

When the cache is filled with data, fetching new data is only possible when other data is removed[1] from the cache. Which data is removed depends on the *cache policy*. This policy is a compromise between implementation cost and cache performance. The Least-Recently-Used policy (LRU), for example, removes the data elements with the longest time since their last access. The optimal policy is removing the elements that will not be used anymore or will be used the furthest away in the future [34]. This, however, requires knowledge about the future and is not possible when executing an arbitrary program.

For simplicity the concept of associativity, which limits the locations in the cache where data elements can be stored based on the address, is not discussed here. For a more detailed study of cache behavior the reader is referred to [40].

Data transfers that are grouped in bursts (transfers with consecutive addresses) are faster and consume less energy. In a cache-based system the burst length corresponds to a cache line. When the data access pattern of a program is known at compile time it may be beneficial to replace the cache with a *scratch pad memory* [23]. In this case, there is no cache replacement policy and all data transfers between the main memory and the scratch pad memory are controlled by the running

[1] Simply removed if the data has not been changed, written back to the main memory if it has.

program [109, 111]. For many applications less transfers and longer burst lengths can be obtained with application controlled scratch pad memories.

In FPGA-based systems buffer memories should be constructed using the on-chip memory blocks. In contrast with general purpose processors no fixed memory hierarchy exists. The memory hierarchy on dedicated hardware (FPGAs and ASICs) is custom-built and thus application specific. The designer has to take care of all memory transfers.

Spatial and temporal locality

To efficiently use the memory hierarchy, accesses to the main memory should be minimized by (re)using data that is present in the cache as much as possible. This can be expressed as having a good data locality. A qualitative definition of the concept of data locality has been given in Section 1.2.2. A high *temporal locality* means that different accesses to a same data element are close to each other in time, while a high *spatial locality* means that accesses that are close to each other in time access elements that are close to each other in the memory. The former depends on the execution order of memory accesses while the latter also depends on the way data elements are arranged (mapped) in the memory. A quantitative measure for temporal locality is given by the reuse distance, which will be defined in Section 3.2.

The execution order of memory accesses has a high impact on the temporal and spatial locality. Therefore, loop transformations, which alter this order, can be used to improve the locality. This has already been demonstrated with the small example in Section 1.2.2. The major subject of this chapter is the description of a polyhedral model that allows to automate the application of loop transformations and the presentation of techniques that guide the search for good loop transformations.

It should be noted that loop transformations are only one step in a series of steps to optimize the memory hierarchy of a system and its usage. A survey of data and storage optimization techniques is found in [135]. Examples of other steps are memory mapping, register allocation and address generation.

Example of a memory limited system

A manual design of an Inverse Discrete Wavelet Transform (IDWT) is described in Section A.3. According to the synthesis and simulation results the execution speed reaches almost 79 frames/s (design 26 and 27 in Table 5.9). However, when put into a real system with several modules (all clocked at 50 MHz), which share an external DDR-SDRAM memory, the frame rate drops down to 11.3 frames/s due to the time wasted when waiting for data. When other blocks also initiate memory transactions the frame rate is further reduced. With this IDWT implementation the RESUME video decoder (Section A.3.1) runs at only 5.14 frames/s, which is far from real-time. That loop transformations can be used to improve the locality will be shown in Section 3.7 and 3.8. That these transformations also increase the performance of a hardware implementation will be demonstrated in Chapter 5.

3.1.2 Parallelism

FPGAs can only outperform processors, which have a much higher clocking frequency, if enough operations can be done in parallel. Loop transformations can be used to raise the exploitable parallelism in an algorithm, typically in a trade-off with the locality. However, this is not the focus of this work and will not be discussed further.

3.2 Reuse distance

3.2.1 Definitions

A quantitative measure for temporal locality is given by the reuse distance. The following (software) terminology is used:

Definition 1. *A* memory reference *corresponds to a read or a write in the source code of a program, while a particular execution of that read or write at run-time is called a* memory access. *A* reuse pair *is a pair of accesses to the same data location without intervening accesses to that location. The* use *of a reuse pair is the first access, the* reuse *is the second access. The* reuse distance *of a reuse pair is the number of unique data locations accessed between use and reuse.*

Definition 2. *The* backward reuse distance (BRD) *of a memory access is the reuse distance of the reuse pair in which this access is the second access.*

```
for (i=0;i<=N;i++){
  for (j=0;j<=N-i;j++){
    B[i+j]=B[i+j]*A[i][j];    // S2(i,j)
  }
}
for (k=0;k<=N-1;k++){
  C[k]=B[k]+B[k+1];           // S3(k)
}
```

Figure 3.1: Simplified version of the code in Figure 1.7.

If the element is accessed for the first time no such pair exists and the BRD is defined as ∞.

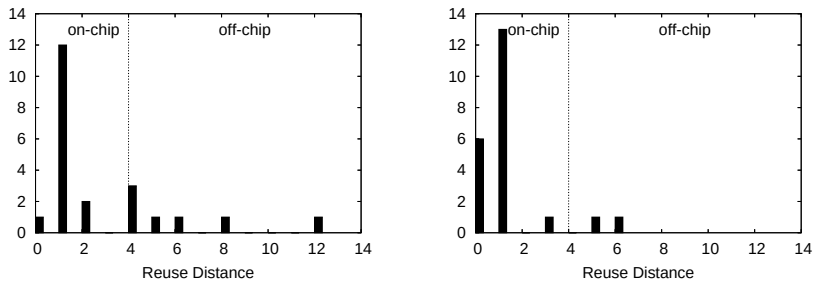
In a (fully associative[2]) cache with the Least Recently Used (LRU) replacement policy, data is retained between use and reuse if and only if the corresponding reuse distance is smaller than the cache or buffer size [42].

The concepts introduced above are illustrated on the code in Figure 3.1. This corresponds to the code in Figure 1.7, but simplified by removing the initialization statement `S1`. Table 3.1 lists the memory accesses for a program run with $N = 3$. It is assumed that only the array references generate memory accesses. A (fully associative) cache with LRU replacement policy and size four is used. The elements in the cache are sorted depending on the time since the last access. If the BRD is smaller than four, i.e. the cache size, this number gives the position in the cache before the access, counting from 0 onwards. For example, the second access to $B[0]$ in iteration $(i, j) = (0, 0)$ has a BRD of 1, which corresponds to the position of $B[0]$ after the access to $A[0][0]$. A histogram of the reuse distances is shown in Figure 3.3(a). A vertical line indicates the cache size and separates the accesses which result in a cache hit (left side) from those which result in a cache miss (right side). After the transformations described in Section 1.2.2 (and removing `S1`)

[2]This means a main memory word can be stored in any location in the cache without restrictions. In a cache with an associativity equal to k, each main memory word can be stored in one of k places (defined by the address) in the cache. The special case when $k = 1$ is called a direct mapped cache. Direct mapped and set-associative caches give rise to *conflict misses*. This means data has been removed from the cache although other data in the cache is unused for a longer time, caused by the restrictions on the location in the cache where a word can be stored. In this work we do not consider the concept of *cache associativity*.

```
for (i'=0;i'<=N;i'++){
  for (j'=0;j'<=i';j'++)
    B[i']=B[i']*A[j'][i'-j']; // S2(j',i'-j')
  if (i'>=1)
    C[i'-1]=B[i'-1]+B[i'];    // S3(i'-1)
}
```

Figure 3.2: Code after loop transformations, corresponding to Figure 1.10 without `S1`.



(a) (b)

Figure 3.3: Histogram of the reuse distances of the example in Section 3.2.1 before (a) and after (b) loop transformations, corresponding to the code in Figure 3.1 and 3.2, respectively. $N = 3$.

(Figure 3.2) a lot of accesses have a smaller reuse distance as illustrated by the histogram in Figure 3.3(b).

As demonstrated above, the reuse distance histogram allows to calculate the number of data transfers to/from the main memory in a cache-based memory hierarchy. Also in other environments the reuse distance histogram can be used as a measure for the temporal locality. It then serves as an indication of the potential to avoid data transfers to an external memory by using on-chip memory.

3.2.2 Reuse distance analysis to guide loop transformations

The reuse histogram shows whether a program has a good or a bad locality. However, more information can be extracted from reuse pairs by also looking at the location of use and reuse in the program. This way the accesses that cause poor locality can be localized.

Three kinds of reuse pairs can occur in a program with nested loops:

Table 3.1: Memory accesses of the execution of the code in Figure 3.1 for $N = 3$, with corresponding backward reuse distance (BRD) and cache content (cache with size 4 and LRU replacement policy). The cache content is sorted according to the BRD.

Iteration		Access	BRD	Cache Content (after access)			
				0	1	2	3
$(i,j) =$	$(0,0)$	$B[0]$	∞	$B[0]$			
		$A[0][0]$	∞	$A[0][0]$	$B[0]$		
		$B[0]$	1	$B[0]$	$A[0][0]$		
	$(0,1)$	$B[1]$	∞	$B[1]$	$B[0]$	$A[0][0]$	
		$A[0][1]$	∞	$A[0][1]$	$B[1]$	$B[0]$	$A[0][0]$
		$B[1]$	1	$B[1]$	$A[0][1]$	$B[0]$	$A[0][0]$
	$(0,2)$	$B[2]$	∞	$B[2]$	$B[1]$	$A[0][1]$	$B[0]$
		$A[0][2]$	∞	$A[0][2]$	$B[2]$	$B[1]$	$A[0][1]$
		$B[2]$	1	$B[2]$	$A[0][2]$	$B[1]$	$A[0][1]$
	$(0,3)$	$B[3]$	∞	$B[3]$	$B[2]$	$A[0][2]$	$B[1]$
		$A[0][3]$	∞	$A[0][3]$	$B[3]$	$B[2]$	$A[0][2]$
		$B[3]$	1	$B[3]$	$A[0][3]$	$B[2]$	$A[0][2]$
	$(1,0)$	$B[1]$	4	$B[1]$	$B[3]$	$A[0][3]$	$B[2]$
		$A[1][0]$	∞	$A[1][0]$	$B[1]$	$B[3]$	$A[0][3]$
		$B[1]$	1	$B[1]$	$A[1][0]$	$B[3]$	$A[0][3]$
	$(1,1)$	$B[2]$	4	$B[2]$	$B[1]$	$A[1][0]$	$B[3]$
		$A[1][1]$	∞	$A[1][1]$	$B[2]$	$B[1]$	$A[1][0]$
		$B[2]$	1	$B[2]$	$A[1][1]$	$B[1]$	$A[1][0]$
	$(1,2)$	$B[3]$	4	$B[3]$	$B[2]$	$A[1][1]$	$B[1]$
		$A[1][2]$	∞	$A[1][2]$	$B[3]$	$B[2]$	$A[1][1]$
		$B[3]$	1	$B[3]$	$A[1][2]$	$B[2]$	$A[1][1]$
	$(2,0)$	$B[2]$	2	$B[2]$	$B[3]$	$A[1][2]$	$A[1][1]$
		$A[2][0]$	∞	$A[2][0]$	$B[2]$	$B[3]$	$A[1][2]$
		$B[2]$	1	$B[2]$	$A[2][0]$	$B[3]$	$A[1][2]$
	$(2,1)$	$B[3]$	2	$B[3]$	$B[2]$	$A[2][0]$	$A[1][2]$
		$A[2][1]$	∞	$A[2][1]$	$B[3]$	$B[2]$	$A[2][0]$
		$B[3]$	1	$B[3]$	$A[2][1]$	$B[2]$	$A[2][0]$
	$(3,0)$	$B[3]$	0	$B[3]$	$A[2][1]$	$B[2]$	$A[2][0]$
		$A[3][0]$	∞	$A[3][0]$	$B[3]$	$A[2][1]$	$B[2]$
		$B[3]$	1	$B[3]$	$A[3][0]$	$A[2][1]$	$B[2]$
$(k) =$	(0)	$B[0]$	12	$B[0]$	$B[3]$	$A[3][0]$	$A[2][1]$
		$B[1]$	8	$B[1]$	$B[0]$	$B[3]$	$A[3][0]$
		$C[0]$	∞	$C[0]$	$B[1]$	$B[0]$	$B[3]$
	(1)	$B[1]$	1	$B[1]$	$C[0]$	$B[0]$	$B[3]$
		$B[2]$	6	$B[2]$	$B[1]$	$C[0]$	$B[0]$
		$C[1]$	∞	$C[1]$	$B[2]$	$B[1]$	$C[0]$
	(2)	$B[2]$	1	$B[2]$	$C[1]$	$B[1]$	$C[0]$
		$B[3]$	5	$B[3]$	$B[2]$	$C[1]$	$B[1]$
		$C[2]$	∞	$C[2]$	$B[3]$	$B[2]$	$C[1]$

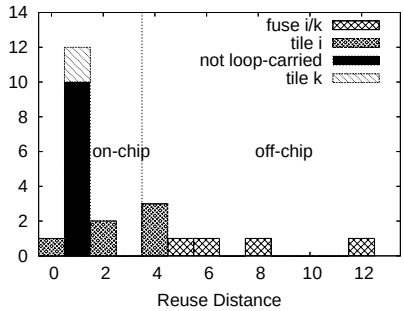


Figure 3.4: Reuse distance histogram before transformations (Equivalent to Figure 3.3(a)). The textures indicate loop transformations that may improve the locality of the corresponding class of reuses.

- *The use and reuse occur across different iterations of a single loop.* This pattern arises when the loop traverses a "data structure" (as small as a single scalar variable or as large as all the data in the program) in every iteration of the loop. The distance of reuses across iterations can be reduced by ensuring that only a small part of the data structure is traversed in any given iteration. As such, reuses of data elements between consecutive iterations are separated by only a small amount of data, instead of the complete data structure. A large number of transformations have been proposed that all aim at increasing temporal locality in this way, such as loop tiling [164], loop interchange [124] and loop chunking [30]. These transformations will be called *tiling-like* optimizations.

- *The use and reuse occur between different loops.* A data structure is traversed by the first loop, after which it is retraversed by the second loop. The reuses can be brought closer together by only doing a single traversal, performing computations from both loops at the same time. This kind of optimization is known as loop fusion. The required transformation will be called a *fusion-like* optimization.

- *The use and reuse occur inside one iteration of a loop.* The use and reuse occur at a close distance, inside the same basic block. If the reuse distance needs to be made shorter, some simple reordering of code in that basic block needs to be performed. This is a *non-loop-carried reuse*.

Techniques to classify the reuse pairs in a program run and to pro-

duce the corresponding code refactoring (loop transformation) hints are described in [43, 41, 66] and implemented in the SLO tool (Suggestions for Locality Optimizations) [39].

Figure 3.4 shows the same reuse histogram as Figure 3.3(a), but with the classification of the reuse pairs. It indicates that fusing loop i with loop k is required to optimize the reuse pairs with the longest reuse distance. Next, a tiling-like transformation on loop i can reduce the distance of reuses carried by loop i. This histogram only is an indication of the origin of reuse pairs. It does not guarantee that the suggested transformations are possible and, if they are, will not worsen the reuse distance of other pairs. With these techniques a priority ranking can be given to different potential optimizations. For example, optimizing the reuses carried by loop k is useless as long as the other, longer reuses are not optimized.

3.3 Vectors, polyhedra and polytopes

Until now loop transformations have been used in some *magical* way to improve the data locality. Before loop transformations and, more particularly, their mathematical representation are studied in more detail, some concepts have to be introduced. The definitions and notations are based on [55, 119, 40, 159]. A more general study on polytopes is found in [170]. Some definitions will not be used until Chapter 4, but are put in this section to clarify the interrelationships.

3.3.1 Vectors

To simplify the notations, implicit usage will be made of the isomorphism between the set of n-dimensional vectors and the set of column matrices with n rows over a given field, e.g., $\mathbb{Q}^n$ and $\mathbb{Q}^{n\times 1}$. This allows to multiply matrices with vectors and write n-tuples as column matrices. No distinction will be made between expressions as (i, j) and $[i\ j]^{\mathrm{T}}$ or $\begin{bmatrix} i \\ j \end{bmatrix}$.

Partial ordering vectors is defined element-wise:

$$(a_0, a_1, \ldots, a_k) \geq (b_0, b_1, \ldots, b_k) \Leftrightarrow a_i \geq b_i, \forall i \in \{0, 1, \ldots, k\} \ . \quad (3.1)$$

Definition 3. *Vectors can be totally ordered using the* lexicographical or-

dering '$\prec$':

$$\begin{gathered}(a_0, a_1, \ldots) \prec (b_0, b_1, \ldots) \\ \Updownarrow \\ \exists k \in \mathbb{N} \mid a_k < b_k \wedge \forall i < k, a_i = b_i \ .\end{gathered} \tag{3.2}$$

The definition for '$\preceq$' follows from this definition:

$$a \preceq b \Leftrightarrow a \prec b \vee a = b \ .$$

Note that the two vectors do not need to have the same length to be compared. They do of course to be equal.

Some examples are $(0, 3) \prec (1, 2)$ and $(1, 0) \prec (1, 1, 0)$. With this definition the ordering of, e.g., $(0, 1)$ and $(0, 1, 2)$ is undefined. Such cases will be prevented in this work, though the definition could be extended to include them. As such, the lexicographical ordering results in a total ordering.

3.3.2 Non-parameterized polyhedra and polytopes

An equality can always be written as two conjugate inequalities:

$$ax = bp + c \Leftrightarrow ax \leq bp + c \wedge ax \geq bp + c \ .$$

As a result, a system of equalities and inequalities can always be written as a system with inequalities only.

Definition 4. *A* convex rational polyhedron *P is defined by a finite set of linear inequalities:*

$$P = \{x \in \mathbb{Q}^n \mid Ax \geq c\} \ , \tag{3.3}$$

where A is a constant integer matrix and c is a constant integer vector. This corresponds to the intersection of a finite set of closed linear half-spaces (each bounded by a hyperplane). Formula 3.3 is called the implicit representation *of a polyhedron. A bounded polyhedron is called a* polytope*. A k-dimensional polyhedron/polytope is called a k-polyhedron/polytope.*

Note that the dimension k of a polytope can be smaller than the dimension n of the space it is defined in. For example, the triangle with vertices $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ is a 2-dimensional polytope embedded in a 3-dimensional space.

Some sources define a rational polyhedron as a subset of $\mathbb{R}^n$ instead of $\mathbb{Q}^n$. The term *rational* only points to the fact that A and c are rational (integer after multiplication with the least common multiple of the denominators of the elements of both).

Recall that the line segment between the points $A(x_1, y_1)$ and $B(x_2, y_2)$ can be written as

$$AB = \left\{ (x, y) \in \mathbb{Q}^2 \mid \begin{bmatrix} x \\ y \end{bmatrix} = \nu \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + (1 - \nu) \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}, \ \nu \in [0, 1] \right\} \tag{3.4}$$

or

$$AB = \left\{ (x, y) \in \mathbb{Q}^2 \mid \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} \begin{bmatrix} \nu_1 \\ \nu_2 \end{bmatrix}, \ \nu_i \geq 0, \ \sum_i \nu_i = 1 \right\} \ , \tag{3.5}$$

which is more intuitive than, but equivalent to using a system of inequalities as in (3.3) :

$$\begin{bmatrix} y_2 - y_1 & x_1 - x_2 \\ y_1 - y_2 & x_2 - x_1 \\ 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_2 y_1 - x_1 y_2 \\ x_1 y_2 - x_2 y_1 \\ -\min(x_1, x_2) \\ \max(x_1, x_2) \\ -\min(y_1, y_2) \\ \max(y_1, y_2) \end{bmatrix} \geq 0 \ .$$

Polyhedra can be defined in a way similar to (3.5).

Definition 5. *A polyhedron can also be represented using its* Minkowski (or geometric) representation*:*

$$P = \left\{ x \in \mathbb{Q}^n \mid x = L\lambda + R\mu + V\nu, \ \mu \geq 0, \ \nu \geq 0, \ \sum_i \nu_i = 1 \right\} \ . \tag{3.6}$$

The columns of matrices L, R and V contain the lines, rays *and* vertices *of P, respectively.*

A polytope is represented by the term $V\nu$ only, since the terms $L\lambda$ and $R\mu$ are not bounded. The term $L\lambda$ with unconstrained λ can be omitted by replacing $L\lambda + R\mu$ by $R'\mu'$ with $R' = [L \ -L \ R]$ and $\mu' \geq 0$, as demonstrated below. One can write

$$L\lambda = L\zeta - L\xi \ ,$$

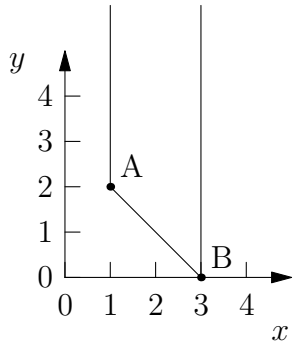


Figure 3.5: Graphical representation of the polyhedron in example 1.

with

$$\zeta_i = \begin{cases} 0 & \text{if } \lambda_i < 0 \\ \lambda_i & \text{if } \lambda_i \geq 0 \end{cases} \quad , \quad \xi_i = \begin{cases} -\lambda_i & \text{if } \lambda_i < 0 \\ 0 & \text{if } \lambda_i \geq 0 \end{cases} \quad ,$$

and thus

$$L\lambda + R\mu = [L \ -L \ R] \begin{bmatrix} \zeta \\ \xi \\ \mu \end{bmatrix} = R'\mu' \quad ,$$

with $\zeta \geq 0$, $\xi \geq 0$ and thus $\mu' \geq 0$.

Example 1. *Figure 3.5 shows the polyhedron P defined as*

$$P = \left\{ (x, y) \in \mathbb{Q}^2 \mid \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \geq \begin{bmatrix} 1 \\ -3 \\ 3 \end{bmatrix} \right\} \quad .$$

This can be written in a more compact way as

$$P = \left\{ (x, y) \in \mathbb{Q}^2 \mid 1 \leq x \leq 3, \ x + y \geq 3 \right\} \quad .$$

The Minkowski representation is given by

$$P \quad = \quad \left\{ \begin{array}{r} (x, y) \in \mathbb{Q}^2 \mid \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \mu + \begin{bmatrix} 1 & 3 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} \nu_0 \\ \nu_1 \end{bmatrix}, \\ \mu \geq 0, \ \nu_i \geq 0, \ \sum_i \nu_i = 1 \end{array} \right\} \quad ,$$

which indicates the presence of one ray (0,1) and 2 vertices A (1,2) and B (3,0).

Definition 6. *F is a* face *of a polyhedron P if and only if F is a non-empty subset of P and*

$$F = \{x \in P \mid A'x = b'\} \quad , \tag{3.7}$$

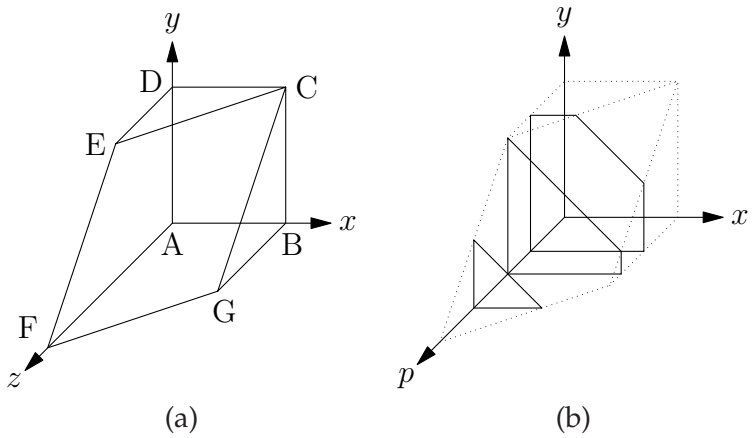


Figure 3.6: The 3-polytope of example 2 (a) corresponds to the representation of the parameterized 2-polytope of example 3 in the combined data/parameter space (b). Intersections are shown for p equal to 3, 5 and 8.

for some subsystem $A'x \geq b'$ of $Ax \geq b$. Each face is a polyhedron itself, and is called a k-face if it is a k-polyhedron. The 0-dimensional faces are the vertices of the polytope.

Example 2. *The 3-polytope in Figure 3.6(a) is defined as*

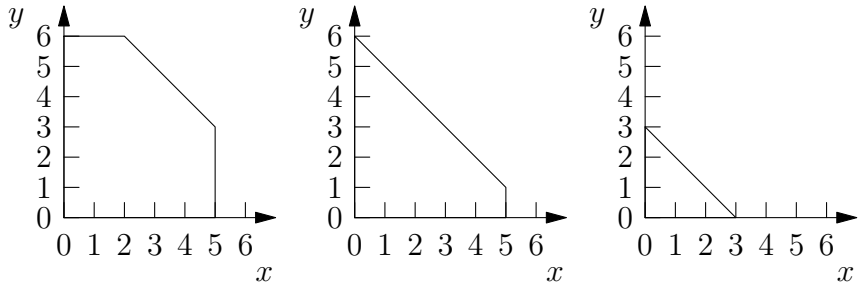
$$P = \{(x, y, z) \in \mathbb{Q}^3 \mid 0 \leq x \leq 5,\ 0 \leq y \leq 6,\ 0 \leq z \leq 11 - x - y\} \ ,$$

or

$$P = \left\{ V\nu,\ \nu_i \geq 0,\ \sum_i \nu_i = 1 \right\} \ , \ V = \begin{bmatrix} 0 & 5 & 5 & 0 & 0 & 0 & 5 \\ 0 & 0 & 6 & 6 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 11 & 6 \end{bmatrix} \ .$$

Its vertices (0-faces, columns of V) are A (0,0,0), B (5,0,0), C (5,6,0), D (0,6,0), E (0,6,5), F(0,0,11) and G (5,0,6). The other faces are the line segments (1-faces) AB, BC, CD, DA, DE, EC, EF, FG, FA, GC, GB, the triangles and quadrangles (2-faces) ABCD, CDE, ADEF, ABGF, CBG, CEFG, the polytope P itself (3-face) and the empty set.

The 2-faces are found by adding one equality to the system of inequalities, e.g., adding $x = 0$ leads to the face ADEF, adding $x = 5$ to CBG. These are the intersections of the polyhedron with planes. The 1-faces are found by adding two equalities, e.g., $x = 0 \wedge y = 0$ leads to AF, $x = 5 \wedge y = 0$ to BG. To find the 0-faces, three equalities are needed, e.g., $x = 0 \wedge y = 0 \wedge z = 11 - x - y$, which determines F. The empty set is also a face, e.g., by using $x = 0 \wedge x = 5$.



(a) $p = 3$ (b) $p = 5$ (c) $p = 8$

Figure 3.7: Parameterized polytope of example 3 for three parameter values.

3.3.3 Parameterized polyhedra and polytopes

Definition 7. *A* rational parameterized polyhedron P_p *is a family of polyhedra, defined as*

$$P_p = P(p) = \{x \in \mathbb{Q}^n \mid Ax \geq Bp + c\} \qquad \textit{for } p \in \mathbb{Q}^m \ , \tag{3.8}$$

where A and B are constant integer matrices, c is a constant integer vector, and p is a vector of parameters.

Example 3. *The parameterized polytope*

$$P_p = \left\{(x, y) \in \mathbb{Q}^2 \mid 0 \leq x \leq 5,\ 0 \leq y \leq 6,\ 0 \leq p,\ x + y \leq 11 - p\right\} \ ,$$

is shown in Figure 3.7 for 3 values of p. Figure 3.6(b) represents these polytopes as intersections of a polytope in the combined data/parameter space (corresponding to the polytope of example 2 with $z = p$) and planes with constant parameter values p.

As indicated by example 3, each parameterized polyhedron can be rewritten as a non-parameterized polyhedron in the combined data/parameter space:

$$P' = \left\{ \begin{bmatrix} x \\ p \end{bmatrix} \in \mathbb{Q}^{n+m} \mid A' \begin{bmatrix} x \\ p \end{bmatrix} \geq c \right\} \ , \tag{3.9}$$

where $A' = [A \quad -B]$.

Table 3.2: Parametric vertices of P_p (example 3) with corresponding 1-faces (line segments) of the polyhedron P' in the combined data/parameter space (example 2).

Line segment (1-face) P'	Vertex P_p	Domain
DE	$(0, 6)$	$0 \leq p \leq 5$
CE	$(5 - p, 6)$	$0 \leq p \leq 5$
CG	$(5, 6 - p)$	$0 \leq p \leq 6$
BG	$(5, 0)$	$0 \leq p \leq 6$
EF	$(0, 11 - p)$	$5 \leq p \leq 11$
GF	$(11 - p, 0)$	$6 \leq p \leq 11$
AF	$(0, 0)$	$0 \leq p \leq 11$

Let $\mathcal{S}(\hat{p})$ be the subspace of $\mathbb{Q}^{n+m}$ with constant parameter value $p = \hat{p}$ and proj_d the projection on the data space, i.e.

$$\mathcal{S}(\hat{p}) = \left\{ \begin{bmatrix} x \\ p \end{bmatrix} \in \mathbb{Q}^{n+m} \mid p = \hat{p} \right\} , \tag{3.10}$$

$$\mathrm{proj}_d : \mathbb{Q}^{n+m} \to \mathbb{Q}^n, \begin{bmatrix} x \\ p \end{bmatrix} \mapsto x . \tag{3.11}$$

Then,

$$P_p = \mathrm{proj}_d(P' \cap \mathcal{S}(p)) . \tag{3.12}$$

Theorem 1. *The (parametric) vertices of a parameterized polyhedron P_p correspond to projections on the data space of intersections of $\mathcal{S}(p)$ with the m-faces of P', its representation in the combined data/parameter space (Equation 3.9), where m is the number of parameters [119].*

Table 3.2 shows how the vertices of the 2-polytope of example 3 correspond to the projections of intersections with line segments (1-faces) of the 3-polytope in example 2. Each vertex does only exist within a certain parameter domain (the projection of the m-face on the parameter space). The parametric vertices can be represented as affine functions of the parameter(s). Some 1-faces, e.g., AB, do not have a corresponding parametric vertex, as the projection of its intersection with $\mathcal{S}(p)$ is not 0-dimensional.

From here on, the parameter values are restricted to integers: $p \in \mathbb{Z}^m$. The set of integer points in a rational polyhedron is sometimes called an *integer polyhedron* [40]. However, more often this

name (and also *lattice polyhedron* [24]) is reserved for a rational polyhedron of which all the vertices have integer coordinates. Therefore, the name $\mathbb{Z}$-polyhedron/polytope is used to unambiguously denote the intersection of a rational polyhedron/polytope with an integer lattice (here always $\mathbb{Z}^n$), which corresponds to the set of integer points within a rational polyhedron/polytope.

3.4 Program representation in the polyhedral model

3.4.1 Statement and iteration domain

The concept of *statement* is known in all procedural programming languages and will not be defined here. It should be noted that a statement is defined recursively, e.g., a compound statement is a statement composed of a sequence of statements and a selection statement contains statements preceded by a condition. Even a loop can be considered as one statement.

Definition 8. *The (lexical)* depth *of a loop or a statement is the number of loops that surround it. A statement* `S` *is executed for a set of values of its* iteration vector, $\mathcal{I}_{\mathtt{S}}$, *the vector grouping the iterators of the surrounding loops. This vector has a dimension equal to the depth,* $D_{\mathtt{S}}$, *of the statement. A single execution of a statement is called a* statement invocation. *The* iteration domain $\mathcal{D}_{\mathtt{S}}$ *is the set of values of the iteration vector for which the statement* `S` *is executed.*

In this work only parts of a program with static control, *Static Control Part* or *SCoP* [57], are considered. These consist of (non-perfectly) nested loops with constant strides and loop bounds that are linear (affine) expressions of the iterators of the surrounding loops and some (global) parameters. Conditional execution should be statically predictable and depend on Boolean expressions of affine inequalities. This means that the iteration domains are not data dependent, but can be determined at compile time as a function of the program parameters. An iteration domain will be a parameterized $\mathbb{Z}$-polytope or a union of parameterized $\mathbb{Z}$-polytopes.

Example 4. *The code in Figure 1.7 contains three statements* `S1`, `S2` *and* `S3`. *However, the inner of the first loop nest can also be considered as a single com-*

```
#define S12(i,j)            \
    if (i==0)               \
      B[j]=1;               \
    B[i+j]=B[i+j]*A[i][j];

#define S3(k)               \
  C[k]=B[k]+B[k+1];

for (i=0;i<=N;i++){
  for (j=0;j<=N-i;j++){
    S12(i,j);
  }
}
for (k=0;k<=N-1;k++){
  S3(k);
}
```

Figure 3.8: Separation of statement definitions and loop control. The code is equivalent to the code in Figure 1.7, but the choice to define two instead of three statements was made.

pound statement, `S12`*, as shown in Figure 3.8. In both figures the statements are printed as functions of their iteration vectors,* (i, j) *and* (k)*, respectively.*

The iteration domains are given by:

$$\mathcal{D}_{\texttt{S1}} = \{(i,j) \in \mathbb{Z}^2 | i = 0 \wedge 0 \leq j \leq N\} \tag{3.13}$$

$$\mathcal{D}_{\texttt{S2}} = \mathcal{D}_{\texttt{S12}} = \{(i,j) \in \mathbb{Z}^2 | 0 \leq i \leq N \wedge 0 \leq j \leq N - i\} \tag{3.14}$$

$$\mathcal{D}_{\texttt{S3}} = \{(k) \in \mathbb{Z}^1 | 0 \leq k \leq N - 1\}. \tag{3.15}$$

3.4.2 Schedule vector

Definition 9. *A program's top-level (depth 0) is a sequence of loops and statements which can be numbered from 0 onwards (Figure 3.9). Each loop in turn contains a sequence of statements and loops that are also numbered from 0 onwards. Each statement is uniquely identified by the sequence of numbers indicating the position in each of the surrounding loops and the top level. This sequence is called the* ordering vector.[3] *It has dimension* $D_{\mathrm{S}} + 1$ *with* D_{S} *the depth of statement* S*. The ordering vector can easily be constructed from the Abstract Syntax Tree (AST) (Figure 3.10).*

[3]Note that in this context *ordering vector* is completely different from the concept mentioned in [159, Section 3.5].

Depth 0 1 2	Ordering vector	Iteration vector	Schedule vector
0: `for (i=0;i<=N;i++){`			
0: `for (j=0;j<=N-i;j++){`			
0: `if (i==0) S1(i,j);`	$(0,0,0)$	(i,j)	$(0,i,0,j,0)$
1: `S2(i,j);`	$(0,0,1)$	(i,j)	$(0,i,0,j,1)$
`}`			
`}`			
1: `for (k=0;k<=N-1;k++){`			
0: `S3(k);`	$(1,0)$	(k)	$(1,k,0)$
`}`			

Figure 3.9: Program example with indication of statement depth, ordering vector and schedule vector.

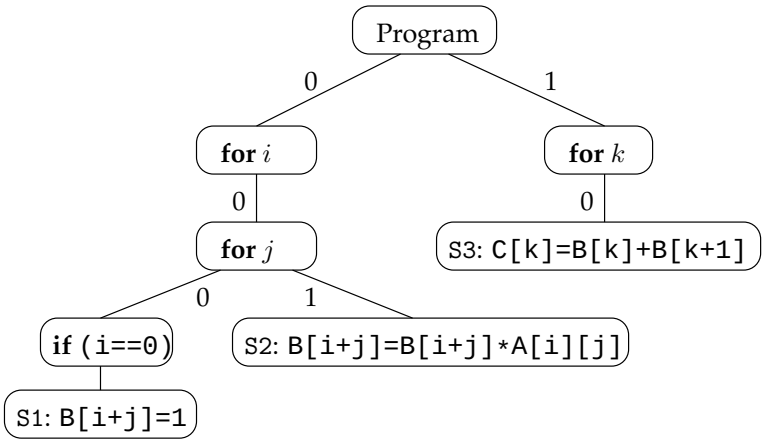


Figure 3.10: Simplified abstract syntax tree (AST) of the code in Figure 3.9. The ordering vector of a statement is formed by the labels of the branches taken from the top node to the statement node. The schedule vector is constructed in a similar way by also including the iterators of the traversed loop nodes.

The execution order of statement invocations depends on the position in the program code, e.g., `S1`$(0,0)$ in Figure 3.9 is executed before `S2`$(0,0)$, indicated by the ordering vectors, and on the value of the iterators, e.g., `S2`$(0,0)$ is executed before `S2`$(0,1)$, indicated by the iteration vectors. These two aspects are combined in the *schedule vector*.

Definition 10. *The* schedule vector *of a statement is the vector with the ele-*

ments of the ordering vector as odd elements and the iterators as even elements (Figure 3.9 and 3.10). The dimension is $2D_{\mathrm{S}}+1$. The execution order of statement invocations follows the lexicographical order of their schedule vectors. The uniqueness of the ordering vectors ensures that the statement invocations are strictly ordered.

This schedule vector is an instance of the more general concept of schedule or schedule function [84, 85] (also called *scattering function* [27]). The schedule function of a statement `Sx` is a function of its iteration vector, $\theta_{\mathtt{Sx}}(\mathcal{I}_{\mathtt{Sx}})$, which defines the execution order of the statement invocations by mapping each iteration on a *time-stamp*, i.e. a logical execution time. A statement invocation $\mathtt{Sx}(i_1, j_1, \ldots)$ precedes another statement invocation $\mathtt{Sy}(i_2, j_2, \ldots)$ if and only if $\theta_{\mathtt{Sx}}(i_1, j_1, \ldots) \prec \theta_{\mathtt{Sy}}(i_2, j_2, \ldots)$. For the example given above the schedule function is expressed as the schedule vector or

$$\begin{aligned}
\theta_{\mathtt{S1}}(i,j) &= (0,i,0,j,0) \\
\theta_{\mathtt{S2}}(i,j) &= (0,i,0,j,1) \\
\theta_{\mathtt{S3}}(k) &= (1,k,0) \ .
\end{aligned}$$

By altering the schedule function (vector) the execution order of the statement invocations will be changed. This is a way to describe loop transformations in the polyhedral model. The schedule function may be one-dimensional or multidimensional, as the schedule vector presented above. This is referred to as scheduling in one-dimensional or multidimensional time, respectively [84, 85]. In this chapter the choice of the schedule vector will always result in a total order of the statement invocations. A partial order can also be used, e.g., to describe potential parallelism. In chapter 4 one-dimensional schedules will be used that map different invocations on the same time-stamp.

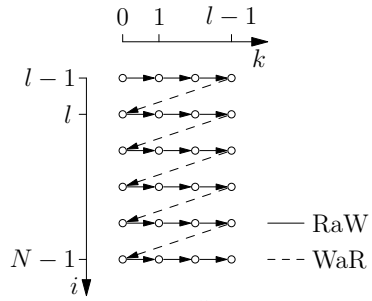
3.4.3 Dependences

If a statement invocation reads a data value that is written by another statement, there is a dependence between those invocations and the executions should stay in the same order, to ensure correct execution results. Two invocations without dependences may change places. Three types of dependences can be distinguished:

Read after Write (RaW) A data element is first written by one statement invocation and then read by another statement invocation. Also called producer-consumer or data dependence.

```
for (i=l-1;i<=N-1;i++){
  for (k=0;k<=l-1;k++){
    if (k==0)
      s=0;
    s=s+a[i-k]*f[k];
    if (k==l-1)
      b[i]=s;
  }
}
```

(a)



(b)

```
for (i=l-1;i<=N-1;i++){
  for (k=0;k<=l-1;k++){
    if (k==0)
      s[i][0]=0;
    s[i][k+1]=s[i][k]+a[i-k]*f[k];
    if (k==l-1)
      b[i]=s[i][k+1];
  }
}
```

(c)

Figure 3.11: Program code (a) of example 5 for calculating a convolution with geometric representation of true and false dependences as arrows between points in the iteration domain (b). After conversion of the code to single assignment (c) the false dependences are removed.

Write after Read (WaR) A data element is read by one statement invocation before its value is overwritten by another statement invocation. Also called consumer-producer or anti-data dependence.

Write after Write (WaW) A data element is written by a first statement invocation and then overwritten by another statement invocation. Also called producer-producer or output dependence.

The first (RaW) is called a *true dependence* as there is a value passed from the first to the second statement invocation. The other two cause a *false dependence* since a same data location is chosen to contain two different data values. If a different data location is chosen to contain the different data values these dependences are removed.

A program in which each data location is written on at most once is said to have the *single assignment* property [83]. Figure 3.22 gives an example of single assignment code. Several techniques exist to rewrite

a program into a single assignment form.

Example 5. *Figure 3.11(a) shows a program fragment that calculates the convolution b of a signal a and a filter with coefficients f_k according to Equation A.1. The "`+=`" operator is avoided for not hiding the read from the scalar s. All dependences are caused by accesses to this scalar. The assignments "`s=0;`" and "`b[i]=s;`" could be put in front of and behind the inner loop but are put within the inner loop to have all the dependences in a single 2-D space as shown in Figure 3.11(b). The WaR dependences can be removed by expansion of the scalar s into a 2-D array as shown in Figure 3.11(c). As a result, the different iterations of the outer loop can be executed in any order or in parallel.*

If the array indices are linear expressions of the iterators and parameters, the dependences between the statement invocations can also be represented as the integer points in polyhedra or unions of polyhedra, which are called *dependence domains* [31].

Definition 11. *Consider the statements* `Sy` *and* `Sx`*. Their* dependence domain, $\mathcal{D}_{\texttt{Sx}\delta\texttt{Sy}}$, *using the simplified notation of [31], is defined as*

$$\begin{array}{c} \texttt{Sy}(i_1) \textit{ depends on } \texttt{Sx}(i_0) \\ \Updownarrow \\ (i_0, i_1) \in \mathcal{D}_{\texttt{Sx}\delta\texttt{Sy}} \ , \end{array} \tag{3.16}$$

with $\texttt{Sy}(i_1)$ *and* $\texttt{Sx}(i_0)$ *arbitrary statement invocations.*

For the example of Figure 3.9 the non-empty dependence domains are:

$$\begin{aligned} \mathcal{D}_{\texttt{S1}\delta\texttt{S2}} &= \{(i_0, j_0, i_1, j_1) \in \mathcal{D}_{\texttt{S1}} \times \mathcal{D}_{\texttt{S2}} | j_1 = j_0 \wedge i_1 = i_0\} \\ \mathcal{D}_{\texttt{S2}\delta\texttt{S2}} &= \{(i_0, j_0, i_1, j_1) \in \mathcal{D}^2_{\texttt{S2}} | i_0 + j_0 = i_1 + j_1 \wedge i_1 = i_0 + 1\} \\ \mathcal{D}_{\texttt{S2}\delta\texttt{S3}} &= \{(i, j, k) \in \mathcal{D}_{\texttt{S2}} \times \mathcal{D}_{\texttt{S3}} | k \leq i + j \leq k + 1 \wedge j = 0\} \ , \end{aligned}$$

from which the dependence domains for Figure 3.8 can simply be derived since $\mathcal{D}_{\texttt{S12}\delta\texttt{S12}} = \mathcal{D}_{\texttt{S2}\delta\texttt{S2}}$ and $\mathcal{D}_{\texttt{S12}\delta\texttt{S3}} = \mathcal{D}_{\texttt{S2}\delta\texttt{S3}}$. Figure 3.14(a) shows these dependences with arrows.

In fact, these dependences are still too restrictive. Using the associativity and commutativity of the addition over $\mathbb{R}$, the operations that work on a certain element of array B (`S2`) can be executed in any order, as long as they are all finished before the read of that element by an

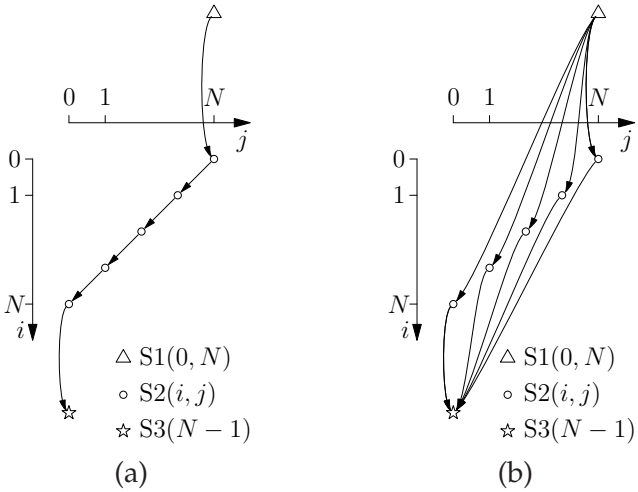


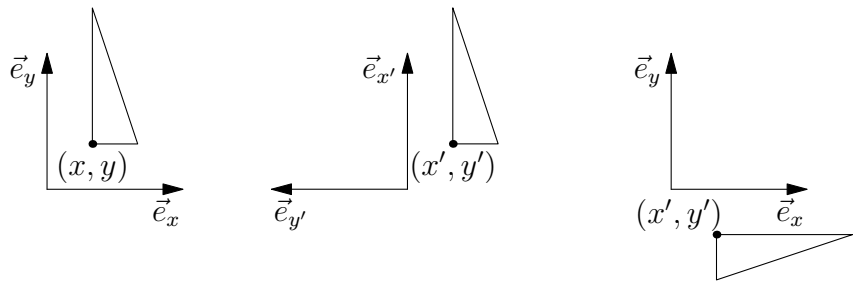
Figure 3.12: Dependences of the statement invocations accessing $B[N]$ without (a) and with (b) taking commutativity of the addition into account.

invocation of statement S3, and the element of B is first initialized by the corresponding invocation of S1. With this in mind the dependence domains would look like:

$$\begin{aligned}
\mathcal{D}_{\mathtt{S1}\delta\mathtt{S2}} &= \{(i_0, j_0, i_1, j_1) \in \mathcal{D}_{\mathtt{S1}} \times \mathcal{D}_{\mathtt{S2}} | j_0 = i_1 + j_1\} \\
\mathcal{D}_{\mathtt{S2}\delta\mathtt{S2}} &= \phi \\
\mathcal{D}_{\mathtt{S2}\delta\mathtt{S3}} &= \{(i, j, k) \in \mathcal{D}_{\mathtt{S2}} \times \mathcal{D}_{\mathtt{S3}} | k \leq i + j \leq k + 1\} \ .
\end{aligned}$$

This is illustrated in Figure 3.12 for a selection of the statement invocations. The fact that more arrows are shown although there are less dependences might be misleading. One should note that only the *direct dependences* are shown and included in the dependence domains. A lot of *indirect dependences* exist due to the transitivity of the dependences (if B depends on A and C depends on B then C depends on A). The total set of direct and indirect dependences is thus the transitive closure of the dependence domains. Since when a transformation does not violate direct dependences then also the indirect dependences are satisfied, it suffices to study the direct dependences.

Writing statement S2 in a single assignment form would destroy this extra freedom. A method to analyze commutativity using symbolic analysis can be found in [141]. Also fractal symbolic analysis [129] is under some conditions more powerful than dependence analysis. Typically, analysis techniques only give sufficient and not necessary con-



(a) (b) (c)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

(d)

Figure 3.13: The change of co-ordinate system ($\vec{e}_{x'} = \vec{e}_y$, $\vec{e}_{y'} = -\vec{e}_x$)(b) or the linear transformation (rotation)(c) are both described by the same mathematical formula (d).

ditions for the semantic equivalence of two schedules of the statement invocations.

In the rest of this work dependences will be based on memory accesses, without considering associativity and commutativity, unless stated otherwise.

3.5 Loop transformations

The purpose of loop transformations is already discussed in Section 3.1: reordering the execution of statement invocations to improve locality, burst usage and parallelism. This section will show how loop transformations can be described using the polyhedral model and will demonstrate on an example how the locality can be improved by applying a sequence of loop transformations.

3.5.1 Principle and example

The starting point is a program from which the iteration domains of the statements and their initial ordering and schedule vector can be derived. Program transformations in the polyhedral model can be spec-

ified by well chosen transformations of the schedule functions [27]. They modify the source polyhedra into target polyhedra containing the same statement invocations but in a new coordinate system, thus with a new lexicographical order. The new iterators are used to scan the image of the domains under applying the schedule function.

With this definition the iteration domains are kept untouched but only expressed within another co-ordinate system. In the informal definition of loop transformations sometimes terminology will be used that expresses a transformation of the iteration domain, while the actual implementation only changes the schedule function. This is analogous to the resemblance of linear transformations with a fixed co-ordinate system and transformations of the co-ordinate system itself. They are both described with the same formulas (Figure 3.13). However, the representation of some transformations in the polyhedral model requires transforming both the iteration domains and the schedule vectors, e.g., when new iterators augment the dimension of a statement domain (e.g., strip-mine) or new statements are created (e.g., peeling) [90, 91].

Below, an example sequence of transformations will be described that leads from the code in Figure 3.1 to the code in Figure 3.2. An overview of the schedule vectors and statement invocation arguments during the transformation sequence is found in Table 3.3. The corresponding polyhedral representations are found in Figure 3.14. The original domains are shown in Figure 3.14(a). The iterations of `S2` are represented by circles and those of `S3` by stars. The arrows indicate the dependences through the reuse of elements of array B. The schedule vectors of the statements are 5- and 3-dimensional, respectively, and are projected to a common 2-dimensional space. The execution order is from left to right and from top to bottom (scan-line order).

The reuses in the first loop nest can be put along one axis by a loop transformation called skewing (Table 3.4). This transformation alters the schedule of statement `S2`:

$$\theta_{\texttt{S2}}(i,j) = (0, i, 0, i+j, 0) \ .$$

With the introduction of new iterators $i' = i$ and $j' = i + j$ this is expressed as the interleaving of the old ordering vector $(0, 0, 0)$ and a new iteration vector (i', j') with corresponding mapping of the statement invocations between the two iteration spaces:

$$\theta_{\texttt{S2}}(i',j') = (0, i', 0, j', 0)$$
$$\texttt{S2}(i,j) \mapsto \texttt{S2}(i', j' - i') \ .$$

Table 3.3: Schedule vector during the example sequence of transformations (Figure 3.14) expressed as a function of the original iterators and as a function of new iterators that scan the image of the domains under the schedule function, i.e. the time domain, in lexicographical order. The statement invocations are mapped to corresponding invocations in function of the new iterators.

	(a) Original	(b) After skewing	(c) After interchange	(d) After fusion	(e) After shift
$\theta_{\texttt{S2}}$(orig it)	$(0, i, 0, j, 0)$	$(0, i, 0, i+j, 0)$	$(0, i+j, 0, i, 0)$	$(0, i+j, 0, i, 0)$	$(0, i+j, 0, i, 0)$
$\theta_{\texttt{S2}}$(new it)	$(0, i, 0, j, 0)$	$(0, i', 0, j', 0)$	$(0, i'', 0, j'', 0)$	$(0, i'', 0, j'', 0)$	$(0, i'', 0, j'', 0)$
$\texttt{S2}(i, j) \mapsto$	$\texttt{S2}(i, j)$	$\texttt{S2}(i', j' - i')$	$\texttt{S2}(j'', i'' - j'')$	$\texttt{S2}(j'', i'' - j'')$	$\texttt{S2}(j'', i'' - j'')$
$\theta_{\texttt{S3}}$(orig it)	$(1, k, 0)$	$(1, k, 0)$	$(1, k, 0)$	$(0, k, 1)$	$(0, k+1, 1)$
$\theta_{\texttt{S3}}$(new it)	$(1, k, 0)$	$(1, k, 0)$	$(1, k, 0)$	$(0, k, 1)$	$(0, k', 1)$
$\texttt{S3}(k) \mapsto$	$\texttt{S3}(k)$	$\texttt{S3}(k)$	$\texttt{S3}(k)$	$\texttt{S3}(k)$	$\texttt{S3}(k' - 1)$

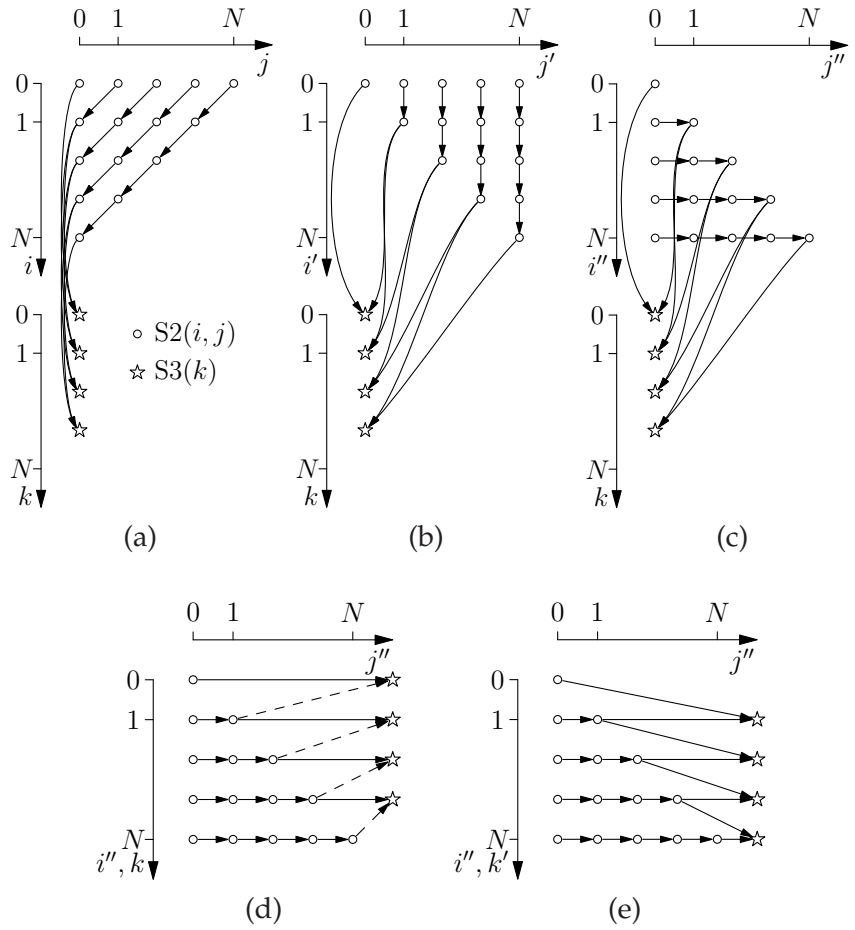


Figure 3.14: Polyhedral representation of the statements in Figure 3.1 after each step in a sequence of transformations. The 5- and 3-dimensional schedule vectors are projected to a common 2-D space. The execution order is from left to right and from top to bottom (scan-line order). Original code (a) and after skewing (b), interchange (c), fusion (d) and shift (e). The (dashed) arrows indicate (violated) dependences.

The iteration domain in the new (i', j') space is shown in Figure 3.14(b).

Now, the reuses occur between different iterations of the outer loop. By interchanging the two loops, the reuses occur between different iterations of the inner loop and inside one iteration of the outer loop (Figure 3.14(c)). The locality of the reuses of B in the first loop nest is now optimized. This skewing and interchange corresponds to the *tile i* transformation proposed in Figure 3.4.

The other major suggestion in Figure 3.4 is to fuse the loops i and k (Figure 3.14(d)), done by adjusting the ordering vector of `S3`. This transformation results in an illegal execution order. Several elements are read before they receive the correct value as indicated by the dashed arrows. This can be solved by shifting the iterations of statement `S3` over one iteration (Figure 3.14(e)).

The final result corresponds to the code in Figure 3.2 (disregarding the names of the iterators) and the histogram in Figure 3.3(b), which proves the improvement in data locality.

Note that the new iterators do not have to be introduced after each transformation step. It is possible to perform all transformations on the schedule function using the original iterators. Only during the *code generation* phase after the last transformation step, new iterators, which scan the images (in the time domain) of the iteration domains under the schedule functions, are introduced.

3.5.2 Available infrastructure

For the experiments described in this work, use was made of the WRaP-IT, URUK tool suite [57, 29, 91] and the CLooG code generator [27, 28], developed at several French laboratories (INRIA Futurs, LRI (Paris-Sud), PRiSM (Versailles), . . .). These tools are available under the GNU General Public License (GPL).

Loop transformations

In the polyhedral model loop transformations are described as transformations on the matrices representing the iteration domains and schedule functions. A way is needed to transform a program into a polyhedral representation and apply the transformations, preferably specified at a higher level than matrix operations.

The WRaP-IT infrastructure (WHIRL Represented as Polyhedra - Interface Tool) translates a program from the WHIRL intermediate representation (Winning Hierarchical Intermediate Representation Language) of the Open64/ORC (Open Research Compiler) [5] into a polyhedral representation, WHIRL Represented as Polyhedra (WRaP), and back again (after transformations).

The URUK tool (Unified Representation Universal Kernel) applies loop transformations using the WRaP intermediate format. The loop

Table 3.4: List of common loop transformations [165]. The first 5 transformations are elementary. Tiling can be constructed ad-hoc from elementary transformations. More detailed descriptions with examples are found in [165, 90]

Name	Effect
Loop Fusion	Combining two adjacent loops into one loop
Loop Peeling	Loop peeling removes (peels off) the first (or last) iteration of a loop and places it into separate code. This can be generalized into peeling off several iterations of the loop.
Loop Interchange	Interchanging two nested loops switches the inner and outer loop.
Strip-mine	Strip-mining decomposes a single loop into two nested loops. The resulting inner loop (the element loop) executes a fixed number of consecutive iterations of the original loop, called a *strip*. The outer loop (the strip loop) iterates over the strips.
Skewing	Skewing changes the iteration vectors for each iteration by adding (a multiple of) one loop index value to another loop index.
Tiling	Tiling rewrites the program as an iteration over blocks (tiles), each executing a fraction of the statement invocations. This is an extension of strip-mining to multiple nested or non-nested loops. It is usually not achieved by a single transformation but needs several transformation steps, e.g., strip-mine and interchange in a perfect loop nest.

```
skew(enclose(S2),1,2,1)
interchange(enclose(S2))
fusion(enclose(S2,2))
shift(S3,{[0,1]})
```

Figure 3.15: URUK script of the transformation sequence shown in Figure 3.14

transformation sequence can be specified in a script with each line containing one elementary transformation step. Figure 3.15 shows the script corresponding to the transformation sequence described in Section 3.5.1. Next to the elementary transformations delivered with

URUK, new user-defined transformations can be used after specifying them in a "`.def`" file. This file type allows to declare new elementary transformations as a combination of other elementary transformations and operations on the matrices representing the domains and schedule functions.

A detailed overview of the implementation of the transformation process and the definition of many common loop transformation in the polyhedral model is given by [90, 91]. The general form of a schedule function used is

$$\theta_{\mathtt{S}}\left(\mathcal{I}_{\mathtt{S}}\right) = \Theta_{\mathtt{S}} \begin{bmatrix} \mathcal{I}_{\mathtt{S}} \\ \mathcal{I}_{gp} \\ 1 \end{bmatrix}, \tag{3.17}$$

with $\mathcal{I}_S$ the vector of iterators, $\mathcal{I}_{gp}$ the vector of global parameters and $\Theta_{\mathtt{S}}$ a matrix of the form[4]

$$\Theta_{\mathtt{S}} = \left[\begin{array}{ccc|c} 0 & \dots & 0 & \beta_0 \\ \Phi_{1,1} & \dots & \Phi_{1,d+d_{gp}} & \Phi_{1,d+d_{gp}+1} \\ 0 & \dots & 0 & \beta_1 \\ \Phi_{2,1} & \dots & \Phi_{2,d+d_{gp}} & \Phi_{2,d+d_{gp}+1} \\ \vdots & \ddots & \vdots & \vdots \\ \Phi_{d,1} & \dots & \Phi_{d,d+d_{gp}} & \Phi_{d,d+d_{gp}+1} \\ 0 & \dots & 0 & \beta_d \end{array}\right]. \tag{3.18}$$

The even lines of $\Theta_{\mathtt{S}}$ (counting from 0) represent the sequential ordering of statements as described by the ordering vector. This means these lines contain 0 in the first $d + d_{gp}$ columns ($d = D_{\mathtt{S}}$ = number of iterators, d_{gp} =number of parameters) and an element of the ordering vector (called $\beta_{\mathtt{S}}$). The odd lines express the order of iterations. The first d columns contain the coefficients of the iterators and the next $d_{gp} + 1$ the coefficients of the parameters and constant. The matrix Φ, used in (3.18), is therefore split into the matrices A and Γ:

$$\Phi = \left[\begin{array}{ccc|ccc} A_{1,1} & \dots & A_{1,d} & \Gamma_{1,1} & \dots & \Gamma_{1,d_{gp}+1} \\ A_{2,1} & \dots & A_{2,d} & \Gamma_{2,1} & \dots & \Gamma_{2,d_{gp}+1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ A_{d,1} & \dots & A_{d,d} & \Gamma_{d,1} & \dots & \Gamma_{d,d_{gp}+1} \end{array}\right]. \tag{3.19}$$

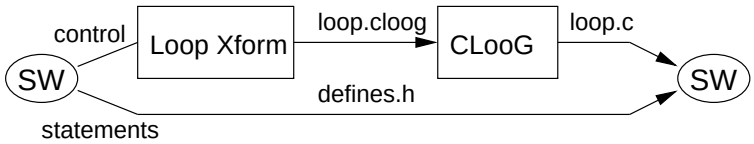


Figure 3.16: The Chunky Loop Generator (CLooG) generates (control) code from a polyhedral representation (*.cloog). Together with the statement definitions (*.h) this results in executable software.

Code generation

After all transformations have been applied, the program is still represented in the polyhedral model, i.e. a set of statements each with a domain and schedule function. Code has to be generated that iterates over the statement invocations in such a way that the lexicographical order defined by the schedule function is respected. The Chunky Loop Generator (CLooG) [28, 27, 26] performs this task. The basic concept of code generation consists of projecting the images of the iteration domains under the schedule functions on the several axes, corresponding to the (new) iteration dimensions.

A version of CLooG, called UrGenT (Uruk GeneraTor), is used by WRaP-IT to generate WHIRL after loop transformations. CLooG also exists as a stand-alone tool and library. The generated code (similar to the lower part of Figure 3.8) has to be extended with statement definitions (similar to the upper part of Figure 3.8) to create executable code (Figure 3.16).

A trade-off between code size and run-time control complexity can be made as illustrated by the following example:

Example 6. *Consider two statements with the same square iteration domain but with an offset* $(2, 2)$ *forced by the schedule function (Figure 3.17).*

$$\mathcal{D}_{\mathtt{S1}} = \mathcal{D}_{\mathtt{S2}} = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq M, 0 \leq j \leq M\}$$

$$\begin{aligned} \theta_{\mathtt{S1}} &= (0, i, 0, j, 0) \\ \theta_{\mathtt{S2}} &= (0, i+2, 0, j+2, 1) \end{aligned}$$

Code generated by CLooG v. 0.12.2 is shown in Figure 3.18 for two settings of the "`-f`*" control optimization option. Without control optimization ("*`-f`

[4]In [90, 91] the statement labels are written as superscript instead of subscript.

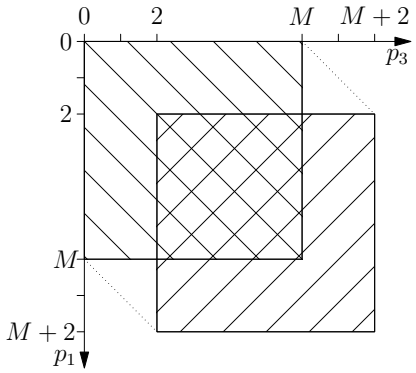


Figure 3.17: Two square domains of the statements in example 6. The dotted lines indicate the convex hull of the union of the domains.

`-1`") (Figure 3.18(a)) the iterators scan the convex hull (dotted lines in Figure 3.17) of the union of the two domains and the statements are guarded *by conditions that check if the iterator values are in the iteration domain of the corresponding statement. Note that for two iterations,* $(p1, p3) = (M + 1, 1)$ *and* $(p1, p3) = (1, M + 1)$*, none of the statements are invoked, which leads to useless execution time. Also the run-time evaluation of the conditional guards causes an overhead. With control optimization ("`-f 1`") the code is much longer (Figure 3.18(b)) but no time is wasted at run-time with useless iterations and condition checking.*

The situation of domains that are roughly the same and only have non-overlapping parts at the borders is very common after loop transformations, e.g., caused by shifts that fix violated dependences. Also many image processing algorithms have irregular behavior at the borders compared to the body of the image.

Automation of loop transformations in this work

In this chapter all loop transformations and sequences of them are specified manually, directly in the polyhedral model or at a higher level using URUK scripts. The application of them is automated by the tools mentioned above. Only in Chapter 4 transformations, or to be more exact schedules, will be constructed in an automated way.

```
for (p1=0;p1<=M+2;p1++) {
  for (p3=max(p1-M,0);p3<=min(p1+M,M+2);p3++) {
    if ((p1 <= M) && (p3 <= M)) {
      S1(p1,p3) ;
    }
    if ((p1 >= 2) && (p3 >= 2)) {
      S2(p1-2,p3-2) ;
    }
  }
}
```

(a)

```
for (p1=0;p1<=1;p1++) {
  for (p3=0;p3<=M;p3++) {
    S1(p1,p3) ;
  }
}
for (p1=2;p1<=M;p1++) {
  for (p3=0;p3<=1;p3++) {
    S1(p1,p3) ;
  }
  for (p3=2;p3<=M;p3++) {
    S1(p1,p3) ;
    S2(p1-2,p3-2) ;
  }
  for (p3=M+1;p3<=M+2;p3++) {
    S2(p1-2,p3-2) ;
  }
}
for (p1=M+1;p1<=M+2;p1++) {
  for (p3=2;p3<=M+2;p3++) {
    S2(p1-2,p3-2) ;
  }
}
```

(b)

Figure 3.18: Code generated for the statements of example 6, with control optimization options "`-f -1`" (a) and "`-f 1`" (b).

3.6 Composing sequences of loop transformations

3.6.1 Introductory example

In the example transformation sequence of Section 3.5.1 doing the shift before the loop fusion would have led to the same result. In general,

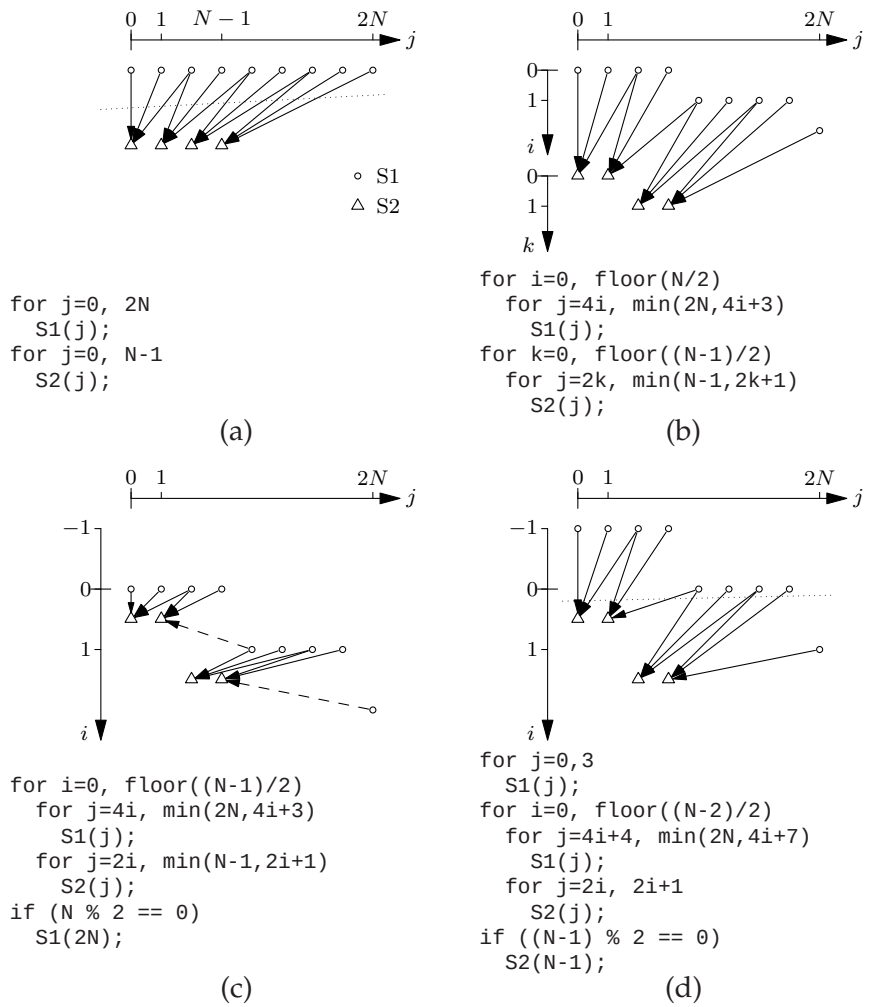


Figure 3.19: Geometrical representation of a sequence of transformations, resulting in a tiled variant. $N = 4$. The corresponding code blocks (with simplified for loop syntax as defined on page 56) are shown only for clarification and are normally not produced until the final code generation. A dotted line represents an execution point and the arrows that are cut by this line represent the data elements that have to be stored at that moment. (a) original code, (b) after strip-mine, (c) after fusion, (d) after shift.

however, changing the order of transformation steps may have an impact as is demonstrated by the following example.

Consider the following piece of code:

```
                                  //       Schedule
for (j=0;j<=2*N;j++)              //        vector
  B[j]=A[2j]+A[2j+1];             //S1(j)  (0,j,0)
for (j=0;j<=N-1;j++)
  C[j]=B[2j]+B[2j+1]+B[2j+2]; //S2(j)  (1,j,0)
```

A geometrical representation of the iteration domains of the two statements is shown in Figure 3.19(a). The horizontal axis shows iterator j, and the vertical is a projection of the ordering vector. The execution order is in scan-line order: from left to right and from top to bottom. The data dependences are indicated by arrows. It shows that for the original code all operations within one loop are independent.

This variant of the code exhibits poor data locality. First, all elements of B are produced by the first loop. Then, all elements are consumed by the second loop (dotted line in Figure 3.19(a)). The locality can be improved by interleaving the operations of the two loops, e.g., by a loop transformation called tiling. This transformation reorders the program as an iteration over blocks (or stripes in the 1-D case) in which small parts of both loops are executed (Table 3.4). Figure 3.19 shows the transformation steps leading to a tiled variant. The corresponding code blocks are shown only for clarification and are normally not produced until the final code generation. For compactness the C syntax of a loop statement "`for(i=l;i<=u;i++)`" is abbreviated to a FORTRAN like syntax "`for i=l, u`".[5] The tile size is chosen such that each tile executes 4 iterations of `S1` and 2 iterations of `S2`.

Strip-mining (Figure 3.19(b)) partitions the iteration domains into strips by adding extra iterators, i and k, respectively. From here, the schedule vectors are 5-dimensional and again projected on a common 2-dimensional space with scan-line execution order. After fusion (Figure 3.19(c)), invocations of both statements are executed in each tile. The dashed arrows indicate that some data dependences are violated. Each tile consumes an element that is only produced in the next tile. This can be solved by shifting `S1` over one tile (Figure 3.19(d)). A dotted line represents an execution point and has the executed points at one side and the still to be executed points at the other side. The execution of the program can be seen as a shift of this line, invoking statement instances that pass from one side to the other. The arrows that are cut by this line represent the data elements that are alive and have to be stored at that moment.

[5]Real FORTRAN uses "`DO`" instead of "`for`" loops.

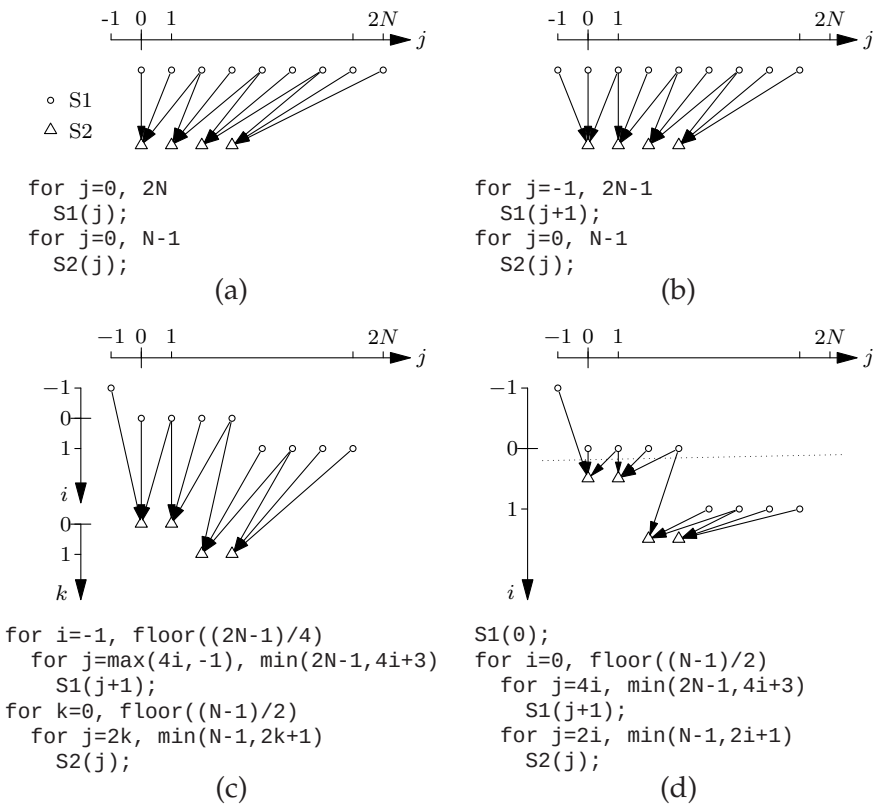


Figure 3.20: The sequence of Figure 3.19 can be improved by doing a shift before the strip-mine and fusion. This results in a better data locality. $N = 4$. (a) original code, (b) after shift, (c) after strip-mine, (d) after fusion.

After these transformations the locality has improved but is not optimal yet. Eight results produced by `S1` (originally $2N + 1$) have to be stored before they are read by `S2` (dotted line in Figure 3.19(d)) while five would have been sufficient (dotted line in Figure 3.20(d)). Better results are obtained by doing a shift (Figure 3.20(b)) before the strip-mine (Figure 3.20(c)) and fusion (Figure 3.20(d)).

3.6.2 Duplication of statement invocations

In the example in the previous section each tile consumes data generated by the previous tile and by the tile itself. As a result, the tiles can only be executed in the order indicated by the iterator i.

The tiles can be made independent by duplicate production of shared data elements. This introduces a trade-off between parallelism (number of independent tiles) and computation cost. If the tiles are large the impact of the duplicated calculations is relatively small.

After skewing (Figure 3.21(a)), all tiles but the first and possibly also the last (if N is odd), have the same *shape*. The data elements read by 2 different tiles are now produced by the invocations of `S1` for which $j = 3$. In the figures only 2 full tiles are shown ($N = 4$). For larger N there will be more tiles ($\lceil N/2 \rceil$) and shared elements ($\lceil N/2 \rceil$-1). To make the tiles independent the statement invocations producing the shared elements are duplicated. A new statement is constructed: `S1`$'$, indicated with a star, which is a copy of `S1` with only the invocations for which $j = 3$. This can be regarded as peeling off those invocations without removing them from the loop, i.e., peeling with duplication (Figure 3.21(b)). Each duplicate is shifted to the next tile and fused with the first loop within that tile (Figure 3.21(c)). The tiles are now independent and can run in parallel. Two useless statement invocations (1 if N is odd), are removed by adding constraints to the systems representing the iteration domains of `S1` and `S1`$'$ (Figure 3.21(d)).

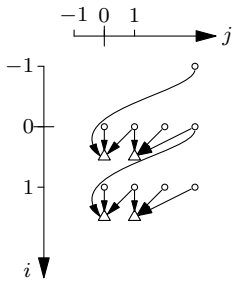
Finally, executable code can be generated from the polyhedral representation:

```
for (i=0;i<=floor((N-1)/2);i++){
  B[4i]=A[8i]+A[8i+1];            //S1'(4i);
  for (j=0;j<=min(3,2N-4i-1);j++)
     B[j+4i+1]=A[2j+8i+2]
       +A[2j+8i+3];               //S1(j+4i+1);
  for (j=0;j<=min(1,N-2i-1);j++)
     C[j+2i]=B[2j+4i]+B[2j+4i+1]
       +B[2j+4i+2];               //S2(j+2i);
  }
```

The code generator considers `S1` and `S1`$'$ as different statements. Therefore, the code is larger than needed. Knowing that they are copies of each other it is possible to remove the line with `S1`$'$ and let the iterator j around `S1` start from -1 instead of 0.

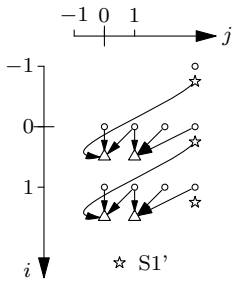
3.6.3 Commutativity of elementary loop transformations

The examples in Section 3.5.1 and 3.6.1 illustrate that sometimes the order of transformation steps is important and sometimes it is not. In



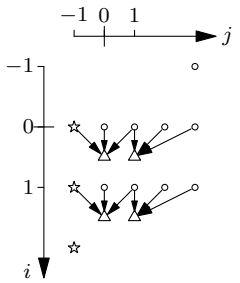

```
S1(0);
for i=0, floor((N-1)/2)
  for j=0, min(3,2N-1-4i)
    S1(j+4i+1);
  for j=0, min(N-1-2i,1)
    S2(j+2i);
```

(a)



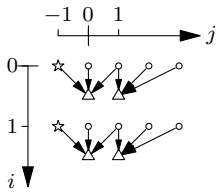

```
S1(0);
S1'(0);
for i=0, floor((N-2)/2)
  for j=0, 3
    S1(j+4i+1);
  S1'(4i+4);
  for j=0, 1
    S2(j+2i);
if ((N-1) % 2 == 0)
  for j=0, 1
    S1(j+2N-1);
  S2(N-1);
```

(b)




```
S1(0);
for i=0, floor((N-1)/2)
  S1'(4i);
  for j=0, min(3,2N-1-4i)
    S1(j+4i+1);
  for j=0, min(1,N-1-2i)
    S2(j+2i);
if (N % 2 == 0)
  S1'(2N);
```

(c)




```
for i=0, floor((N-1)/2)
  S1'(4i);
  for j=0, min(3,2N-1-4i)
    S1(j+4i+1);
  for j=0, min(1,N-1-2i)
    S2(j+2i);
```

(d)

Figure 3.21: The tiles can be made independent by duplication of shared data elements. $N = 4$. They now can run in parallel at the cost of doing some calculations twice. (a) after skewing, (b) after peeling with duplication, (c) after shift and fusion, (d) after adding constraints.

this section this is discussed in more detail. Two elementary transformations that affect different statements or disjoint sets of statements do always commute (in the polyhedral representation). Therefore, this case is not considered below. We consider the permutation of transformations that affect at least one statement in common and find four cases:

1. Some transformation steps can be applied in any order without altering the description of the transformation steps. For example, two shifts, a shift and a fusion, two loop interchanges that work on distinct couples of iterators.

2. Some transformations can be applied in a different order with an appropriate adjustment of the operands. For example, a unimodular transformation (multiplication of A and Γ in (3.19) with a unimodular matrix U, e.g., loop interchange and skewing) preceded by a shift (addition of a vector M to Γ) can be written as the same modular transformation followed by a transformed shift.

$$\begin{aligned} A' &= UA \\ \Gamma' &= U(\Gamma + M) = U\Gamma + UM = U\Gamma + M' \end{aligned}$$

3. In some cases the expressiveness of one order of two kinds of transformations is larger than the other. For example, one shift before a peeling can be expressed as two (identical) shifts after the peeling, working on the main loop and the peeled-off part, respectively. In general, shifts after the peeling can not be replaced by shifts before the peeling as the latter can not control the deviation between the two different statements created by the peeling.

4. For a last class of cases, each order of two kinds of transformations can express transformations that can not be expressed by the other order. As shown in the example in Section 3.6.1 a shift before a strip-mine can control the grouping of statement invocations in the tiles, which is not possible with shifts after the strip-mine. However, the strip-mine introduces a new variable and only shifts after the strip-mine can independently affect the element and strip loop iterator (Table 3.4).

3.6.4 A canonical order

One problem in automated search space exploration of loop transformation is that different transformation sequences may lead to the same result. This is, a.o., caused by the fact that some loop transformations commute (possibly with adjustments in the operands). Therefore, it is worthwhile to wonder if some kind of canonical order of transformation steps could be constructed. If such an order would exist it would be easier to avoid the generation of different sequences that have equivalent results.

Under certain restrictions a canonical order exists. For example, when the transformations are restricted to unimodular transformations and shifts (cf. case 2 in Section 3.6.3). However, in the general case, when the number of statements and iterators may change, the situation becomes much more complex. Then, at least a *partial canonical order* can be constructed. For example, if the order of all transformation steps that are not shifts is given, the shifts can be moved over other steps in the transformations sequence until only shifts remain at the beginning of the sequence and after each transformation step that is of a kind for which not having a shift after it would reduce the expressiveness (cf. cases 3 and 4 in Section 3.6.3). This has been used to examine the differences between the transformation sequences LB HV and LB VH and between RC HV and RC VH mentioned at the end of Section 3.8.

3.7 Useful transformation sequences for the 2-D IDWT

Figure 3.22 shows pseudo code of a Row-Column-wise 2-D IDWT, corresponding to the manual design described in Section A.3. This algorithm is far from optimal with regard to several criteria, such as data locality or burst mode usage. Therefore, it has to be modified in accordance to the desired properties. Variants with a better data locality can be constructed by transforming the original variant. The purpose is to reduce the external memory bandwidth, by retaining data that is frequently used in on-chip buffers. To effectively exploit the created locality, obviously memory allocation and mapping of the data is needed. We refer to [158] for a possible approach.

Many variants of the (I)DWT have been described in the literature. An overview is found in Section A.5. Since none of the cited papers

```
1: for l = k, 1
2:   R_l = R / 2^l
3:   C_l = C / 2^l
4:   for j = 0, 2C_l − 1                          // Vertical filtering.
5:     for i = 0, R_l − 1
6:       B_{l,2i,j}   = Σ_{n=−1}^{1} A_{l,i+n,j} h_{3−2n} + Σ_{n=−2}^{1} A_{l,R_l+i+n,j} g_{3−2n}
7:       B_{l,2i+1,j} = Σ_{n=−1}^{2} A_{l,i+n,j} h_{4−2n} + Σ_{n=−2}^{2} A_{l,R_l+i+n,j} g_{4−2n}
8:   for i = 0, 2R_l − 1                          // Horizontal filtering.
9:     for j = 0, C_l − 1
10:      A_{l−1,i,2j}   = Σ_{n=−1}^{1} B_{l,i,j+n} h_{3−2n} + Σ_{n=−2}^{1} B_{l,i,C_l+j+n} g_{3−2n}
11:      A_{l−1,i,2j+1} = Σ_{n=−1}^{2} B_{l,i,j+n} h_{4−2n} + Σ_{n=−2}^{2} B_{l,i,C_l+j+n} g_{4−2n}
```

$$A_l = \begin{bmatrix} LL_l & HL_l \\ LH_l & HH_l \end{bmatrix}, B_l = [L_l\ H_l] \text{ for } l \geq 1 \text{ and } A_0 = LL_0.$$

Figure 3.22: Simplified representation of the Row-Column-wise 2-D IDWT. R and C are parameters representing the number of rows and columns of the image that is transformed over k levels. For each level l, A_l and B_l are two-dimensional arrays used to store the different subbands. The vectors g and h contain the filter coefficients, i.e., the impulse response, of the wavelet synthesis filters G and H. They have lengths 9 and 7, respectively.

mentions the path to a certain implementation it is assumed that they are all constructed by doing code transformations manually. This task is error-prone, time consuming and makes it hard to explore many variants of an implementation. As mentioned in Section 3.5.2, tools exist to automate the application of loop transformations. The user only has to specify the sequence of transformations to perform. This sequence can be found by manual analysis of the code, possibly aided by tools, such as SLO (Section 3.2.2). The transformation tools can also verify the correctness of the result of a transformation by verifying that no data dependences are violated.

We would like to use the polyhedral model to improve the locality of the IDWT. However, the code in Figure 3.22 contains an exponential function in the loop boundaries (R_l and C_l) and therefore it does not fit into the model since the loop bound expressions are not linear. This problem can be resolved in two ways. If the number of levels k is known the outer loop can be unrolled. Else one can restrict the polyhedral model to the body of the outer loop, where 2^l becomes a parameter,

Table 3.5: Primary transformations expressed as a sequence of elementary transformations. The first three generate IDWT variants starting from the original row-column-based variant. The fourth transforms a tiled variant, created after applying Tile V or Tile H (or both as in Section 3.8), into a variant with overlapping tiles. The numbers indicate the order of the transformation steps. Steps between brackets may become unnecessary when combining several of these sequences.

	Stretch	Interchange	Fusion	Shift	Strip-mine	Peeling	Peeling with duplication	Skew	Add Constraint
RC2LB	1	2	3	(4)					
Tile V		(2)	4	3	1				
Tile H		4	6	2,5	3	1			
Overlap			4	3			2	(1)	5

and put the result after transformation back inside the l-loop. Writing the program in a single assignment form adds flexibility by removing false dependences.

The poor locality of the Row-Column-Based IDWT can be improved in several ways. The accesses of the horizontal and vertical filtering can be brought closer together but also the accesses of the different transformation levels (loop l). Below we will discuss three transformation sequences, each improving the locality of the original algorithm in a different way. We will call them *primary sequences*. A fourth sequence (Overlap) starts from a tiled variant, to make the tiles overlapping. Table 3.5 shows how the sequences are composed of elementary loop transformations (cf. Table 3.4). A similar table can be constructed for the DWT. Section 3.8 will show that the primary sequences can be combined to form larger sequences.

3.7.1 From row-column-based to line-based (RC2LB)

Initially, the horizontal and vertical filtering scan the image in different directions. Doing the vertical filtering line by line (interchange of

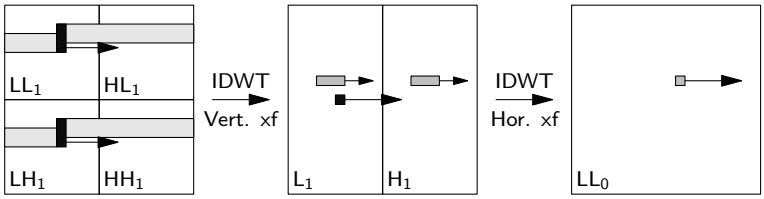


Figure 3.23: Vertical filtering in a horizontal scan order leads to a line-based IDWT

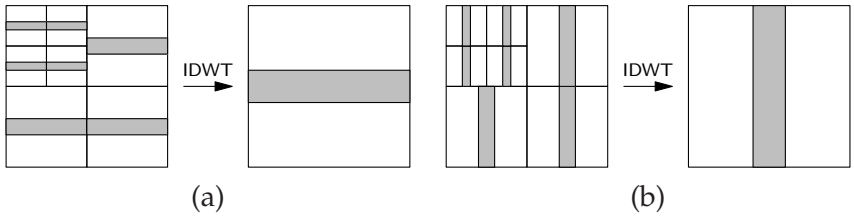


Figure 3.24: Vertical and horizontal tiling

the loops on line 4 and 5 of Figure 3.22, left side of Figure 3.23), allows to interleave (or execute in parallel through pipelining) the horizontal and vertical filtering operations (fusion of the loops with iterator i). As a result less data has to be stored; only several lines instead of an entire frame. Stretching the vertical filtering in the vertical direction, i.e., doubling the range of iterator i from R_l to $2R_l$ (line 5 in Figure 3.22) and only execute for even values, is needed to adjust the rate of the vertical transformation to the rate of the horizontal transformation, for which iterator i goes from 0 to $2R_l - 1$. A shift is needed where dependences are violated. The resulting *line-based* IDWT (LB) is still done level by level. Coarse-grain parallelism can be introduced by pipelining the different transformation levels as in [54].

3.7.2 Vertical tiling (Tile V)

Vertical tiling divides the iteration domains into horizontal stripes, with heights proportional to the subband they are contained in (Figure 3.24(a)). Operations are not done level by level but for all levels within a set of corresponding stripes, one in each subband. This is reached by strip-mining all loops with iterator i by a factor proportional to R_l, so that each subband is divided into the same number of stripes. Loop interchanges are needed to make the resulting strip

iterators (Table 3.4), which iterate over the stripes within one subband, the most significant, i.e., outside the unrolled l-loop. After fusion there remains only one strip iterator that iterates over sets of data consisting of one stripe in each subband (Figure 3.24(a)). Shifts restore violated dependences. This sequence is similar to the tiling demonstrated in Figure 3.20. We will call this variant a vertically tiled IDWT (Tile V). Since this loop transformation sequence combines operations of different wavelet transformation levels, it can only be applied in the polyhedral model if the number of levels k is fixed and the l-loop is unrolled. However, for any value of k a tiling transformation sequence can be constructed. Manually, it is possible to construct a tiled implementation where k is a parameter again.

3.7.3 Horizontal tiling (Tile H)

Horizontal tiling is analogous to vertical tiling except for an initial peeling and shift. After naively strip-mining the loops with iterator j, the first half of the stripes would do a vertical transform on elements of the LL_l and LH_l subbands, generating elements of the L_l, and the other half would generate the elements of H_l. Horizontal filtering in vertical stripes (Figure 3.24(b)) is only possible if corresponding stripes in the L_l and H_l band (C_l apart) are produced simultaneously. Else, the horizontal transformation could only start after more than half of the stripes are processed by the vertical filtering, i.e., when samples from both the L_l and H_l band are available. Therefore, the left and right halves of the vertical filtering domains (first and last C_l iterations of line 4 in Figure 3.22), producing the L_l and H_l band, respectively, are split (peeling) and interleaved (shift of the right half over $-C_l$).

3.7.4 Overlap

After Vertical or Horizontal tiling, some intermediate results at the edges of a stripe are also used by neighboring stripes, potentially leading to large overlap buffers and inter-stripe dependences. To make the different blocks or stripes independent of each other the statement invocations calculating these data can be duplicated, generating overlapping stripes. This decreases the buffer size and increases parallelism at the cost of extra bandwidth and calculations. This is already demonstrated by the process leading to Figure 3.21(d).

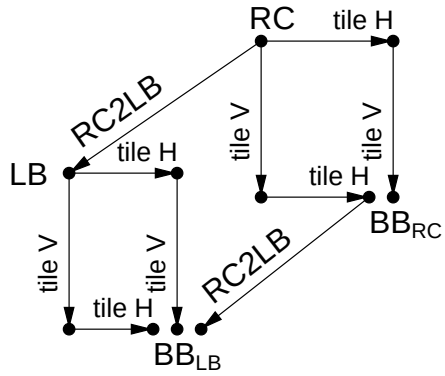


Figure 3.25: Tree of the IDWT variants listed in Table 3.6.

3.8 Building sequences from subsequences

The locality of the IDWT can be improved in different ways, as shown by the sequences described in Section 3.7, each dealing with one cause of the poor locality. Therefore, the combinations of these sequences are promising when one wants to tackle several causes at the same time. Although the application of transformation sequences is automated, still substantial effort of the user was needed to figure out how a primary sequence could be constructed from elementary transformations, i.e., those offered by the tool URUK.

The described primary sequences can be combined to form larger sequences, just by applying the URUK scripts in sequence. Only minor interventions are needed, e.g., adjusting code labels and other transformation operands (mainly shift offsets), omitting redundant transformation steps (between brackets in Table 3.5), typically taking in the order of ten minutes all together, much less than for the construction of the primary sequences. This way a space of variants, e.g., as shown in Figure 3.25, can be explored in a short time. The abstraction level is raised from combining elementary transformation steps (Section 3.7) to combining application specific primary sequences (this section), by which larger steps in promising directions can be taken (cf. number of elementary transformation steps in Table 3.6). Below we will discuss some variants and examine their properties.

In Table 3.6 the temporal locality is indicated by the reuse distances. Recall that in a cache or local memory with the Least Recently Used

Table 3.6: IDWT variants with the order of primary sequences performed. The number of elementary transformation steps is given for a 3-level IDWT ($k = 3$). The outer loop (iterator l) has been unrolled to fit in the polyhedral model. As a result, many transformation steps occur three times; once for each level. The number of lines in the generated code is given as generated by CLooG [26] with (a) and without (b) control optimizations (command line options "`-f 1`" and "`-f -1`", respectively, cf. example 6). The temporal locality is indicated by the number of accesses with a reuse distance larger than a given size (for an image of 72×88 pixels). Detailed results are found in appendix B.

	RC2LB	Tile H	Tile V	#El. steps	# lines C		# Reuse Distances			
					(a)	(b)	$> 2^9$	$> 2^{11}$	$> 2^{13}$	$> 2^{14}$
RC				0	32	32	10202	8954	4868	489
RC H		1		28	166	55	12731	10981	6035	0
RC HV		1	2	54	456	69	14213	10695	2114	0
RC V			1	19	98	46	11845	9106	2103	0
RC VH		2	1	47	400	57	14267	10801	2120	0
VH LB	3	2	1	62	1887	75	15660	10876	2008	0
LB	1			18	83	56	33646	1881	1119	211
LB H	1	2		40	926	71	14871	13124	6508	0
LB HV	1	2	3	53	2765	73	16617	12456	6748	0
LB V	1		2	27	425	64	34242	4678	144	0
LB VH	1	3	2	52	5639	75	16685	11899	774	0

(LRU) replacement policy, data is retained between reuses if and only if the corresponding reuse distance is smaller than the cache or buffer size [42]. Based on this the number of cache misses can be plotted as a function of the cache size as in Figure 3.26. The detailed measurement results are found in appendix B. They indicate that no variant outperforms another one for all possible cache sizes.

As already mentioned in Section 3.7, the row-column-based IDWT (RC) has a bad data locality. An entire frame has to be stored between the vertical and horizontal filtering. In a line-based variant (LB) this is reduced to several lines. If the lines are too long to be stored on-chip, horizontal tiling may help by dividing the lines in smaller parts (LB H). One can see in Figure 3.26 that the misses of LB are strongly reduced by

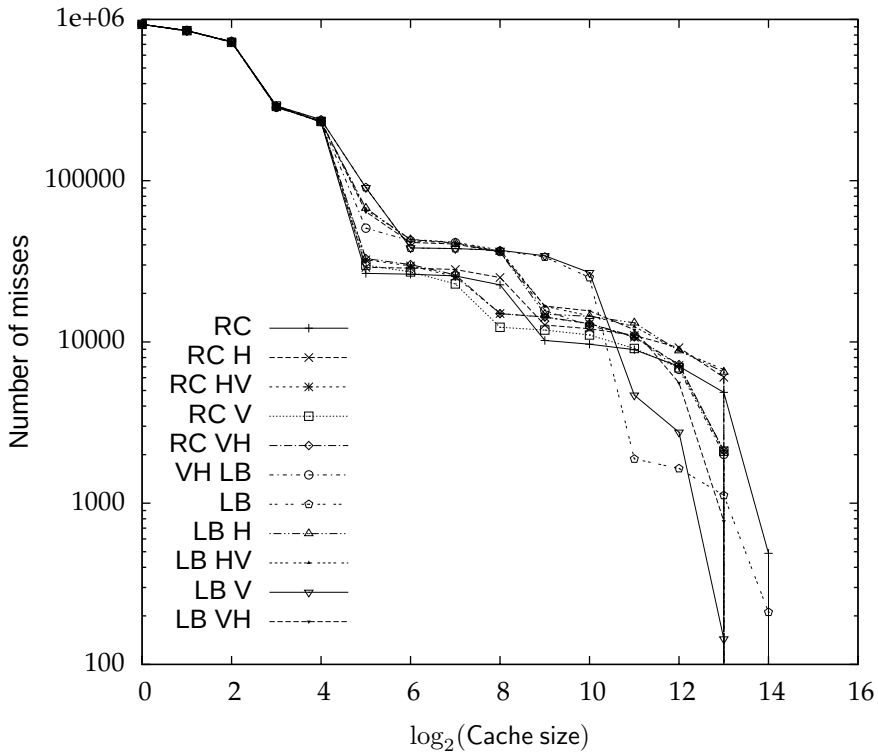


Figure 3.26: Cache misses in function of cache size (assuming LRU) based on the reuse distance histograms (Appendix B).

increasing the cache size from 2^{10} to 2^{11}, i.e. when the cache becomes large enough to save several lines of an image, while LB H already takes a step down (be it a smaller one) at a size of 2^9. However, in most cases LB H is outperformed by RC H.

If the IDWT is done level by level (RC and LB), each reconstructed LL-subband has to be stored between the different transformation levels. When the different levels run simultaneously, the buffers between them can be reduced. This is possible in hardware by implementing separate designs for each level and let them work in a pipe-line [54], or in software by placing each level in a separate thread. This is only beneficial if the scan-patterns of the different levels correspond (LB and not RC). Another way to reduce the data stored between levels (for both LB and RC) is to interleave the operations of different levels by tiling.

A block-based IDWT [167] can be reached in several ways. First

tiling vertically and then horizontally (RC VH, LB VH and VH LB) or vice versa (RC HV and LB HV). With (BB_{LB}) or without (BB_{RC}) transition to line-based influences the scanning order within a tile; line-wise for the former and row-column-wise for the latter. In both cases the order of scanning the tiles is the same. The order in which the primary sequences are applied, e.g., LB VH vs. LB HV and VH LB, has an influence on the resulting variant. This is illustrated by the number of generated lines of C code, but also reflects in the resulting locality. This is similar to the influence of the location of the shift in Figure 3.19 and 3.20. In a manual implementation, trying several permutations of the application of transformations is typically not considered. This demonstrates the usefulness of the proposed methodology. In a more thorough exploration the tile size should be varied, which is not done here.

The original RC variant performs quite well if little buffer memory is available. Note that the reuse distance only contains information about the temporal locality. Other important factors that determine the performance have to be considered, such as spatial locality (burst mode usage) and exploitable parallelism. Furthermore, if the IDWT is only one block in a system it is usually beneficial to adjust the data write and read pattern of successive modules to each other.

Why horizontal and vertical tiling do not commute

It is easy to understand that tiling before or after going to a line-based variant leads to different results as mentioned above. However, it is less obvious where the differences come from by interchanging the order of horizontal and vertical tiling. Therefore, we took a closer look at the transformation sequences RC HV and RC VH. With some small changes the sequences can be made equivalent.

The peeling in the horizontal tiling makes that the vertical tiling works on a different number of loops and statements if it is done before or after the horizontal tiling. This can be compensated by inserting three loop fusions (one for each wavelet transform level) at the end of the RC HV URUK script. Also the sequences LB HV and LB VH could be transformed into sequences with equivalent results by adding a series of shifts on two places in the LB HV script.

3.9 Comparison with alternative representations

3.9.1 Alternative polyhedral models

As announced in Section 3.4.2, the schedule vector used until now, defining a multidimensional schedule with dimensions for iterators and dimensions for the ordering of statements, is only one possible format of the schedule function.

Early versions of polyhedral models only dealt with perfect loop nests and had schedule vectors of a dimension equal to the number of iterators. Applying loop transformations then consisted of applying linear transformations on the schedule vector (= iteration vector). In the resulting schedule vector each element is a linear combination of the iterators (and possibly also parameters).

In the affine 1-dimensional scheduling case [84], the schedule is a single, i.e. 1-dimensional, linear combination of the iterators and parameters:

$$\theta_{\mathtt{Sx}}(\mathcal{I}_{\mathtt{Sx}}) = u \begin{bmatrix} \mathcal{I}_{\mathtt{Sx}} \\ \mathcal{I}_{gp} \\ 1 \end{bmatrix},$$

with u a vector.

In [137] Pouchet et al. suggest that by only considering 1-dimensional affine schedules and adding some bounds on the schedule coefficients the search space of legal transformations can be made small enough for an exhaustive scan. In the examples they give, the search space consists of in the order of hundreds or thousands of schedules. For larger programs a heuristic search method is proposed, which first tries to find the optimal iterator coefficients and then the others. However, often multidimensional schedules are needed to find legal or more optimal schedules. The extension of this work to multidimensional schedules [136, 138] enlarges the search space with many orders of magnitude. Therefore, a genetic algorithm is used to traverse the optimization spaces, with satisfiable results.

3.9.2 Transformations on ASTs

Loop transformations can also be applied directly on an AST representation of a program. This avoids the work of extracting iteration domains of statements and simplifies the code generation. A loop strip-

mine, e.g., can be done by adding an extra loop node in the tree, a loop fusion by combining two nodes. The size of the internal representation is always proportional with the size of the corresponding code. After a sequence of transformation steps this number can become very large (cf. Table 3.6). An optimization option like "`-f`" can not easily be applied.

3.9.3 Advantages of the polyhedral model

The polyhedral model has several benefits over an AST representation when performing loop transformations on software as shown in [57, 90, 91].

The application of a long sequence of transformations often leads to a large increase in code size when using a textual representation and thus also an AST representation grows rapidly. As can be seen in Figure 3.19(d), extra code is needed at the borders of the iteration domains. For larger examples or when more parameters exist this only becomes worse. The polyhedral representation in contrast only increases slightly. The resulting code sizes in Figure 3.6 illustrate this. This code increase makes working on ASTs more complex and may inhibit long transformation sequences.

The polyhedral model allows to, e.g., fuse loops with different bounds or tile non-perfectly nested loops. Other methods fail to recognize (by pattern-matching) the opportunity to perform these transformations. For example, in Figure 3.10 a loop interchange of i and j is not possible because the upper loop bound of j is a function of i. In the polyhedral model loop bounds are only calculated by the code generation after the transformations are performed and the interchange is simply done by switching i and j in the schedule vector.

Often, a long sequence of enabling transformations is needed before the actual optimizing transformations are possible. The application of one transformation might produce code that inhibits another transformation. When using the polyhedral model, code is only generated after the last transformation. The intermediate representation offers more flexibility as invocations of statements are not bound to a control structure.

3.10 Conclusions

Loop transformations are indispensable for improving aspects of an algorithm implementation, such as locality and parallelism. A polyhedral representation allows to automate the application of loop transformations. This makes exploring a part of the loop transformation design space more feasible. However, to fully automate the choice of transformations to apply, a lot of research is still needed.

When applying a long sequence of transformations, the order of transformation steps may have an impact on the final result. We have demonstrated that by combining subsequences long transformation sequences can be constructed more easily. It is an open question if the adjustments needed to have a correct subsequence combination can be automated. Some tasks, such as adjusting the vectors over which statements are shifted, appear likely to be automated. To automate the adjustment of labels and strip-mine factors probably some deeper understanding (future research) is needed.

Chapter 4

Bounds on quasi-polynomials for static program analysis

To guide loop transformations, some desired program properties, such as locality or memory usage, have to be evaluated. This can be done by profiling a program run or by static analysis of the code. In the latter case, the evaluation problem can often be expressed as counting the number of integer points in a (parameterized) polyhedron. The result will then be a quasi-polynomial in the parameters. In some cases, the extremum of such a quasi-polynomial over the integer points in a polyhedron has to be found. This will be the focus of this chapter.
Several techniques exist to bound the range of polynomials over continuous domains. The maximal difference between the continuous- and discrete-domain extrema of polynomials will be studied. This allows to know when the continuous-domain extrema can be used as a sufficient approximation of the discrete-domain extrema. Finally, the results will be used to find bounds on quasi-polynomials, by converting them into (sets of) polynomials.

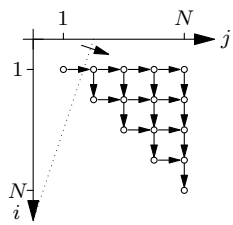
4.1 An example of static program analysis

As described in Chapter 3, loop transformations can be used to improve some properties, such as temporal and spatial locality and memory usage. However, to guide the transformation process these properties have to be measured or estimated in some way. This can be done by

```
#define S1(i,j)                \
  a=((j>i) ? A[i][j-1] : 0); \
  b=((i>1) ? A[i-1][j] : 0); \
  A[i][j]=f(a,b);

for (i=1;i<=N;i++){
  for (j=i;j<=N;j++){
    S1(i,j);
  }
}
```

(a)



(b)

Figure 4.1: (a) Program example. (b) Iteration domain of the example in (a). The arrows indicate dependences. The dotted line represents a point of execution of the linear schedule $\theta_{\mathtt{S1}}(i,j) = i + 3j$. The origins of the arrows cut by this line correspond to the live set at that point.

profiling or by static analysis. The former gives precise numbers at the cost of program compilation and running the program for each desired set of parameter values. The latter allows a more analytic analysis of the influence of parameters but might not be applicable in all cases.

In many cases the problem of calculating program properties without running the program, but by statically analyzing the code can be reduced to a polyhedral counting problem (the exact definition follows in Section 4.2), e.g., counting the number of points in a $\mathbb{Z}$-polytope. In [40] the calculation of reuse distances is expressed as such a counting problem. In some cases an extremal value of the solution of a counting problem is needed, as is demonstrated in the example below.

Consider the program in Figure 4.1(a). The number of array elements accessed is equal to the number of points in the iteration domain

$$\mathcal{D}_{\mathtt{S1}} = \{(i,j) \in \mathbb{Z}^2 \mid 1 \leq i \leq N, i \leq j \leq N\} \ , \tag{4.1}$$

and thus is equal to

$$|\mathcal{D}_{\mathtt{S1}}| = \frac{N(N+1)}{2} \ .$$

However, less memory is needed if an element is only kept in memory as long as its value is needed by a future statement invocation. The arrows in Figure 4.1(b) indicate these *Read-after-Write dependences*. The

set of dependences can be written as

$$\begin{aligned}\mathcal{D}_{\mathtt{S1}\delta\mathtt{S1}} \quad = \quad & \{(i_1, j_1, i_2, j_2) \in \mathcal{D}^2_{\mathtt{S1}} \mid \\ & (i_2, j_2) = (i_1, j_1) + (0, 1) \vee (i_2, j_2) = (i_1, j_1) + (1, 0)\} \ .\end{aligned}$$

After the execution of $\mathtt{S1}(1, N)$, the elements $A[1][2]$ till $A[1][N]$ are in the memory. After this iteration, the memory used decreases monotonically. Each time a new element is produced, one or two elements are consumed for the last time and can be removed. The memory usage is thus $N - 1$.

By changing the execution order of the statement invocations the memory usage can be reduced. We introduce a one-dimensional affine (or linear) schedule [84], $\theta_{\mathtt{S1}}(i, j) = s_1 i + s_2 j$ that defines a new execution order. In a valid schedule, data should be produced before it is consumed:

$$V(s_1, s_2): \quad \forall (i_1, j_1, i_2, j_2) \in \mathcal{D}_{\mathtt{S1}\delta\mathtt{S1}} : \theta_{\mathtt{S1}}(i_1, j_1) < \theta_{\mathtt{S1}}(i_2, j_2) \ .$$

The set of live elements of array A at a certain point t is the set of elements that are written before and read after the point t. Since in this example every element of array A is only written once and every other access to that element is a read, this set is equal to the set of elements that are accessed both before and after the point t, which is expressed as:

$$\begin{aligned}LE_A(t) = \{(i_1, j_1) \in \mathcal{D}_{\mathtt{S1}} \mid \exists (i_2, j_2) \in \mathcal{D}_{\mathtt{S1}} : \\ (i_1, j_1, i_2, j_2) \in \mathcal{D}_{\mathtt{S1}\delta\mathtt{S1}} \ \wedge \ \theta_{\mathtt{S1}}(i_1, j_1) < t \ \wedge \ t \leq \theta_{\mathtt{S1}}(i_2, j_2)\} \ .\end{aligned} \tag{4.2}$$

Here the assumption is made that the output data is immediately removed when ready, and input data is only fetched when it is used for the first time. In a practical implementation an input and output buffer will be needed.

The memory size needed for the execution of a valid schedule is the maximum number of elements in $LE_A(t)$ for all values of t, i.e.

$$\max_t |LE_A(t)| \ . \tag{4.3}$$

The schedule with the minimal memory requirements corresponds to

$$\min_{s_1, s_2 : V(s_1, s_2)} \max_t |LE_A(t)| \ .$$

The focus of this chapter is on finding a way to find the maximum (or minimum) of the solution of a counting problem (a *(piecewise) quasi-polynomial*, see Section 4.2) over a polyhedral domain, or in this example (4.3). Note that the value of this extremum is more important than its location.

4.2 Quasi-polynomials and polyhedral counting problems

The general form of an Ehrhart quasi-polynomial[1] is defined in terms of periodic numbers.

Definition 12. *A* rational periodic number $u(n)$ *is a function* $u : \mathbb{Z} \to \mathbb{Q}$, *such that* $u(n) = u(n')$ *whenever* $n \equiv n' \pmod{d}$, *where* $d \in \mathbb{N}$ *is called the period of* $u(n)$.

A periodic number can be represented by an array of rational numbers:

$$u(n) = [u_0, u_1, \ldots, u_{d-1}]_n \ ,$$

which means

$$u(n) = u_{n \bmod d} = \begin{cases} u_0 & \text{when } n \equiv 0 \pmod{d} \\ u_1 & \text{when } n \equiv 1 \pmod{d} \\ \vdots & \\ u_{d-1} & \text{when } n \equiv d-1 \pmod{d} \end{cases} .$$

Alternatively, $u(n)$ can be expressed using fractional parts of linear expressions. For example,

$$\left\{\frac{n}{3}\right\} + \left\{\frac{n}{2}\right\} = \left[0, \frac{5}{6}, \frac{2}{3}, \frac{1}{2}, \frac{1}{3}, \frac{7}{6}\right]_n ,$$

where the notation $\{.\}$ denotes the fractional part, i.e. $\{x\} = x - \lfloor x \rfloor$. Henceforth, we shall only use the fractional notation. It is straightforward to convert a fractional notation into an array representation. Since

$$\left\{\frac{n-1-i}{d}\right\} - \left\{\frac{n-i}{d}\right\} = \begin{cases} \frac{d-1}{d} = 1 - \frac{1}{d} & \text{when } (n-i) \in d\mathbb{Z} \ , \\ & \quad \text{i.e. } n \equiv i \pmod{d} \ , \\ -\frac{1}{d} & \text{otherwise} \ , \end{cases} \tag{4.4}$$

[1]For brevity *Ehrhart* will be omitted from here on.

the opposite can be done, e.g., by using

$$[u_0, u_1, \ldots, u_{d-1}]_n = \sum_{i=0}^{d-1} u_i \left(\frac{1}{d} + \left\{ \frac{n-1-i}{d} \right\} - \left\{ \frac{n-i}{d} \right\} \right) \quad . \tag{4.5}$$

Definition 13. *A* quasi-polynomial $f(p)$ *of degree* g *in one variable* p *is a polynomial expression in* p *over the rational periodic numbers, i.e.*

$$f(p) = \sum_{i=0}^{g} u_i(p) p^i \quad ,$$

where the $u_i(p)$ *are periodic numbers.*

The definitions of periodic number and quasi-polynomial can be extended to the multivariate case [162]. A *piecewise* quasi-polynomial is a function that is a quasi-polynomial on each element of a partition of the domain into polyhedral subdomains, called *chambers*. Our interest in quasi-polynomials stems from the following theorem.

Theorem 2 (Clauss and Loechner [55])**.** *The number of integer points in a parameterized polytope* P_p *of dimension* n *can be expressed on each chamber of a partition of the parameter space by a quasi-polynomial of degree* n *in* p*.*

Here p represents a vector of integer parameters, $p \in \mathbb{Z}^m$. The partition into polyhedral chambers originates from the fact that parametric vertices (Section 3.3.3) may only exist for a subset of the parameter values. The polyhedral chambers are defined in such a way that in each chamber a certain set of parametric vertices exist for all parameter values within that chamber. The quasi-polynomial in that chamber can then be computed from the corresponding parametric vertices.

More generally, the number of points in the integer projection of a parameterized $\mathbb{Z}$-polytope, as well as the number of solutions to a so-called Presburger formula (linear inequalities combined with the logical operators and quantifiers $\wedge$, $\vee$, $\forall$, $\exists$, $\neg$) can also be expressed as a piecewise quasi-polynomial [162, 161].

Several methods can be used to actually compute the number of integer points in such sets. An overview can be found in [159, 40]. For the experiments in this chapter we will use the `barvinok` library [160], which implements and extends the techniques proposed by Barvinok [25]. This library can generate the solution of a polyhedral counting problem as a piecewise quasi-polynomial, using either the fractional or

the array representation of periodic numbers. In our experiments, we will use the fractional representation, as it is polynomially-sized (for a fixed number of dimensions n), while the array representation can be exponentially large (even for fixed dimensions).

For integer polytopes, i.e. polytopes with integer vertices, the quasi-polynomial that gives the number of integer points in the polytope as a function of the (integer) parameters is actually a polynomial [20, 127].

Let us go back to the example in Section 4.1. We consider s_1 and s_2 as constants so that $LE_A(t)$ is defined by linear constraints. In this case the number of elements in this set can be described by a piecewise quasi-polynomial in the parameters N and t. For example, for $\theta_{\mathtt{S1}}(i,j) = i + 3j$ and $N = 100$ we get

$$|LE_A(t)| = \begin{cases} 1 & \text{if } 5 \leq t \leq 6 \\ \frac{1}{4}t + [-1, -\frac{1}{4}, -\frac{1}{2}, -\frac{3}{4}]_t & \text{if } 7 \leq t \leq 301 \\ 75 & \text{if } t = 302 \\ -\frac{3}{4}t + [301, \frac{1207}{4}, \frac{603}{2}, \frac{1205}{4}]_t & \text{if } 303 \leq t \leq 398 \\ -t + 401 & \text{if } 399 \leq t \leq 400 \\ 0 & \text{otherwise} \end{cases} . \quad (4.6)$$

If N is not given a value but is kept as a parameter the expression becomes much larger and is therefore not shown here.

4.3 Overview and methodology

An exact way to find the extrema of a (quasi-)polynomial over a discrete domain is evaluating it in every point of the domain. For large domains this is clearly infeasible. De Loera et al. [62] have shown that the problem of finding the extrema of arbitrary polynomials over the integer points in (non-parameterized) polytopes is NP-hard (even for fixed dimensions).

Several techniques exist that yield bounds on polynomials over continuous domains (Section 4.4). Continuous-domain extrema can be used as approximations of the discrete subdomain extrema, provided that the difference between the two satisfies the accuracy requirements. In Section 4.5 we study the maximal approximation error. If this error is too large it can be reduced by partial evaluation of the polynomial for

the values of a selection of the variables. For example, the polynomial

$$f(x,y) = -2x^2y^2 + 6x^2y - 2x^2 + 6xy^2 - 18xy + 6x - 2y^2 + 6y \quad (4.7)$$

over the domain $[0,9] \times [0,2]$ has (continuous-domain) extrema 139.5 and -108, while the extrema in the discrete subdomain $\mathbb{Z}^2 \cap [0,9] \times [0,2]$ are 112 and -108. Partially evaluating $f(x,y)$ for the possible values of y leads to 3 polynomials in one variable

$$\begin{aligned} f_0(x) &= f(x,0) = -2x^2 + 6x \\ f_1(x) &= f(x,1) = 2x^2 - 6x + 4 \\ f_2(x) &= f(x,2) = 2x^2 - 6x + 4 \ , \end{aligned}$$

with continuous-domain extrema $9/2$, -108, and twice 112 and $-1/2$, respectively. The continuous-domain extrema now coincide with the discrete-domain extrema.

Section 4.6 proposes three methods to convert quasi-polynomials into (sets of) polynomials so that bounds on quasi-polynomials can be derived. Experiments (Section 4.7) investigate the accuracy and computation cost of the different methods. By combining different methods hybrid methods can be constructed. This will be discussed in Section 4.8.

4.4 Related work: bounding the range of polynomials in continuous domains

An overview and comparison of methods to find bounds on polynomials over continuous domains can be found in [139, 122, 144].

Interval Arithmetic and its variants study the range of an expression as a function of the intervals in which the variables of the expression lie. The intervals are added, subtracted, multiplied, ... in such a way that the computed interval is guaranteed to contain the value of the expression for all values of the variables within the intervals. A problem is that the correlation between different terms is neglected. For example,

$$\begin{aligned} x \in [-1,1] \ &\Rightarrow \ (1-x) \in [0,2] \\ &\Rightarrow \ x + (1-x) \in [-1,1] + [0,2] = [-1,3] \ . \end{aligned}$$

However,

$$x + (1-x) = 1 \ ,$$

for any x. Similar problems occur in the evaluation of polynomials and can be addressed by not writing a polynomial using the power basis but using a Bernstein basis, Horner form or centered form [139].

Rivlin's method [142], extended to the multivariate case by Garloff [89], samples the polynomial in equidistant points and, based on the mean value theorem, derives a bound on the difference between the extrema of the samples and the extrema of the polynomial itself. For a given polynomial $f(x) = \sum_{i=0}^{g} a_i x^i$ of degree g the following holds on $x \in [0, 1]$:

$$\min_{0 \le i \le k} f\left(\frac{i}{k}\right) - \alpha_k \le f(x) \le \max_{0 \le i \le k} f\left(\frac{i}{k}\right) + \alpha_k \ , \tag{4.8}$$

where

$$\alpha_k = \frac{1}{8k^2} \sum_{i=0}^{g} i(i-1)|a_i| \ . \tag{4.9}$$

Note that (4.9) gives a measure for the maximal absolute error for a given polynomial.

Gopalsamy [92] uses a similar technique. He does not sample the polynomial in equidistant points but in well chosen points, only depending on the degree of the polynomial.

The Bernstein expansion [144, 89, 56] writes a polynomial as a linear combination of Bernstein basis polynomials. The maximal and minimal Bernstein coefficients provide bounds on the polynomial. This method, summarized in Appendix C, has the nice property that if the minimum or maximum is reached at a vertex of the domain, then that bound will be exact. For example, in Figure 4.2 the lower bound $\underline{F}_{\mathrm{b}}$ on f derived by Bernstein expansion is exact since the minimum of f over the interval $[0, 2]$ is reached for $x = 0$. Bernstein expansion can be used over a polytope domain [56, 107] and also over parameterized domains [56], in contrast with most other methods that need a box-shaped domain with unparameterized borders.

For most techniques (Rivlin, Bernstein) the approximation error can be reduced by dividing the domain into smaller parts and applying the technique on each subdomain. For example, in Figure 4.2, the upper bound $\overline{F}_{\mathrm{b}}$ on the polynomial function f found by Bernstein expansion over the interval $[0, 2]$ is

$$\overline{F}_{\mathrm{b}[0,2]} = 5 \ ,$$

while subdivision in two parts leads to

$$\max(\overline{F}_{\mathrm{b}[0,1]}, \overline{F}_{\mathrm{b}[1,2]}) = \max(3, 4) = 4 \ .$$

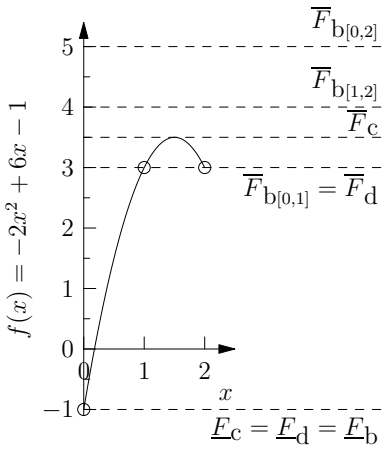


Figure 4.2: Example polynomial function with extrema in the discrete ({0,1,2}) and continuous ([0,2]) domain. The Bernstein expansion gives an upper bound on the maxima, which becomes more accurate when the interval is split into smaller parts.

Note that the upper bound in the interval $[0, 1]$ is exact just like the lower bound mentioned above. When computing the Bernstein coefficients in one subregion, intermediate values for the computation of the coefficients in neighboring subregions are produced. This allows to compute the Bernstein coefficients of a subdivision more economically, as demonstrated for a triangle in [107].

4.5 Continuous- versus discrete-domain extrema of polynomials

4.5.1 Univariate case

Consider an arbitrary polynomial $f(x) = \sum_{i=0}^{g} a_i x^i$ of degree g. We will use the notation $\overline{F}_{\mathrm{d}}$ and $\underline{F}_{\mathrm{d}}$ for the maximum and minimum of this polynomial in the discrete domain $\mathbb{Z} \cap [N_{min}, N_{max}]$, $N_{min}, N_{max} \in \mathbb{Z}$, and $\overline{F}_{\mathrm{c}}$ and $\underline{F}_{\mathrm{c}}$ for the extrema in the enclosing continuous interval $[N_{min}, N_{max}]$.[2] For $N_{min} = N_{max}$ these intervals obviously coincide. Therefore, we restrict ourselves to the case $N = N_{max} - N_{min} > 0$. Without loss of generality we will further assume $N_{min} = 0$ and $N_{max} = N$.

[2]Here F refers to the polynomial f. When another name is used for the polynomial, e.g., g or h this becomes $\overline{G}_{\mathrm{d}}$, $\underline{G}_{\mathrm{d}}$, $\overline{G}_{\mathrm{c}}$ and $\underline{G}_{\mathrm{c}}$, or $\overline{H}_{\mathrm{d}}$, $\underline{H}_{\mathrm{d}}$, $\overline{H}_{\mathrm{c}}$ and $\underline{H}_{\mathrm{c}}$, respectively.

An example is shown in Figure 4.2.

Definition 14. *If one uses the continuous-domain extrema of a polynomial f as an approximation of the discrete-domain extrema, the relative approximation error on the range, RE (Relative Error), is defined as*

$$RE = \begin{cases} \dfrac{(\overline{F}_{\mathrm{c}} - \underline{F}_{\mathrm{c}}) - (\overline{F}_{\mathrm{d}} - \underline{F}_{\mathrm{d}})}{\overline{F}_{\mathrm{d}} - \underline{F}_{\mathrm{d}}} & \textit{when } \overline{F}_{\mathrm{d}} \neq \underline{F}_{\mathrm{d}} \ ; \\ 0 & \textit{when } \overline{F}_{\mathrm{d}} = \underline{F}_{\mathrm{d}} \wedge \overline{F}_{\mathrm{c}} = \underline{F}_{\mathrm{c}} \ ; \\ \infty & \textit{when } \overline{F}_{\mathrm{d}} = \underline{F}_{\mathrm{d}} \wedge \overline{F}_{\mathrm{c}} \neq \underline{F}_{\mathrm{c}} \ . \end{cases} \tag{4.10}$$

For the example in Figure 4.2 we get $RE = \frac{0.5}{4} = 0.125$. Note that, by definition, $\overline{F}_{\mathrm{c}} \geq \overline{F}_{\mathrm{d}}$ and $\underline{F}_{\mathrm{c}} \leq \underline{F}_{\mathrm{d}}$. As a result, RE is always non-negative. Note also that it is scale and position invariant, i.e. the polynomial $af(x) + b$ with $a, b \in \mathbb{R}$ will yield the same value for RE as $f(x)$ does. In the cases $g = 0$ or $g = 1$, the polynomial is a linear function, with extrema in 0 and N, which results in $RE = 0$. For higher degrees we present the following theorems.

Theorem 3. *If $g \geq 2$ and $N \geq g$ then there exists an upper bound on the relative approximation error RE, caused by using the continuous-domain range as an approximation of the discrete-domain range, that can be reached for an arbitrary polynomial of degree g over the interval $[0, N]$. Otherwise, no such bound exists.*

Proof. If $N < g$ then a polynomial can be constructed with $RE = \infty$. For example, the polynomial $h(x) = \prod_{i=0}^{g-1}(x - i)$ has degree g, and for $n \in \{0, 1, \ldots, N\}$ it follows that $h(n) = 0 = \overline{H}_{\mathrm{d}} = \underline{H}_{\mathrm{d}}$, while $\overline{H}_{\mathrm{c}} \neq \underline{H}_{\mathrm{c}}$ and thus $RE = \infty$.

If $N \geq g > 0$ then $\overline{F}_{\mathrm{d}} \neq \underline{F}_{\mathrm{d}}$, for an arbitrary $f(x)$. Indeed, $\overline{F}_{\mathrm{d}} = \underline{F}_{\mathrm{d}} = f(n) = c$, for all $n \in \{0, 1, \ldots, N\}$ would imply that $f(x) - c$ is a polynomial of degree g with $N + 1 > g$ zeroes, which is impossible if $g > 0$.

In view of the invariance of RE, we can assume $\underline{F}_{\mathrm{d}} = -1$ and $\overline{F}_{\mathrm{d}} = 1$. The polynomial can be rewritten using Lagrange interpolants [61, 143] as

$$f(x) = \sum_{i=0}^{g} f(x_i) L_{S,i}(x) \ ,$$

where the $L_{S,i}(x)$ are degree-g Lagrange interpolants or basis functions, given by

$$\begin{aligned} L_{S,i}(x) &= \prod_{0\le j\le g, j\neq i} \frac{x-x_j}{x_i-x_j} \qquad (4.11) \\ &= \frac{(x-x_0)(x-x_1)\cdots(x-x_{i-1})(x-x_{i+1})\cdots(x-x_g)}{(x_i-x_0)(x_i-x_1)\cdots(x_i-x_{i-1})(x_i-x_{i+1})\cdots(x_i-x_g)} \end{aligned}$$

for $i \in \{0,1,\ldots,g\}$, with $S = \{x_0,\ldots,x_g\}$ an arbitrary subset of $\{0,1,\ldots,N\}$ with $g+1$ elements, denoted $S \in \mathcal{S}_N$, with

$$\mathcal{S}_N = \{S \subset \{0,1,\ldots,N\}, |S| = g+1\} \ . \qquad (4.12)$$

By construction,

$$L_{S,i}(x_j) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \ , \\ 0 & \text{if } i \neq j \ . \end{cases} \qquad (4.13)$$

By setting $\underline{F}_{\mathrm{d}} = -1$ and $\overline{F}_{\mathrm{d}} = 1$ we know that

$$|f(n)| \le 1, \forall n \in S \subset \{0,1,\ldots,N\}$$

and that for all $x \in [0,N]$ and for all $\binom{N+1}{g+1}$ possible choices of $S \in \mathcal{S}_N$,

$$\begin{aligned} |f(x)| &= \left|\sum_{i=0}^{g} f(x_i) L_{S,i}(x)\right| \\ &\le \sum_{i=0}^{g} |f(x_i)||L_{S,i}(x)| \\ &\le \sum_{i=0}^{g} |L_{S,i}(x)| \ . \end{aligned}$$

The sum of the absolute value of the Lagrange basis functions is termed *Lebesgue function* [150, 120, 143]. Since S is an arbitrary element of $\mathcal{S}_N$,

$$\begin{aligned} |f(x)| &\le \min_{S\in\mathcal{S}_N} \sum_{i=0}^{g} |L_{S,i}(x)| \triangleq M_N(x) \\ &\le \max_{x\in[0,N]} \left(\min_{S\in\mathcal{S}_N} \sum_{i=0}^{g} |L_{S,i}(x)| \right) \triangleq K \ , \qquad (4.14) \end{aligned}$$

where K is a function of N and g only. This results in

$$RE = \frac{(\overline{F}_{\mathrm{c}} - \underline{F}_{\mathrm{c}}) - 2}{2} \leq \frac{2K - 2}{2} = K - 1 \tag{4.15}$$

and thus an upper bound on the RE that can be reached for an arbitrary polynomial of the given degree g exists. □

If $g = 2$, the polynomial function f can only have one local extremum and therefore, either $\overline{F}_{\mathrm{c}} = \overline{F}_{\mathrm{d}} = 1$ or $\underline{F}_{\mathrm{c}} = \underline{F}_{\mathrm{d}} = -1$. As a result, the bound on RE in (4.15) can be refined to

$$RE \leq \frac{(K+1) - 2}{2} = \frac{K-1}{2} \ . \tag{4.16}$$

Definition 15. *We introduce $MRE(g, N)$ to denote the supremum of RE over all polynomials of degree g on an interval with size N. $MRE(g, N)$ can be a real number or infinity.*

If K is written as a function of g and N it follows from (4.15) that

$$MRE(g, N) \leq K(g, N) - 1 \ . \tag{4.17}$$

One can prove that $K(g, N) \geq K(g, N+1)$ and thus $K(g, N)$ is monotonically nonincreasing with N, for a given g. We will prove that $K(g, N)$ converges to 1 (and thus MRE converges to 0), as $N \to \infty$ (Theorem 4), but first present a lemma needed to prove the theorem.

Lemma 1. *The function $M_N(x) = \min_{S \in \mathcal{S}_N} \sum_i |L_{S,i}(x)|$ is symmetric about the axis $x = N/2$.*

Proof. We define linear transformations on sets as

$$kS + l = \{ku + l \mid u \in S\}, \text{ for } k, l \in \mathbb{R} \ ,$$

with the following property for the Lagrange interpolants:

$$L_{kS+l,i}(kx + l) = L_{S,i}(x) \ . \tag{4.18}$$

One can easily prove that

$$S \in \mathcal{S}_N \Leftrightarrow S' = ((-1)S + N) \in \mathcal{S}_N \ .$$

From (4.18) it follows that $L_{S',i}(x)$, with $S' = (-1)S + N$, is the mirror image of $L_{S,i}(x)$ about the axis $x = N/2$, and thus $\sum_i |L_{S',i}(x)|$ is the mirror image of $\sum_i |L_{S,i}(x)|$. This means that for each function $\sum_i |L_{S,i}(x)|$ that is included in the minimization over $\mathcal{S}_N$, also the mirror image about $x = N/2$ is included. As a result, the obtained function, $M_N(x)$ is symmetric about $x = N/2$. □

Theorem 4. *For a fixed degree $g \geq 2$, $MRE(g, N)$, and thus the maximal RE that can be reached by an arbitrary polynomial, converges to 0 as N increases indefinitely.*

Proof. A similar notation is used as in the previous theorem. An index N is used to indicate the dependence on N.

$$K_N \quad = \quad \max_{x \in [0,N]} M_N(x)$$

It suffices to show that K_N converges to 1. This follows from (4.17). Convergence of K_N requires that for an arbitrary $\epsilon > 0$ there exists an N_m for which $1 - \epsilon < K_{N'} < 1 + \epsilon$, for all $N' \geq N_m$. Note that by construction $1 \leq K_{N'}$ and thus $1 - \epsilon < K_{N'}$ for any $\epsilon > 0$.

If $x_j \in S$ then

$$\sum_i |L_{S,i}(x_j)| = \sum_i \delta_{ij} = 1 \ ,$$

using (4.13) and thus

$$M_N(n) = 1, \forall n \in \{0, 1, \ldots, N\}, \forall N \geq g \ .$$

Choose an arbitrary value of N, e.g., $N_g = g$. Since $M_{N_g}(x)$ is a continuous function and since $M_{N_g}(0) = 1$,

$$\exists\, b \in (0, 1] \text{ such that } M_{N_g}(x) < 1 + \epsilon, \text{ for } x \in [0, b] \ .$$

Since $M_{N_g}(x)$ is symmetric about $x = \frac{N_g}{2}$, it follows that $M_{N_g}(x) < 1+\epsilon$ for $x \in [N_g - b, N_g]$. Let

$$a = \left\lceil \frac{1}{b} \right\rceil, \text{ i.e. } a \in \mathbb{N} \text{ and } ab \geq 1 \ ,$$

then

$$M_{N_g}\left(\frac{x}{a}\right) < 1 + \epsilon \ , \qquad \forall x \in [0, ab] \cup [aN_g - ab, aN_g] \ ,$$

hence also, since $ab \geq 1$,

$$M_{N_g}\left(\frac{x}{a}\right) < 1 + \epsilon \ , \qquad \forall x \in [0, 1] \cup [aN_g - 1, aN_g] \ . \tag{4.19}$$

Choose

$$N' \geq 2aN_g - 2 = N_m, \ N' \in \mathbb{N} \ , \tag{4.20}$$

and create $\mathcal{S}_{N',j} \subset \mathcal{S}_{N'}$ from $\mathcal{S}_{N_g}$ by transforming the elements in $\mathcal{S}_{N_g}$ as follows

$$\mathcal{S}_{N',j} = \{aS + j \mid S \in \mathcal{S}_{N_g}\},\ j = 0, 1, \ldots, N' - aN_g \ .$$

Using (4.18) and (4.19) this results in

$$\begin{aligned}
\min_{S \in \mathcal{S}_{N',j}} \sum_i |L_{S,i}(x)| &= \min_{S \in \mathcal{S}_{N_g}} \sum_i |L_{aS+j,i}(x)| \\
&= \min_{S \in \mathcal{S}_{N_g}} \sum_i \left| L_{S,i}\left(\frac{x-j}{a}\right)\right| \\
&= M_{N_g}\left(\frac{x-j}{a}\right) \\
&< 1 + \epsilon, \text{ when } x \in [j, j+1] \text{ or} \\
&\quad\ x \in [aN_g - 1 + j, aN_g + j] \ .
\end{aligned}$$

Knowing that $\cup_j \mathcal{S}_{N',j} \subset \mathcal{S}_{N'}$, and using (4.20)

$$\begin{aligned}
M_{N'}(x) &= \min_{S \in \mathcal{S}_{N'}} \sum_i |L_{S,i}(x)| \\
&\leq \min_{S \in \cup_j \mathcal{S}_{N',j}} \sum_i |L_{S,i}(x)| \\
&< 1 + \epsilon \ ,
\end{aligned}$$

when

$$x \in \bigcup_{j=0}^{N'-aN_g} [j, j+1] \cup [aN_g - 1 + j, aN_g + j] = [0, N'] \ .$$

Hence, $K_{N'} < 1 + \epsilon$, and the theorem follows. □

The bound (4.14) obtained in the proof of Theorem 3, may be an overestimate of the real bound. For $g = 2$ and $g = 3$ an exact bound can be obtained. With the above notation we have the following theorem:

Theorem 5. *For any polynomial of degree 2, the relative error (RE) in the interval* $[0, N]$, $N \geq 2$, *satisfies*

$$RE \leq \begin{cases} \frac{1}{(N+1)^2-1} & \textit{if } N \textit{ is even} \\ \frac{1}{N^2-1} & \textit{if } N \textit{ is odd} \ . \end{cases}$$

Proof. Let $f(x) = ax^2+bx+c$. Without loss of generality we can assume $c = 0$ (vertical shift does not alter $\overline{F} - \underline{F}$) and $a < 0$.

If the maximum of f over $\mathbb{R}$ ($x_t = \frac{-b}{2a}$, $f(x_t) = \frac{-b^2}{4a}$) lies outside $[0, N]$ then $RE = 0$. Therefore, we assume

$$0 \leq \frac{-b}{2a} = x_t \leq N$$

and even

$$\frac{N}{2} \leq \frac{-b}{2a} = x_t \leq N \ ,$$

as mirroring horizontally about $x = \frac{N}{2}$ and shifting vertically does not alter RE. This implies

$$\underline{F}_{\mathrm{c}} = \underline{F}_{\mathrm{d}} = 0, \overline{F}_{\mathrm{c}} = \frac{-b^2}{4a}$$
$$RE = \frac{\overline{F}_{\mathrm{c}} - \overline{F}_{\mathrm{d}}}{\overline{F}_{\mathrm{d}}} \ .$$

Now, RE is maximal for a given $\overline{F}_{\mathrm{c}}$ if $\overline{F}_{\mathrm{d}}$ is minimal. Varying $\overline{F}_{\mathrm{c}}$ by vertically scaling $f(x)$ does not alter RE. Let $\frac{-b}{2a} = x_t$ vary between $\frac{N}{2}$ and N while keeping $\overline{F}_{\mathrm{c}} = \frac{-b^2}{4a}$ constant and $f(0) = 0$. This is done by horizontal scaling. Then

$$\overline{F}_{\mathrm{d}} = \max(f(\lfloor x_t \rfloor), f(\lceil x_t \rceil))$$

reaches a local minimum when

$$f(\lfloor x_t \rfloor) = f(\lceil x_t \rceil)$$

or, thanks to the symmetry of $f(x)$ about the axis $x = x_t$,

$$\{x_t\} = \frac{1}{2} \ .$$

As $f(x)$ is more flat for lower values of $|a|$ or higher values of x_t the globally minimal $\overline{F}_{\mathrm{d}}$ and maximal RE are found for the smallest x_t that satisfies:

$$\{x_t\} = \frac{1}{2} \text{ and } \frac{N}{2} \leq x_t \leq N \ .$$

Thus

$$x_t = \frac{N+1}{2}, \ \overline{F}_{\mathrm{d}} = f\left(\frac{N+2}{2}\right), \ RE = \frac{1}{(N+1)^2 - 1} \ ,$$

Table 4.1: Overview of MRE and $K-1$ for several values of N and g. Figures rounded to 5 decimals. The values of $K-1$ are plotted in Figure 4.3.

(a) $MRE(g, N)$

N	g = 2	g = 3
2	0.12500	∞
3	0.12500	0.50343
4	0.04167	0.18808
5	0.04167	0.18808
6	0.02083	0.18808
7	0.02083	0.10017
8	0.01250	0.06257
9	0.01250	0.06257
10	0.00833	0.06257
11	0.00833	0.04289
12	0.00595	0.03127
13	0.00595	0.03127
14	0.00446	0.03127
15	0.00446	0.02382
16	0.00347	0.01876
17	0.00347	0.01876
18	0.00278	0.01876
19	0.00278	0.01516

(b) $K(g, N) - 1$

N	g = 3	g = 4	g = 5	g = 6
2	-	-	-	-
3	0.63113	-	-	-
4	0.29314	1.20782	-	-
5	0.18808	0.71234	2.10630	-
6	0.18808	0.41106	1.18573	3.54934
7	0.11482	0.29165	0.87606	2.10933
8	0.08137	0.22147	0.59396	1.33818
9	0.06257	0.17014	0.43558	1.05870
10	0.06257	0.17014	0.34837	0.85179
11	0.04726	0.13640	0.28921	0.65248
12	0.03772	0.10781	0.23577	0.50811
13	0.03127	0.08985	0.19622	0.42642
14	0.03127	0.07590	0.16961	0.36470
15	0.02568	0.06490	0.16358	0.31936
16	0.02171	0.05714	0.14893	0.27609
17	0.01876	0.05714	0.12627	0.23998
18	0.01876	0.04906	0.11088	0.21203
19	0.01612	0.04332	0.09901	0.18901

when N is even and

$$x_t = \frac{N}{2},\ \overline{F}_{\mathrm{d}} = f\left(\frac{N+1}{2}\right),\ RE = \frac{1}{N^2-1}\ ,$$

otherwise. □

One can prove that this bound equals $\frac{K-1}{2}$. Since the proof of Theorem 5 provides values of x_t for which this bound is reached, the bound is an exact bound.

For $g = 3$ the maximum REs can be expressed in a more complex way. Guidelines for the proof are found in Appendix D. Here only the maximal RE will be given together with polynomials of the form $f(x) = x^3 + bx^2 + c$ with corresponding values of N where the MRE is reached.

We have the following cases:

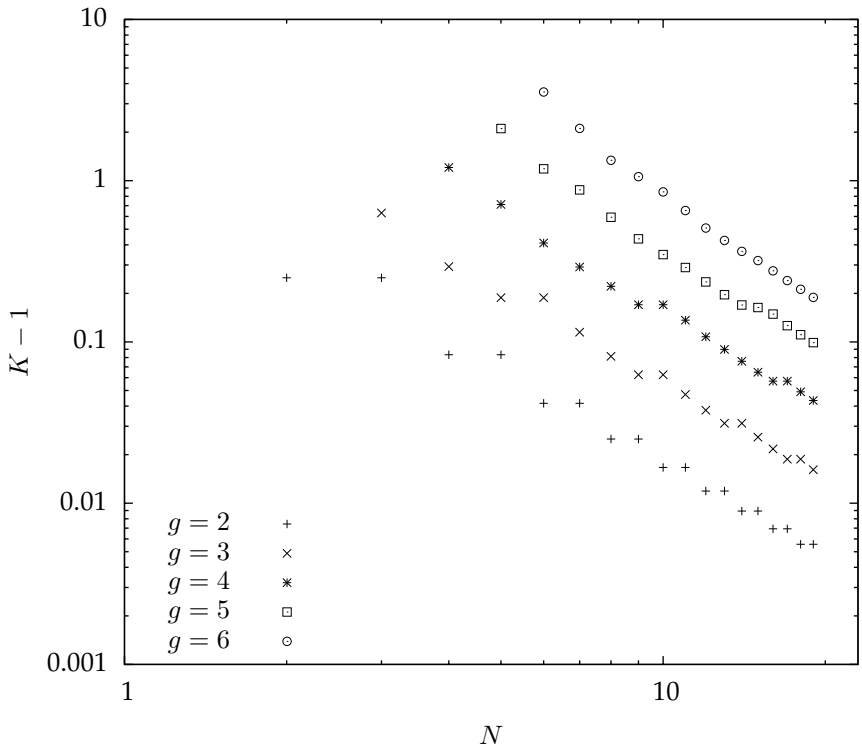


Figure 4.3: $K-1$ as a function of N for several degrees g (Table 4.1(b)).

- Case $N = 4k-1$, $k \in \mathbb{N}$:

$$\begin{aligned} MRE &= \frac{1}{18}\frac{\sqrt{36\,k^2+3}\left(12\,k^2+1\right)}{4\,k^3-k} - 1 \\ &\approx \frac{3}{8k^2}, \quad \text{for large } k \end{aligned}$$

reached by polynomials that satisfy

$$\begin{aligned} b &= -\frac{3}{2}N \\ c &= \frac{9}{16}N^2 - \frac{3}{8}N - \frac{7}{16} \ . \end{aligned}$$

- Case $4k \leq N \leq 4k+2$, $k \in \mathbb{N}$:

$$\begin{aligned} MRE &= \frac{2}{9}\frac{\sqrt{9\,k^2+9\,k+3}\,\left(3\,k^2+3\,k+1\right)}{k\,(2\,k^2+3\,k+1)} - 1 \\ &\approx \frac{1}{3k^2}, \quad \text{for large } k \end{aligned}$$

reached by polynomials that satisfy

$$\begin{aligned} b &= -6l-3-3i \\ c &= 9l^2+(12i+3)l+3i^2+6i-4 \ , \end{aligned}$$

$$\begin{array}{llll} \text{with} & l \in \mathbb{N}_0, & i \in \{0,1,2\}, & 4l+4 \leq N \leq 4l+4+i \\ \text{or} & l = 0, & i \in \{1,2\}, & 3+i \leq N \leq 4+i \ . \end{array}$$

Apparently, for $N \leq 19$, $K(3,N)-1 = MRE(3,N)$ if $N = 4k+1$ or $N = 4k+2$ (Table 4.1 (a) and (b)). This means that for these values of N the bound on RE defined in (4.15) is an exact bound. We did already mention that for $g = 2$ the bound given in (4.16) is exact for all N. The values of $K-1$ listed in Table 4.1(b) are plotted in Figure 4.3.

We conclude that for *large* intervals (large N) the continuous extrema are a good approximation of the discrete extrema and the methods described in Section 4.4 can be used. For *small* intervals it is better to evaluate the polynomial in discrete points, but then the evaluation cost is small. What is *large* or *small* obviously depends on the degree of the polynomial.

4.5.2 Multivariate case

Theorem 3 can be extended to the multivariate case over box-shaped domains. Here only the case of two variables is considered, but it can easily be generalized to more variables.

Theorem 6. *If $N_x < g_x$ ($g_x \geq 2$) or $N_y < g_y$ ($g_y \geq 2$) then the RE of an arbitrary polynomial $f(x,y) = \sum_{i=0}^{g_x}\sum_{j=0}^{g_y} a_{ij}x^iy^j$ over $[0,N_x] \times [0,N_y]$ is not bounded. If $N_x \geq g_x$ and $N_y \geq g_y$ then RE is bounded.*

Proof. The scheme of the proof is analogous to the proof of Theorem 3. A polynomial $f(x,y)$ can be written as a linear combination of Lagrange interpolants:

$$f(x,y) = \sum_{i=0}^{g_x}\sum_{j=0}^{g_y} f(x_i,y_i)L_{S_x,i}(x)L_{S_y,j}(y) \ ,$$

with $S_x \in \mathcal{S}_{g_x,N_x}$ and $S_y \in \mathcal{S}_{g_y,N_y}$, sets of cardinality g_x+1 and g_y+1, respectively. Without loss of generality we can assume $\underline{F}_{\mathrm{d}} = -1$ and $\overline{F}_{\mathrm{d}} = 1$ and thus

$$|f(n_x,n_y)| \leq 1, \forall (n_x,n_y) \in S_x \times S_y \subset \{0,1,\ldots,N_x\} \times \{0,1,\ldots,N_y\} \ .$$

For all $(x,y) \in [0,N_x] \times [0,N_y]$ and for all possible S_x and S_y we have

$$\begin{aligned}
|f(x,y)| &\leq \sum_{i=0}^{g_x}\sum_{j=0}^{g_y} |f(x_i,y_i)||L_{S_x,i}(x)||L_{S_y,j}(y)| \\
&\leq \max_{\substack{x\in[0,N_x]\\ y\in[0,N_y]}} \left(\min_{\mathcal{S}_{g_x,N_x}} \min_{\mathcal{S}_{g_y,N_y}} \sum_{i=0}^{g_x}\sum_{j=0}^{g_y} |L_{S_x,i}(x)||L_{S_y,j}(y)| \right) \\
&\leq \max_{\substack{x\in[0,N_x]\\ y\in[0,N_y]}} \left(\min_{\mathcal{S}_{g_x,N_x}} \sum_{i=0}^{g_x} |L_{S_x,i}(x)| \right) \left(\min_{\mathcal{S}_{g_y,N_y}} \sum_{j=0}^{g_y} |L_{S_y,j}(y)| \right) \\
&\leq K(g_x,N_x)K(g_y,N_y) \triangleq K_{xy}(g_x,N_x,g_y,N_y) \ .
\end{aligned}$$

The bound on RE now becomes

$$RE \leq K_{xy}(g_x,N_x,g_y,N_y) - 1 = K(g_x,N_x)K(g_y,N_y) - 1 \ . \tag{4.21}$$

□

The convergence of K_{xy} follows from the convergence of $K(g_x,N_x)$ and $K(g_y,N_y)$. Note that $MRE(g_x,N_x,g_y,N_y)+1$ may differ from $(MRE(g_x,N_x)+1)(MRE(g_y,N_y)+1)$, e.g., for $g_x = g_y = 2$ and $N_x = N_y = 2$ we have $(MRE(g_x,N_x)+1)(MRE(g_y,N_y)+1) = (9/8)^2 = 81/64 = 1.265625$ (Theorem 5), while on this domain the polynomial $f(x,y)$ in (4.7) has $RE = 13/32 = 0.40625$ and thus $RE+1 = 45/32 = 90/64 > 81/64$.

4.5.3 Optimization strategy

The bounds on RE lead to the following approximation heuristic:

Examine the degrees g_i and ranges N_i of the variables of the given polynomial. Calculate the upper bound on the RE, i.e.

$$\prod_i K(g_i,N_i) - 1 \ .$$

Table 4.2: REs of some 2-dimensional examples.

(a) $N_x = 2$, $N_y = 2$

	$\mathsf{RE_{xy}}$	$\mathsf{MRE(g_x, N_x)}$	$\mathsf{RE_x}$	$\mathsf{MRE(g_y, N_y)}$	$\mathsf{RE_y}$
f(x, y)	13/32=0.40625	1/8	1/4	1/8	1/4
g(x, y)	17/64=0.265625	1/8	1/8	1/8	1/8
h(x, y)	1/8=0.125	1/8	1/8	0	0

(b) $N_x = 9$, $N_y = 2$

	$\mathsf{RE_{xy}}$	$\mathsf{MRE(g_x, N_x)}$	$\mathsf{RE_x}$	$\mathsf{MRE(g_y, N_y)}$	$\mathsf{RE_y}$
f(x, y)	0.125	1/80=0.0125	0	1/8	1/8
g(x, y)	0.130	1/80=0.0125	0.00446	1/8	1/8
h(x, y)	0.00446	1/80=0.0125	0.00446	0	0

If this number does not satisfy the accuracy requirements select the variable(s) with the largest K and partially evaluate the polynomial for the discrete values of this(these) variable(s), resulting in a set of polynomials and corresponding domains. The variables with the highest impact on the RE will typically have a relatively low value of N_i and thus a low cost of partial evaluation.

Example 7. *Consider the polynomials*

$$\begin{aligned} f(x,y) &= -2x^2y^2 + 6x^2y - 2x^2 + 6xy^2 - 18xy + 6x - 2y^2 + 6y \\ g(x,y) &= 4x^2y^2 - 12x^2y - 12xy^2 + 36xy \\ h(x,y) &= -2x^2y + 6xy \end{aligned}$$

over the domain $[0, N_x] \times [0, N_y]$. *The REs* (RE_{xy}) *over this domain for* $N_y = 2$ *and two values of* N_x *are listed in Table 4.2. The polynomials can be partially evaluated for the discrete values of one variable, as demonstrated on* f *in Section 4.3.*

Filling in the values of y *leads to functions in* x*, which result in the listed* RE_x*s, and filling in the values of* x *leads to the* RE_y*s. The indices of* RE *indicate the variables that were kept* continuous *when searching the extrema.*

Note that for f *and* $N_x = 2$ *we find*

$$RE_x = RE_y = \frac{1}{4} > MRE(g_x, N_x) = MRE(g_y, N_y) = \frac{1}{8} \ .$$

This is possible because $\overline{F}_c$ *and* $\underline{F}_c$ *are found for a different value of* y (RE_x) *or* x (RE_y).

We see that it is more beneficial to evaluate variables with a small range and/or high degree than other variables. The values of the several $MRE(g_i, N_i)$ give an indication on this.

4.5.4 Comparison with Rivlin's method

Equations (4.8) and (4.9) also give a bound on the difference between discrete- and continuous-domain extrema of polynomials. After horizontal scaling a bound on the RE of a polynomial over the domain $[0, N]$ can be derived. Finding the discrete- and continuous-domain extrema of $f(x) = \sum_{i=0}^{g} a_i x^i$ for $x \in [0, N]$ is equivalent to finding the extrema of $g(x) = f(Nx)$ for $x \in [0, 1]$, with sampling points $x_j = \frac{j}{N}$, $j \in \{0, \ldots, N\}$.

$$\begin{aligned} g(x) &= f(Nx) = \sum_{i=0}^{g} a_i N^i x^i \\ \alpha_N &= \frac{1}{8N^2} \sum_{i=1}^{g} i(i-1)\,|a_i|\,N^i \\ RE_{rivl} &= \frac{2\alpha_N}{\overline{F}_{\mathrm{d}} - \underline{F}_{\mathrm{d}}} \leq \frac{2\alpha_N}{\overline{F}_{\mathrm{c}} - \underline{F}_{\mathrm{c}} - 2\alpha_N} \quad . \end{aligned}$$

For $f(x) = x^3 + bx^2 + cx$ this becomes

$$\begin{aligned} \alpha_N &= \frac{1}{8N^2}(2\,|b|\,N^2 + 6N^3) \\ &= \frac{1}{4}(|b| + 3N) \quad . \end{aligned}$$

For the polynomials with maximal RE listed in Section 4.5.1 this results in a large overestimate of the RE as shown in Table 4.3. However, for many polynomials with $RE < MRE$, RE_{rivl} will be closer to the real RE than MRE as it uses polynomial specific information. In general one can use:

$$RE \leq \min(RE_{rivl}, MRE) \quad .$$

with MRE depending on g and N and RE_{rivl} depending on N and the polynomial coefficients (but the last two).

The method of Rivlin provides a bound on the absolute difference between continuous- and discrete-domain extrema for a given polynomial, which might be a large overestimate as indicated above. The

Table 4.3: Comparison of the exact RE (rounded to 4 decimals) with the approximation RE_{rivl} obtained by using Rivlin's method for some polynomials of the form $x^3 + bx^2 + cx$ with maximal RE.

N	b	c	RE_{rivl}	RE	$\frac{RE_{rivl}}{RE}$
3	-4.5	3.5	2.25	0.5034	4.4694
4	-6	5	0.75	0.1881	3.9878
5	-6	5	0.875	0.1881	4.6524
5	-9	20	1	0.1881	5.3170
6	-9	20	1.125	0.1881	5.9817
7	-10.5	24.5	0.525	0.1002	5.2409

relative error can only be computed after estimation of the extrema. On the other hand, we provide a bound on the relative difference for an arbitrary (worst-case) polynomial, which is a large overestimate of the RE when $RE \ll MRE$ (best-case polynomial), but can be used without knowledge of the polynomial.

In practical cases a larger absolute error can be tolerated on large numbers than on small numbers. This indicates that a bound on the relative error is a better measure than a bound on the absolute error.

4.6 Converting quasi-polynomials into polynomials

4.6.1 Concept

The previous section describes the case in which the continuous-domain extrema can be used as an approximation of the discrete-domain extrema of a polynomial. By converting quasi-polynomials into polynomials these results can be used to find bounds on quasi-polynomials.

For simplicity, we will only present the techniques for quasi-polynomials in one variable in a single domain, i.e. an interval. Our techniques aim at transforming a quasi-polynomial into one or more polynomials, possibly with additional variables. These arise from the elim-

Table 4.4: General overview of methods to eliminate fractional expressions.

Method	New Var	Substitute $\{\frac{an-r}{d}\}$ with	Constraints	Subdomains with constant
(a) Add Var	q	$\frac{q}{d}$	$0 \leq q \leq d-1$	
(b) Mod Classes	q	$\frac{q}{d}$	$0 \leq q \leq d-1$, $q \in \mathbb{Z}$, adjust range n	q
(c) Add Var	k	$\frac{an-r-kd}{d}$	$0 \leq an-r-kd \leq d-1$	
(d) Split Periods	k	$\frac{an-r-kd}{d}$	$0 \leq an-r-kd \leq d-1$, $k \in \mathbb{Z}$	k
(e) Exact		value of $\{\frac{an-r}{d}\}$		n

ination of the fractional expressions of the form[3] $\{\frac{an-r}{d}\}$ in the quasi-polynomial, where n is the free integer variable and a, d and r are integer constants ($d > 0$). In the discrete points of the domain the obtained polynomials will have the same value as the original quasi-polynomial in the corresponding points of its domain.

To eliminate the fractional expression $\{\frac{an-r}{d}\}$, consider the following. For given d, an integer number $an - r$ can be written in a unique way as

$$an - r = kd + q \ , \tag{4.22}$$

$$\text{with} \qquad k \in \mathbb{Z},\ q \in \mathbb{Z} \cap [0, d-1] \ . \tag{4.23}$$

This corresponds to the definition of modulo reduction and integer division:

$$q = (an - r) \bmod d \tag{4.24}$$

$$k = \left\lfloor \frac{an-r}{d} \right\rfloor \ . \tag{4.25}$$

[3]This form is used in (4.5) (with $a = 1$) and thus does not result in a loss of generality.

Hence we can write

$$\left\{\frac{an-r}{d}\right\} = \frac{q}{d} \tag{4.26}$$

$$= \frac{an-r}{d} - \left\lfloor\frac{an-r}{d}\right\rfloor = \frac{an-r}{d} - k \ . \tag{4.27}$$

We eliminate the fractional expression $\{\frac{an-r}{d}\}$ by replacing it by simple arithmetic expressions containing an additional variable q or k. Several possibilities how to do this are shown in Table 4.4. In the polynomials so obtained, the variables (n, q) or (n, k) are constrained by the inequalities

$$0 \leq q \leq d-1 \tag{4.28}$$

or

$$0 \leq an - r - kd \leq d-1 \ . \tag{4.29}$$

We are looking for the extremal values of this polynomial for all discrete values in the domain defined by the range of n and (4.28) or (4.29), respectively. These extremal values can be approximated by the extremal values over the continuous domain defined by the range of n and (4.28) or (4.29) in which the variables (n, q) or (n, k) are real-valued. As shown in Section 4.5, more accurate results can be obtained by restricting one of the variables (e.g., q or k) to integer values, leading to different polynomials on different (real) subdomains.

4.6.2 Overview of methods with examples

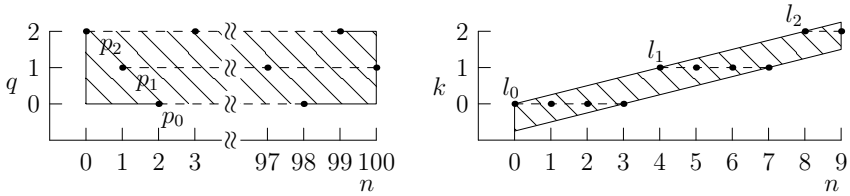
The simplest method to eliminate a fractional expression is to replace it with q/d (inspired by (4.26)), where q is a free variable within the (real) range $[0, d-1]$ (*Add Var*, Table 4.4(a)). The correlation between n and the fractional is then ignored (q is not bound to n by an equation like (4.24) and can even have non-integer values). More accuracy is obtained by splitting the domain into d subdomains with a constant integer value of q $(0, 1, \ldots, d-1)$ (*Mod Classes*, Table 4.4(b)). In each subdomain the (continuous) range of n is adjusted to the value of q (dashed lines in Figure 4.4(a)) :

$$n \in [\min D_q, \max D_q] \ ,$$

with,

$$D_q = \{m \in [0, N] : \ (am - r) \equiv q \pmod{d}\} \ .$$

D_q is the intersection of the domain of n with the residue class corresponding to the value of q.



(a) $n \in [0, 100]$, $d = 3$, $a = 2$, $r = 1$ (b) $n \in [0, 9]$, $d = 4$, $a = 1$, $r = 0$

Figure 4.4: Domains represented in the (n, q) and (n, k) space. The dots correspond with the (exact) discrete domain. The shaded areas represent the continuous domains of *Add Var*. The dashed lines indicate the subdomains obtained by *Mod Classes* (a) and *Split Periods* (b), respectively.

Example 8. *(small period, large domain)*

$$f(n) = n + [4, 2, 0]_n = n + 6\left\{\frac{2n-1}{3}\right\} \qquad 0 \le n \le 100$$

By replacing $\left\{\frac{2n-1}{3}\right\}$ *with* $\frac{q}{3}$ *this quasi-polynomial can be converted into a polynomial:*

$$g(n, q) = n + 2q \qquad 0 \le n \le 100,\ 0 \le q \le 2\ .$$

Note that

$$f(n) = g(n, (2n-1) \bmod 3)\ .$$

The dots in Figure 4.4(a) have coordinates that satisfy $(n, q) = (n, (2n - 1) \bmod 3)$ *and thus correspond to points of the quasi-polynomial* f*. As a result,* $\overline{F}_{\mathrm{d}} \le \overline{G}_{\mathrm{d}} \le \overline{G}_{\mathrm{c}}$*. The continuous-domain maximum of* g*,* $\overline{G}_{\mathrm{c}} = 104$*, is thus an upper bound on* $\overline{F}_{\mathrm{d}}$*.*

The polynomial g *can be split into 3 polynomials, with corresponding domains (using modulo classes) (Figure 4.4(a)):*

$$\begin{array}{ll} p_0(n) = n & 2 \le n \le 98 \\ p_1(n) = n + 2 & 1 \le n \le 100 \\ p_2(n) = n + 4 & 0 \le n \le 99\ . \end{array}$$

The continuous-domain maximum now coincides with the discrete-domain maximum with a value of 103*. This is also the maximum of the original quasi-polynomial* f*. Note that not all discrete points in the domains of the* p_i *correspond with a discrete point in the domain of* f*. This can be solved by horizontally scaling the domain with a factor* $1/3$ *(possibly after a translation), but does not matter if the continuous-domain extrema are used.*

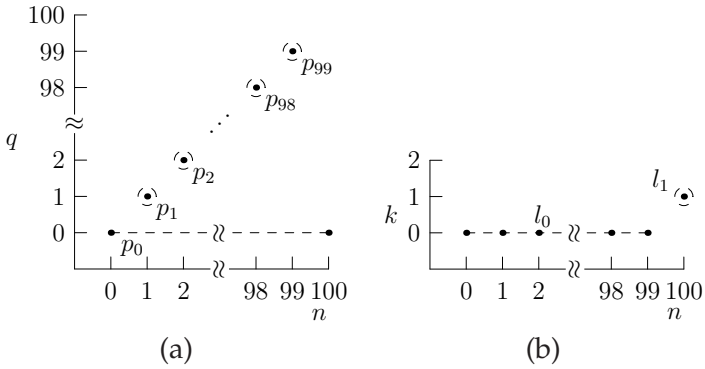


Figure 4.5: Domains of example 9 represented in the (n,q) and (n,k) space. The dots correspond with the (exact) discrete domain. The dashed lines indicate the continuous subdomains. A dashed circle corresponds to a single point.

When d is *large* splitting into modulo classes leads to many domains (high cost) each with only a few points of the exact discrete domain (a distance d apart), probably with a large approximation error[4]. Instead of introducing a variable q one can introduce a variable k and replace the fractionals with (4.27) (*Add Var*, Table 4.4(c)). It can be seen that both *Add Var* methods are equivalent, by using $q = an - r - kd$. The accuracy is now improved by splitting the domain into subdomains with constant integer k (*Split Periods*, Table 4.4(d), Figure 4.4(b)).

Example 9. *(large period, small domain)*

$$f(n) = \frac{51}{50}n - \frac{n^2}{100} + 2\left\{\frac{n}{100}\right\}n - 2\left\{\frac{n}{100}\right\} - 200\left\{\frac{n}{100}\right\}^2 \quad 0 \le n \le 100$$

In the same way as in the previous example we can define

$$p_i = \frac{51}{50}n - \frac{n^2}{100} + \frac{ni}{50} - \frac{i}{50} - \frac{i^2}{50} \quad .$$

This leads to 99 *domains with a single element* ($i \in [1, 99]$) *and the domain* $n \in [0, 100]$ *for* $i = 0$ *(Figure 4.5(a)). The computational effort is roughly the same as evaluating the function in all integer points.*

Fortunately, the fractional expression can be eliminated in a better way by the substitution

$$\left\{\frac{n}{100}\right\} = \frac{n - 100k}{100} \quad \textit{with} \quad 0 \le \frac{n - 100k}{100} \le \frac{99}{100} \quad .$$

[4]Scaling with a factor $1/d$ allows to use the results of Section 4.5.

Table 4.5: Maxima using different methods on several examples.

	Ex. 8	Ex. 9	Ex. 10	Ex. 11
Exact	103	25	2	98020000
Modulo Classes	103	26	12	98020000
Split Periods	103	25	20	98020000
Add Var	104	51	120	99020000

The domain of n is split into parts with constant integer k, called periods. *This results in*

$$f(n) = \begin{cases} l_0(n) = n - \frac{n^2}{100} & \text{if } 0 \leq n \leq 99 \\ l_1(n) = -\frac{n^2}{100} + 3n - 198 = 2 & \text{if } n = 100 \end{cases} .$$

Figure 4.5(b) shows the domains in the (k,n) space.

In example 8 this method would result in a lot of overhead. 68 domains ($[0], [1], [2,3], [4], [5,6], \ldots, [97], [98,99], [100]$ for $k = -1, 0, 1, 2, 3, \ldots, 64, 65, 66$) with a small size (1 or 2). This becomes better after the substitution $\left\{\frac{2n-1}{3}\right\} = 1 - \left\{\frac{n-2}{3}\right\}$, by which an expression with $a = 1$ can be obtained.[5] *Still this method would lead to 34 subdomains ($[0,1], [2,4], \ldots, [98,100]$ for $k = -1, 0, \ldots, 32$) with a size of 2 or 3. Remember that small subdomains may lead to large approximation errors when the continuous-domain extrema are used (cf. Section 4.5). However, the error can not be larger than in the case no evaluation for the discrete values of k is done.*

Exact (Table 4.4(e)) evaluates the quasi-polynomial in every point of the discrete domain.

Example 10. *(small period, small domain)*

$$\begin{aligned} f(n) = \; & 1440\left\{\frac{n}{2}\right\}^2 n^2 - 5760\left\{\frac{n}{2}\right\}^3 n + 5760\left\{\frac{n}{2}\right\}^4 + 2880\left\{\frac{n}{2}\right\}^3 \\ & -3208\left\{\frac{n}{2}\right\}^2 - 720\left\{\frac{n}{2}\right\} n^2 + 1480\left\{\frac{n}{2}\right\} n \\ & +104\left\{\frac{n}{2}\right\} - 10n^2 + 20n + 2 \qquad 0 \leq n \leq 2 \end{aligned}$$

Evaluation in discrete points ($f(0) = f(1) = f(2) = 2 = \overline{F}_{\mathrm{d}}$), gives the exact extrema with little cost. The other methods lead to bad results (Table 4.5).

[5] A smaller value of a always results in less domains. One can always make $a \leq \frac{d}{2}$.

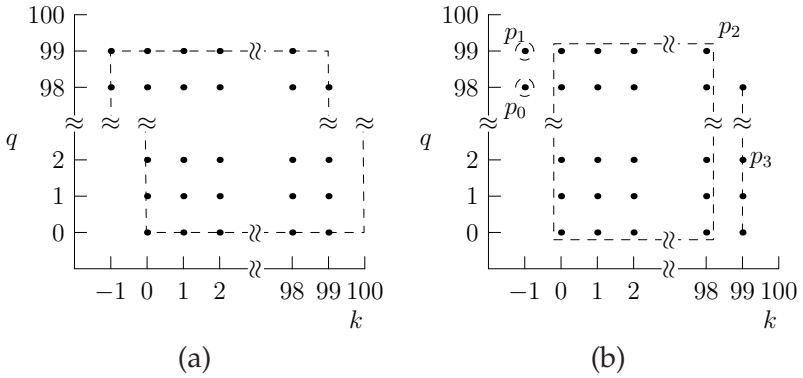


Figure 4.6: Domain (a parallelogram) of example 11 represented in the (k,q)-space with $n = 100k + q + 2$, $0 \leq q \leq 99$. In (b) the domain is split into a rectangle, one line segment and two points. Note that only for points with integer values for both k and q, we can say $k = \lfloor \frac{n-2}{100} \rfloor$, $q = n - 2 \bmod 100$.

Example 11. *(large period, large domain)*

$$f(n) = n^2 \left\{ \frac{n-2}{100} \right\} + 2n \qquad 0 \leq n \leq 10000$$

Mod Classes *and* Split Periods *both find the exact maximum* $\overline{F}_{\mathrm{d}} = 98020000 = f(10000)$, *by splitting the domain into* 100 *modulo classes or* 101 *periods, respectively.* Add Var *makes a little error of 1%,* $\overline{F}_{\mathrm{c}} = 99020000$.

The domain is shown in Figure 4.6(a), using the (k,q)-space ($q = n - r - kd$) *for reasons of compactness. It is bounded by the inequalities*

$$0 \leq q \leq 99,\ 0 \leq 100k + q + 2 \leq 10000 \ .$$

This is a parallelogram, which may not be apparent from the figure due to the broken axes, with vertices (-0.02,0), (99.98,0), (-1.01,99) and (98.99,99).

The introduction of k *makes it possible to increase the accuracy by evaluating* k *in discrete values, but it is not necessary to divide the domain into* 101 *parts. In this example it suffices to split the domain into the main rectangle, a line segment and two points at the borders (Figure 4.6(b)). By doing this, the region on the top right, which caused the overestimate, is removed and we obtain* $\overline{F}_{\mathrm{c}} = \overline{F}_{\mathrm{d}}$. *This can be done in the* (n, k)*-space but this would be harder to visualize here.*

Multiple fractional expressions can be addressed by introducing multiple variables (q_i or k_i). Now, the decision to evaluate a q_i or k_i in

discrete values can be made for the different variables independently, with a trade-off between accuracy and computation time. One could roughly say that a *small* denominator, d, results in a *small* range of q, and a *small* domain, compared to d, results in a *small* range of k.

Exact and *Split Periods* can only be used for (iteration over) fixed domains. The other two allow parametric domains. Bernstein expansion (Section 4.4, Appendix C) can be used to compute parametric bounds on parametric polynomials over parametric continuous domains.

4.6.3 Other methods

In the experiments the techniques described above will be compared with two other methods that address the problem of converting quasi-polynomials into polynomials.

Poly Approx

In Section 4.2 it was mentioned that for integer polytopes the quasi-polynomial is actually a polynomial. In [127] B. Meister studies the periodicity of coefficients in quasi-polynomials. This results in methods to transform a polytope into an integer polytope, i.e. with its number of integer points expressed as a regular polynomial. More in general, a counting problem with a quasi-polynomial as solution can be approximated by a similar counting problem that has a polynomial as solution.

In the experiments we will use the "orthogonal expansion with inflation before expansion" technique presented in [128].

Drop Frac

In this method each fractional expression that appears in the coefficient of a monomial is replaced by either 0 or $(d-1)/d$ (its extremal values), depending on the sign of the monomial. If needed, the subdomains are divided into orthants,[6] such that all monomials have constant signs within a subdomain. For example,

$$\begin{aligned} -\tfrac{3}{4}n &\leq \left\{\tfrac{n}{5}\right\} n^2 - \left\{\tfrac{n-2}{4}\right\} n + \left\{\tfrac{n+1}{3}\right\} \leq \tfrac{4}{5}n^2 + \tfrac{2}{3} && \text{for } n \geq 0 \ , \\ 0 &\leq \left\{\tfrac{n}{5}\right\} n^2 - \left\{\tfrac{n-2}{4}\right\} n + \left\{\tfrac{n+1}{3}\right\} \leq \tfrac{4}{5}n^2 - \tfrac{3}{4}n + \tfrac{2}{3} && \text{for } n \leq 0 \ , \end{aligned}$$

[6] An orthant is the higher dimensional analogue of a quadrant in the plane.

since n^2 and 1 are always positive (or zero) and the sign of $-n$ is opposite to the sign of n.

4.6.4 Implementation

The transformations of quasi-polynomials into polynomials according to Table 4.4 and the methods explained in Section 4.6.3 have been implemented (for the multivariate case) in (or using) the `barvinok` library [160]. The command line interface and options that correspond with the methods used in the experiments can be found in Appendix E.

To improve the accuracy of the *Mod Classes* method, polynomials with subdomains with less than four integer values are evaluated in discrete points.

Split Period is called with a threshold parameter, Th. If the number of values a variable k_i can reach is larger than Th, no subdomains are created for each value of k_i and the method is the same as *Add Var* (for this k_i only). This provides a trade-off between accuracy and computation time.

Incremental Bernstein expansion (see Appendix C) is used to find the extrema of (parametric) polynomials over continuous (parametric) domains. This means that an approximation error caused by the difference between the maximal (minimal) Bernstein coefficient and the continuous-domain maximum (minimum) of the polynomial is added to the error caused by the difference between the continuous- and discrete-domain extrema. This extra approximation error is in many cases larger than the error from the transition to continuous-domain extrema. For example 10, the results listed in Table 4.5 would be 38 instead of 20 for *Split Periods* and even 284 instead of 120 for *Add Var*.

Until now the same method is applied on all fractional expressions of one piecewise quasi-polynomial. A hybrid method could be constructed that would decide at run-time for each fractional expression and in each chamber independently which method to use. This decision could be based on the denominator of the fractionals and the domain size. In Section 4.8 simple hybrid methods will be presented, which choose in each chamber independently between the *Exact* method and another method, based on the domain size.

For $n \in \{L-1, \ldots, N-1\}$:
$$\begin{aligned} c_n &= (a * b)_n \\ &= \sum_{k=0}^{L-1} a_{n-k} b_k \ . \end{aligned}$$

(a)

```
for (i=L-1;i<=N-1;i++){
  for (k=0;k<=L-1;k++){
    C[i]=C[i]+A[i-k]*B[k]; // S1(i,k)
  }
}
```

(b)

Figure 4.7: Definition (a) and corresponding C code (b) of a 1-D FIR filter.

4.7 Memory size estimation experiments

As simple test cases we study the memory usage of a one-dimensional FIR-filter and a matrix multiplication. Incremental Bernstein expansion (see Appendix C) is used to find the extrema of (parametric) polynomials over continuous (parametric) domains.

All measurements are performed on a PC with an AMD Athlon XP Processor running at 1830 MHz.

4.7.1 The 1-D FIR-filter test case

The input signal A with length N is filtered with the filter B with length L, as shown in the code fragment in Figure 4.7. Similar to the example in Section 4.1 we will use extrema of quasi-polynomials to examine the memory usage of several affine schedules of this program.

The iteration domain of the assignment statement `S1` is given by:

$$\mathcal{D}_{\texttt{S1}} = \{(i,k) \in \mathbb{Z}^2 \mid L-1 \leq i \leq N-1,\ 0 \leq k \leq L-1\} \ .$$

1-Dimensional schedule

We now consider affine 1-dimensional schedules of the iteration domain $\theta_{\texttt{S1}}(i,k)$:

$$\texttt{S1}(i_1,k_1) \text{ is executed before } \texttt{S1}(i_2,k_2) \text{ iff } \theta_{\texttt{S1}}(i_1,k_1) < \theta_{\texttt{S1}}(i_2,k_2) \ . \tag{4.30}$$

Table 4.6: Experiments each estimating the memory usage of a 1-D FIR-filter, for 20 different 1-D schedules, defined by the vector (u, v). $L = 50$, $N = 100$.

(a) Experiment over 20 schedule vectors (u, v), with $u \in \mathbb{Z}_0 \cap [-10, 10]$, $v = 1$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	1.3	0.06	99	197.00	0.00
Add Var	1.5	0.08	99	197.00	5.06
Mod Classes	2.3	0.11	99	197.00	0.22
Split Periods $_{Th=10}$	1.4	0.07	99	197.00	2.62
Split Periods $_{Th=100}$	3.7	0.19	99	197.00	0.22
Poly Approx	1.0	0.05	99	198.80	74.42
Drop Frac	0.9	0.05	99	2477.22	1079.81

(b) Experiment over 20 schedule vectors (u, v), with $u = 1$, $v \in \mathbb{Z}_0 \cap [-10, 10]$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	1.3	0.06	100	197.00	0.00
Add Var	1.8	0.09	100	197.00	5.34
Mod Classes	2.3	0.12	100	197.00	0.00
Split Periods $_{Th=10}$	1.7	0.08	100	197.00	3.01
Split Periods $_{Th=100}$	4.5	0.23	100	197.00	0.06
Poly Approx	1.0	0.05	100	202.80	76.52
Drop Frac	0.9	0.05	100	1410.89	715.47

Similar to (4.2), the set of live elements of array A at an execution point t is the set of elements that is accessed both before and after the point t:

$$\begin{aligned} LE_A(t) = \{ & j \in \mathbb{Z} \cap [0, N-1] \mid \exists (i_1, k_1), (i_2, k_2) \in \mathcal{D}_{\mathtt{S1}} : \\ & j = i_1 - k_1 = i_2 - k_2 \wedge \\ & \theta_{\mathtt{S1}}(i_1, k_1) < t \ \wedge \ t \leq \theta_{\mathtt{S1}}(i_2, k_2) \} \ . \end{aligned} \tag{4.31}$$

The live elements of array B and C, i.e. $LE_B(t)$ and $LE_C(t)$, can be defined similarly. According to Section 4.2 the number of live elements of A, i.e. $|LE_A(t)|$, can be expressed as a piecewise quasi-polynomial in t. The required memory for a schedule is the maximal number of live elements:

$$\max_t \left(|LE_a(t)| + |LE_b(t)| + |LE_c(t)| \right) \ .$$

In the experiments we use one-dimensional schedules of the form $\theta_{\mathtt{S1}}(i, k) = ui + vk$.

Tables 4.6 and 4.7 list the results of the measurements. All quasi-polynomials are of degree 2. The experiments each estimate the memory usage of a 1-D FIR-filter, for 20 different 1-D schedules, defined by

Table 4.7: Experiments each estimating the memory usage of a 1-D FIR-filter, for 20 different 1-D schedules, defined by the vector (u, v). $L = 50$, $N = 1000$.

(a) Experiment over 20 schedule vectors (u, v), with $u \in \mathbb{Z}_0 \cap [-10, 10]$, $v = 1$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	11.0	0.55	99	197.00	0.00
Add Var	1.5	0.08	99	197.00	5.12
Mod Classes	2.3	0.11	99	197.00	0.32
Split Periods $_{Th=10}$	1.4	0.07	99	197.00	2.65
Split Periods $_{Th=100}$	1.7	0.08	99	197.00	2.65
Poly Approx	1.0	0.04	99	198.80	74.48
Drop Frac	0.9	0.05	99	24077.22	10113.59

(b) Experiment over 20 schedule vectors (u, v), with $u = 1$, $v \in \mathbb{Z}_0 \cap [-10, 10]$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	3.0	0.15	100	1079	0
Add Var	1.1	0.06	100	1080.15	0.99
Mod Classes	2.0	0.10	100	1079	0
Split Periods $_{Th=10}$	1.0	0.05	100	1080.15	0.94
Split Periods $_{Th=100}$	4.9	0.24	100	1079	0
Poly Approx	1.0	0.05	100	1961	525.51
Drop Frac	0.9	0.04	100	2877.9	1042.92

the vector (u, v). The time, T, to compute the estimates includes both the generation and the bounding of the quasi-polynomials, summed over all 20 schedules. Min and Max are the extremal results (memory requirements of the schedules with the best and worst memory usage, respectively) over the 20 schedules and RMSE is the root mean square error over the 20 schedules of the results compared with the exact solutions.

Timing results for experiments with larger values of N (Figure 4.8) illustrate that the execution time of the *Exact* method is proportional to the domain size.

2-Dimensional schedule

A 2-D schedule is described by a 2×2 matrix M:

$$\theta_{\mathtt{S1}}(i, k) = M \begin{bmatrix} i \\ k \end{bmatrix} .$$

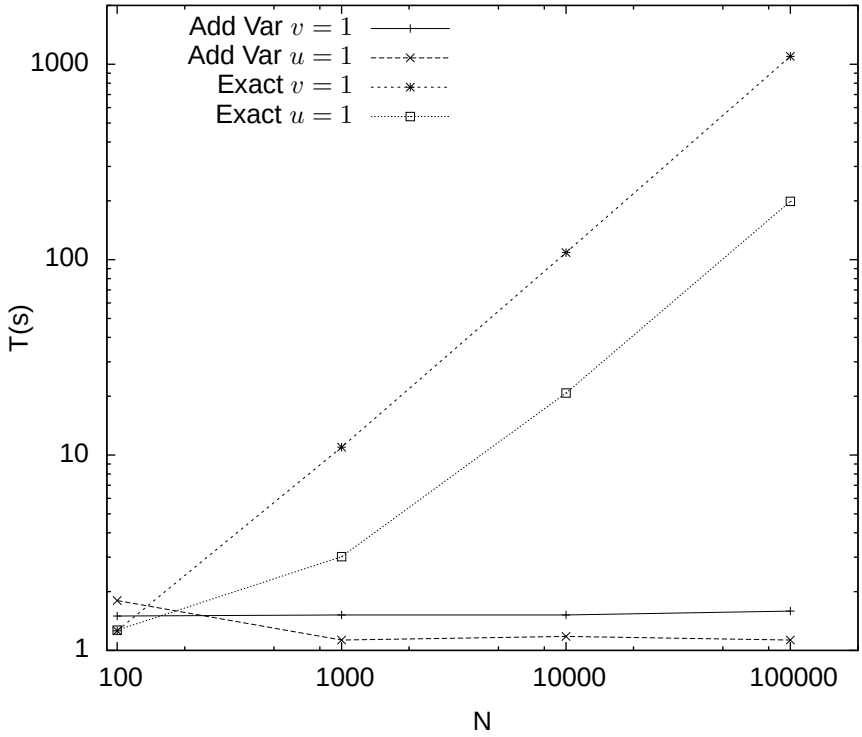


Figure 4.8: Time used for estimating the memory usage of a 1-D FIR filter as a function of N for 20 1-D schedules (cf. Table 4.6 and 4.7). $L = 50$.

The ordering relations $<$ and $\leq$ in (4.30) and (4.31) have to be replaced with the lexicographical ordering $\prec$ and $\preceq$, respectively. They can be transformed into ordinary inequalities by using Presburger formulas. For example,

$$(i_1, j_1) \preceq (i_2, i_2) \Leftrightarrow i_1 < i_2 \vee (i_1 = i_2 \wedge j_1 \leq j_2) \ .$$

Table 4.8 lists the results of 2 experiments. Note that the schedules in Table 4.8(a) are a subset of the schedules in Table 4.8(b).

A note on dealing with commutativity

Since the addition is commutative the products after the summation sign $\sum$ in Figure 4.7(a) can be calculated and summed in any order (cf.

Table 4.8: Experiments each estimating the memory usage of a 1-D FIR-filter, for 2-D schedules, defined by the matrix M. $L = 50$, $N = 100$.

(a) Experiment over 48 schedules: $M \in (\mathbb{Z} \cap [-1, 1])^{2\times 2}$ and $\det(M) \neq 0$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	20.2	0.42	99	197.0	0.00
Add Var	2.6	0.05	99	198.0	0.31
Mod Classes	3.1	0.06	99	197.0	0.00
Split Periods $_{Th=10}$	2.6	0.05	99	197.0	0.00
Split Periods $_{Th=100}$	3.6	0.08	99	197.0	0.00
Poly Approx	2.9	0.06	99	198.5	0.53
Drop Frac	3.7	0.08	99	198.5	0.48

(b) Experiment over 496 schedules: $M \in (\mathbb{Z} \cap [-2, 2])^{2\times 2}$ and $\det(M) \neq 0$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	500.3	1.01	99	197.00	0.00
Add Var	90.1	0.18	99	221.25	19.78
Mod Classes	148.7	0.30	99	197.00	0.01
Split Periods $_{Th=10}$	82.8	0.17	99	218.00	19.02
Split Periods $_{Th=100}$	1247.2	2.51	99	221.56	2.75
Poly Approx	84.4	0.17	99	272.00	30.73
Drop Frac	86.9	0.18	99	343.25	44.74

Section 3.4.3). As a result, all schedules are legal, as long as invocations with a same time stamp ($\theta_{\mathtt{S1}}(i_1, k_1) = \theta_{\mathtt{S1}}(i_2, k_2)$) do not write to the same array element. In the experiments this is ensured by $v \neq 0$ and $\det(M) \neq 0$. The definitions of live elements, as in (4.31), are in accordance with this commutativity. It states that $\theta_{\mathtt{S1}}(i_1, k_1) < \theta_{\mathtt{S1}}(i_2, k_2)$ but it is not needed that $(i_1, k_1) \prec (i_2, k_2)$. Writing the code in a single assignment form would add artificial dependences by ignoring the commutativity of the addition. Therefore, this is not done here.

Suppose a statement invocation $\mathtt{S2}(j)$ reads the value of $C[L-1+j]$. Since it is not known which invocation of $\mathtt{S1}$ will produce the final value of $C[L-1+j]$, all invocations that write to this element should be executed before the read by $\mathtt{S2}(j)$:

$$\mathcal{D}_{\mathtt{S1}\delta\mathtt{S2}} = \{(i, k, j) \mid (i, k) \in \mathcal{D}_{S1}, j \in \mathcal{D}_{S2}, : i = L - 1 + j\} \ .$$

Since we work with linear schedules it suffices to check the dependences for which $k = 0$ or $k = L - 1$. In the schedules considered

here it holds that

$$\begin{gathered}\forall k \in \{0, \ldots, L-1\} : \theta_{\mathtt{S1}}(i,k) \prec \theta_{\mathtt{S2}}(j) \\ \Updownarrow \\ \theta_{\mathtt{S1}}(i,0) \prec \theta_{\mathtt{S2}}(j) \wedge \theta_{\mathtt{S1}}(i,L-1) \prec \theta_{\mathtt{S2}}(j) \ .\end{gathered} \tag{4.32}$$

For the next example, matrix multiplication, the commutativity of the addition is dealt with in a similar way.

Discussion of results

The results in Tables 4.6, 4.7 and 4.8 indicate there is a trade-off between accuracy and computation time. The time needed by the *Exact* method, which iterates over all points, is proportional to the domain size (Figure 4.8). Also *Split Periods* is influenced by this size, but in a different way depending on the threshold. Typically, the error added by the Bernstein expansion is larger than the error introduced by the transition from discrete to continuous domains. Subdivision in smaller domains may improve on this. *Drop Frac* gives the worst results. Each fractional expression is set to 0 or $(d-1)/d$. In *Add Var* each fractional can also vary between 0 and $(d-1)/d$, but here all fractionals of the same linear expression will have the same value as they are connected by one variable. Detailed results, not shown in the tables, indicate that the *Mod Classes* method performs worse for larger periods of the periodic numbers (denominators inside fractionals) and *Add Var* performs relatively better for larger domains.

In the 2-D schedules the *Exact* method is much slower than for the 1-D schedules. This is caused by the fact that the iteration domains in the 2-D case contain more points than in the 1-D case. In the former each statement invocation occurs at a different value of the schedule function $\theta_{\mathtt{S1}}$ (since $\det(M) \neq 0$), while in the latter several invocations are projected on the same logical execution time.

The 20 piecewise quasi-polynomials of the experiment in Table 4.7(a), have domains composed of in total 160 polynomial chambers. Their sizes vary a lot (Figure 4.9). 140 chambers contain less than 64 points, while 20 chambers contain more than 512 points. However, those 20 chambers contain 103326 points of the domains while the 140 small chambers only have 2244 points all together. As a result the execution time of the *Exact* method is dominated by only 20 of the 160 chambers. In this example a hybrid method, using the *Exact* method for

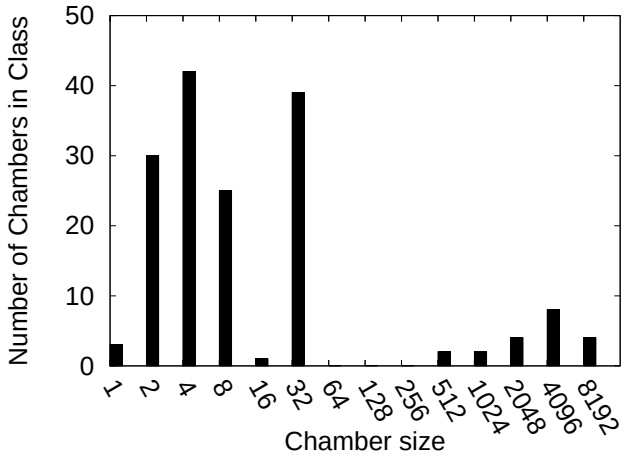


Figure 4.9: Histogram of the chamber sizes for the 20 schedules of the 1-D FIR filter considered in Table 4.7(a).

the small chambers and another method for the large chambers, could be very beneficial, assuming that deciding if a domain is small or large does not introduce a high extra cost. This will be tested in Section 4.8.

4.7.2 The matrix multiplication test case

Let $A \in \mathbb{R}^{M \times L}$, $B \in \mathbb{R}^{L \times N}$, $C \in \mathbb{R}^{M \times N}$. Then, the matrix product $C = BA$ is defined as

$$c_{ij} \quad = \quad \sum_{k=1}^{L} a_{ik} b_{kj}, \text{ for } i \in \{1, \ldots, M\},\ j \in \{1, \ldots, N\} \ .$$

Corresponding C code is found in Figure 4.10. In (a) array C is assumed to be initialized to 0, as is done by memory allocations in several programming languages (not in C). In (b) a statement is added to do the initialization.

```
for (i=1;i<=M;i++){
  for (j=1;j<=N;j++){
    for (k=1;k<=L;k++){
      C[i][j] += A[i][k]*B[k][j]; // S2(i,j,k)
    }
  }
}
```

(a)

```
for (i=1;i<=M;i++){
  for (j=1;j<=N;j++){
    C[i][j] = 0;                  // S1(i,j)
    for (k=1;k<=L;k++){
      C[i][j] += A[i][k]*B[k][j]; // S2(i,j,k)
    }
  }
}
```

(b)

Figure 4.10: C code of a matrix multiplication. In (b) the initialization of array C is made explicit.

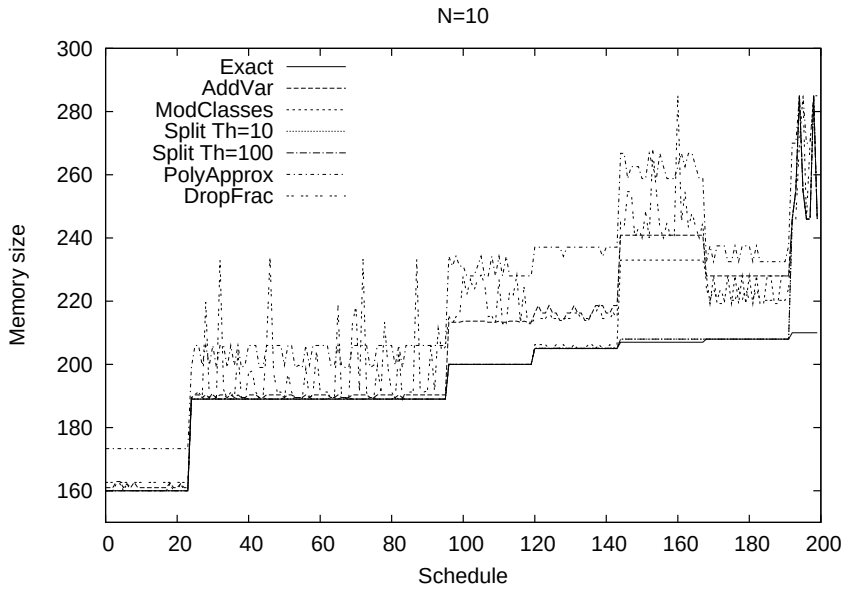
Without initialization statement `S1`

We define 1-dimensional linear schedules of the form $\theta_{\texttt{S2}}(i,j,k) = ui + vj + wk$. The set of live elements of array A can be expressed as

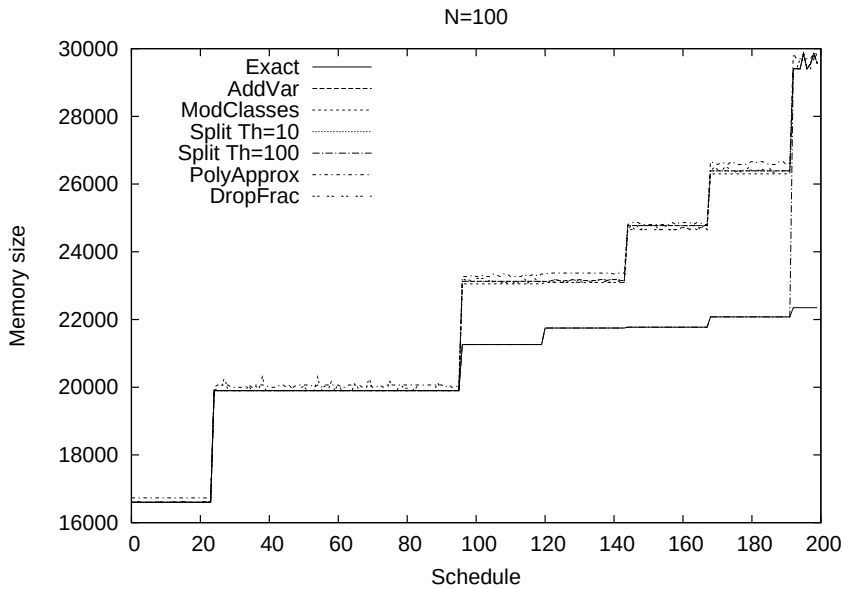
$$\begin{aligned} LE_A(t) = \{(i,k) \in \mathbb{Z}^2 \cap [1,M] \times [1,L] \mid \\ (\theta_{\texttt{S2}}(i,1,k) < t \ \vee \ \theta_{\texttt{S2}}(i,L,k) < t) \\ \wedge (\theta_{\texttt{S2}}(i,1,k) \geq t \ \vee \ \theta_{\texttt{S2}}(i,L,k) \geq t)\} \ , \end{aligned}$$

in which an existential variable j has been avoided by using a property similar to (4.32). As already mentioned in Section 4.7.1, there are little dependences that restrict the set of legal schedules, thanks to the commutativity of the addition.

Table 4.9 lists the results of several sets of schedules. The domain size, proportional to N, influences the computation time of several methods. The time needed by *Exact* is of course proportional to the domain size. The threshold of the *Split Period* method offers a trade-off between accuracy and computation cost. The accuracies of *Mod Classes* and *Split Periods* are always between those of *Exact* and *Add Var*. *Mod Classes* becomes faster when N increases since less modulo classes have less than 4 elements and are evaluated in all points. The time used by *Add Var*, *Poly Approx* and *Drop Frac* is independent of the domain size.



(a)



(b)

Figure 4.11: Memory size requirements of a matrix multiplication without S1 as a function of the schedule θ_{S2}. $u, v, w \in \mathbb{Z}_0 \cap [-3, 3]$, $\gcd(u, v, w) = 1$.

Table 4.9: Experiments each estimating the memory usage of a Matrix Multiplication without initialization statement `S1`. The 1-D schedules are defined by the schedule vector (u, v, w).

(a) Experiment over 200 schedules, with $u, v, w \in \mathbb{Z}_0 \cap [-3, 3]$, $\gcd(u, v, w) = 1$, $N = 10$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	10.8	0.05	160.00	210.0	0.00
Add Var	15.0	0.08	161.00	285.0	17.92
Mod Classes	28.4	0.14	160.00	285.0	14.67
Split Periods $_{Th=10}$	19.5	0.10	160.00	285.0	10.13
Split Periods $_{Th=100}$	19.5	0.10	160.00	285.0	10.13
Poly Approx	14.2	0.07	173.33	285.0	28.99
Drop Frac	15.3	0.08	162.67	285.5	20.10

(b) Experiment over 200 schedules, with $u, v, w \in \mathbb{Z}_0 \cap [-3, 3]$, $\gcd(u, v, w) = 1$, $N = 100$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	27.9	0.14	16600.00	22350	0.00
Add Var	15.0	0.08	16601.00	29850	2459.26
Mod Classes	19.0	0.10	16600.00	29850	2420.90
Split Periods $_{Th=10}$	14.8	0.07	16601.00	29850	2459.30
Split Periods $_{Th=100}$	121.7	0.61	16600.00	29850	1441.06
Poly Approx	14.2	0.07	16733.33	29850	2573.34
Drop Frac	15.3	0.08	16602.67	29850	2451.88

(c) Experiment over 920 schedules, with $u, v, w \in \mathbb{Z}_0 \cap [-5, 5]$, $\gcd(u, v, w) = 1$, $N = 10$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	65.0	0.07	136.0	212.00	0.00
Add Var	658.5	0.72	138.0	285.00	17.28
Mod Classes	234.0	0.25	136.0	285.00	7.18
Split Periods $_{Th=10}$	154.0	0.17	136.0	285.00	4.73
Split Periods $_{Th=100}$	154.0	0.17	136.0	285.00	4.73
Poly Approx	77.8	0.08	152.0	285.81	33.12
Drop Frac	82.0	0.09	149.5	622.85	82.68

For larger N the relative difference in accuracy between the methods becomes smaller.

In Table 4.9(c) schedules with larger coefficients of the iterators in the schedule function are present. As a result, the average denominator

of the fractional expressions, i.e. the period of the periodic numbers, increases. This is in favor of the *Split Periods* and to the detriment of the *Mod Classes* method.

The tables only give information about the average absolute accuracy and computation time of the methods. When searching for the best schedules it is more important that the relative order of the needed memory size as a function of the schedule is preserved. Therefore, in Figure 4.11, the results of the different methods were plotted as a function of the schedules sorted along the exact memory size.

The first 24 schedules are recognized by all methods as having the lowest memory demand. For the other schedules the relative ordering is roughly preserved if $N = 100$. For $N = 10$, especially *Drop Frac* has large deviations.

With initialization statement `S1`

Now, two schedule functions are needed

$$\begin{aligned} \theta_{\texttt{S1}}(i,j,k) &= ai + bj + c \\ \theta_{\texttt{S2}}(i,j,k) &= ui + vj + wk \ . \end{aligned}$$

There is one schedule restriction: each element of C should be initialized by the corresponding invocation of `S1` before its first read in an invocation of `S2`:

$$\begin{aligned} \theta_{\texttt{S1}}(i,j) &< \theta_{\texttt{S2}}(i,j,k) \\ ai + bj + c &< ui + vj + wk \\ c &< (u-a)i + (v-b)j + wk \\ c &\leq \min_{i,j,k} ((u-a)i + (v-b)j + wk) - 1 \ . \end{aligned}$$

For given a, b, u, v and w, the maximal value of c can be expressed as a function of M, N and L. Choosing c smaller than this maximal value can only increase the memory size requirements by augmenting the time between initialization and usage of the array elements. Therefore, the used schedules will be defined by a, b, u, v and w from which the

Table 4.10: Experiments each estimating the memory usage of a Matrix Multiplication with initialization statement `S1`. The 1-D schedules are defined by the vectors (a, b, c) and (u, v, w).

(a) Experiment over 32 schedules, with $a, b, u, v, w \in \{-1, 1\}$, $N = 100$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	3.5	0.11	22400	31257	0.00
Add Var	1.6	0.05	29530	39451	5788.26
Mod Classes	1.6	0.05	29530	39451	5788.26
Split Periods $_{Th=10}$	1.6	0.05	29530	39451	5788.26
Split Periods $_{Th=100}$	1.6	0.05	29530	39451	5788.26
Poly Approx	2.2	0.07	29724	39649	5944.98
Drop Frac	2.1	0.07	29530	39850	5865.26

(b) Experiment over 672 schedules, with $a, b, u, v, w \in \mathbb{Z}_0 \cap [-2, 2]$, $\gcd(a, b) = 1$, $\gcd(u, v, w) = 1$, $N = 100$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	103.8	0.15	19899	31257	0.00
Add Var	49.3	0.07	19899	39525	2861.27
Mod Classes	57.6	0.09	19899	39525	2826.65
Split Periods $_{Th=10}$	48.7	0.07	19899	39525	2861.27
Split Periods $_{Th=100}$	294.8	0.44	19899	39525	1931.47
Poly Approx	57.8	0.09	19999	39825	2949.43
Drop Frac	61.0	0.09	19899	39850	2832.65

(c) Experiment over 5600 schedules, with $a, b, u, v, w \in \mathbb{Z}_0 \cap [-3, 3]$, $\gcd(a, b) = 1$, $\gcd(u, v, w) = 1$, $N = 100$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	1083	0.19	16600	31481	0.00
Add Var	684	0.12	16601	39525	2076.05
Mod Classes	933	0.17	16600	39525	2026.44
Split Periods $_{Th=10}$	680	0.12	16601	39525	2075.97
Split Periods $_{Th=100}$	4767	0.85	16600	39525	980.15
Poly Approx	566	0.10	16733	39833	2184.61
Drop Frac	593	0.11	16602	39850	2063.38

value of c is computed as

$$\begin{aligned} c &= \min_{\substack{i\in[1,M]\\ j\in[1,N]\\ k\in[1,L]}} ((u-a)i + (v-b)j + wk) - 1 \\ &= (u-a)(u-a > 0\ ?\ 1 : M) + (v-b)(v-b > 0\ ?\ 1 : N) \\ &\quad + w(w > 0\ ?\ 1 : L) - 1 \ . \end{aligned}$$

Table 4.10 lists the results of several sets of schedules. The conclusions that can be drawn are similar to the case without initialization. The average computation time per schedule increases as the average denominator increases (from (a) over (b) to (c)).

4.8 Hybrid methods

The results of the experiments in Section 4.7 showed that the *Exact* method is sometimes the slowest and sometimes the fastest depending on the sizes of the domains. The domain sizes can vary a lot between experiments but also between chambers within an experiment as indicated in Figure 4.9.

Therefore, we constructed and tested hybrid methods. In chambers with a size smaller than or equal to a threshold parameter Th_{hybrid} the *Exact* method is used. In larger chambers one of the methods used in Section 4.7 is used. The experiments leading to the results in Table 4.6, 4.7 and 4.9 were repeated with hybrid methods to produce the results listed in Table 4.11, 4.12 and 4.13, respectively. If the method used in the larger chambers is also the *Exact* method the execution time indicates the overhead caused by the selection method. For example using Table 4.9(a) and 4.13(a), we get

$$T_{\text{overhead}} = T_{\text{Hybrid-Exact}} - T_{\text{Exact}} = 11.6 \text{ s} - 10.8 \text{ s} = 0.8 \text{ s} .$$

Here, the overhead is less than 10% of the total execution time. In the other examples it is even less.

We will call the methods used in Section 4.7 *basic methods*. Each basic method has a corresponding hybrid method. When comparing the tables in Section 4.7 with the tables in this section the following can be observed:

- The accuracy of a hybrid method is always higher than, or as high as, the accuracy of the corresponding basic method.

- In experiments where the *Exact* method is one of the fastest basic methods (Tables 4.6(a,b), 4.9(a and c)) all hybrid methods have an execution time close to that of the *Exact* method, except for *Split Periods* with $Th = 100$. The accuracy is also good, except for the hybrid methods using *Poly Approx* or *Drop Frac*.

- In experiments where the *Exact* method is one of the slowest basic methods Tables 4.7(a,b), 4.9(b)) only slight improvements in accuracy and slight differences in execution time are visible when going from a basic method to the corresponding hybrid method. This is caused by the fact that only a little fraction of the chambers have a size smaller than the hybrid threshold. In fact, these are the experiments where *Add Var*, *Mod Classes* and *Split Periods* are fast and accurate.

- A hybrid method can be faster than all basic methods, excluding *Poly Approx* and *Drop Frac* (Table 4.11 and 4.12).

These observations prove that the selection mechanism of the hybrid methods works well.

The hybrid method could be extended with a mechanism that selects which method to use in the large domains. Choosing between *Add Var*, *Mod Classes* and *Split Periods* is more complex than between *Exact* and one of these three methods. Such a method should study algebraic properties of the quasi-polynomials, such as degree and periods of periodic numbers and should have a selection heuristic based on these properties. The differences in accuracy and computation time between these three methods are much smaller than between one of these methods and the *Exact* method. As a result, the gain that can be obtained is smaller than the gain obtained by using the *Exact* method for small domains as presented in this section. Therefore, no effort was spent on extending the selection mechanism of the hybrid methods.

4.9 Related work

4.9.1 Memory size estimation

Computing the exact memory usage of a program, or bounding or approximating it has been the subject of numerous papers. A selection is given in [56]. Here only two methods are mentioned because of their relation to the methods presented in this chapter.

Zhu et al. [169, 168, 22] developed a method that uses polyhedral techniques. For each array the references are decomposed into disjoint linearly bounded lattices. The program code is regarded as a sequence of loop nests, called blocks. The memory size between the blocks is

Table 4.11: Experiments each estimating the memory usage of a 1-D FIR-filter, for 20 different 1-D schedules, defined by the vector (u, v). $L = 50$, $N = 100$. A hybrid method is used. For chambers with a size $\leq 50 = Th_{\text{hybrid}}$ the *Exact* method is used. For larger chambers the method listed in the first column is used. The experiments are similar to those used for constructing Table 4.6.

(a) Experiment over 20 schedule vectors (u, v), with $u \in \mathbb{Z}_0 \cap [-10, 10]$, $v = 1$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	1.3	0.07	99	197	0
Add Var	1.1	0.05	99	197	2.62
Mod Classes	1.1	0.05	99	197	0.22
Split Periods $_{Th=10}$	1.1	0.05	99	197	2.62
Split Periods $_{Th=100}$	3.0	0.15	99	197	0.22
Poly Approx	1.0	0.05	99	198.8	74.42
Drop Frac	1.0	0.05	99	2477	1079.32

(b) Experiment over 20 schedule vectors (u, v), with $u = 1$, $v \in \mathbb{Z}_0 \cap [-10, 10]$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	1.3	0.07	100	197	0
Add Var	1.2	0.06	100	197	3.01
Mod Classes	1.1	0.06	100	197	0
Split Periods $_{Th=10}$	1.2	0.06	100	197	3.01
Split Periods $_{Th=100}$	3.7	0.18	100	197	0
Poly Approx	1.1	0.05	100	202.8	76.52
Drop Frac	1.0	0.05	100	1410	714.75

computed from the linearly bounded lattices using Barvinok's algorithm. To determine the memory usage inside a block, the minimal and maximal iteration vectors that access a data element are searched. This is done for all elements of the corresponding linearly bounded lattice. This last step makes the method unsuitable for parametric problems and makes the computation time increase with the domain size, although less than other methods.

This disadvantage is taken as an argument for the methods presented in [56]. Bernstein expansion does not suffer from these disadvantages and allows to find parametric bounds. However, in the paper only polynomials are considered. With a reference to the methods of Meister [127], dealing with quasi-polynomials is avoided. This has been the motive for the work presented in this chapter.

Table 4.12: Experiments each estimating the memory usage of a 1-D FIR-filter, for 20 different 1-D schedules, defined by the vector (u, v). $L = 50$, $N = 1000$. A hybrid method is used. For chambers with a size $\leq 50 = Th_{\text{hybrid}}$ the *Exact* method is used. For larger chambers the method listed in the first column is used. The experiments are similar to those used for constructing Table 4.7.

(a) Experiment over 20 schedule vectors (u, v), with $u \in \mathbb{Z}_0 \cap [-10, 10]$, $v = 1$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	11.2	0.56	99	197	0
Add Var	1.1	0.05	99	197	2.65
Mod Classes	1.1	0.05	99	197	0.32
Split Periods $_{Th=10}$	1.1	0.05	99	197	2.65
Split Periods $_{Th=100}$	1.1	0.05	99	197	2.65
Poly Approx	1.0	0.05	99	198.8	74.48
Drop Frac	1.0	0.05	99	24077	10113.19

(b) Experiment over 20 schedule vectors (u, v), with $u = 1$, $v \in \mathbb{Z}_0 \cap [-10, 10]$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	3.0	0.15	100	1079	0
Add Var	0.9	0.05	100	1079	0
Mod Classes	1.2	0.06	100	1079	0
Split Periods $_{Th=10}$	0.9	0.05	100	1079	0
Split Periods $_{Th=100}$	4.6	0.23	100	1079	0
Poly Approx	1.0	0.05	100	1961	525.51
Drop Frac	0.9	0.04	100	2877	1042.53

4.9.2 Profiling

In this chapter static analysis methods are used to measure or estimate certain properties of an implementation. Another option is profiling, i.e. running an implementation and measuring the properties one is interested in. To be useful a profiling method has to do more than only measuring. It has to link the measurements with the data sets and pieces of code that are responsible for the results [123].

Typically, profiling slows down the execution and increases the memory usage with a nonnegligible factor. In [38] a statistical model is used to reduce this overhead. Only a subset of the memory accesses is sampled to estimate the reference distances[7]. Sampling windows re-

[7]The *reference distance*, i.e. the number of accesses between use and reuse, is easier to measure than the *reuse distance*, the number of different elements accessed (see Appendix B).

Table 4.13: Experiments each estimating the memory usage of a Matrix Multiplication without initialization statement `S1`, using a hybrid method. For chambers with a size $\leq 50 = Th_{\text{hybrid}}$ the *Exact* method is used. For larger chambers the method listed in the first column is used. The 1-D schedules are defined by the schedule vector (u, v, w). The experiments are similar to those used for constructing Table 4.9.

(a) Experiment over 200 schedules, with $u, v, w \in \mathbb{Z}_0 \cap [-3, 3]$, $\gcd(u, v, w) = 1$, $N = 10$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	11.6	0.06	160	210	0
Add Var	11.5	0.06	160	210	0
Mod Classes	11.5	0.06	160	210	0
Split Periods $_{Th=10}$	11.5	0.06	160	210	0
Split Periods $_{Th=100}$	11.5	0.06	160	210	0
Poly Approx	14.5	0.07	173	226	10.81
Drop Frac	15.3	0.08	162	285	9.23

(b) Experiment over 200 schedules, with $u, v, w \in \mathbb{Z}_0 \cap [-3, 3]$, $\gcd(u, v, w) = 1$, $N = 100$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	29.0	0.15	16600	22350	0.00
Add Var	15.7	0.08	16601	29850	2459.26
Mod Classes	19.7	0.10	16600	29850	2420.90
Split Periods $_{Th=10}$	15.7	0.08	16601	29850	2459.26
Split Periods $_{Th=100}$	117.9	0.59	16600	29850	1441.06
Poly Approx	14.5	0.07	16733.3	29850	2573.34
Drop Frac	15.5	0.08	16602.7	29850	2451.87

(c) Experiment over 920 schedules, with $u, v, w \in \mathbb{Z}_0 \cap [-5, 5]$, $\gcd(u, v, w) = 1$, $N = 10$.

Method	T(s)	T/Sched.(s)	Min	Max	RMSE
Exact	68.6	0.07	136	212	0
Add Var	68.9	0.07	136	212	0
Mod Classes	70.5	0.08	136	212	0
Split Periods $_{Th=10}$	69.0	0.07	136	212	0
Split Periods $_{Th=100}$	68.7	0.07	136	212	0
Poly Approx	79.2	0.09	152	232	18.88
Drop Frac	88.2	0.1	149	622	80.23

duce the error when the miss ratio is not constant during the execution.

The ATOMIUM tool suite [47] is used in the IMEC DTSE method-

ology (Data Transfer and Storage Exploration). It profiles the memory accesses and estimates the energy dissipation of an execution, for each element in a set of possible memory hierarchies.

The SLO tool (Section 3.2.2) not only records reuse distance information but also suggests, in a rough way, transformations that may improve the locality.

A more detailed comparison of existing tools can be found in [43, 41].

Profiling is only possible if the architecture on which an implementation will run is available. This is not a problem when optimizing software for a given processor. However, when targeting a future platform, e.g., a processor that is still in a design phase or an implementation on FPGA, such a platform does not exist yet. In some of these cases simulation is possible, which is orders of magnitude slower. Here, static analysis is likely to be more profitable.

4.10 Conclusion

The differences between extrema of polynomials over continuous and over discrete domains have been studied in this chapter. A bound on the absolute difference for a given polynomial had already been derived by Rivlin. In this section we presented a bound on the relative difference for an arbitrary polynomial of a given degree over a given interval.

The differences between the extrema over continuous and discrete domains are smaller for larger domains or lower degrees. Considering this property, novel methods were introduced for finding bounds on (piecewise) quasi-polynomials in discrete domains, based on bounds of polynomials in continuous domains. The estimation times are not proportional to the domain size, in contrast with the exact method that evaluates a quasi-polynomial in each discrete point of the domain.

The methods were implemented and tested by estimating the memory usage of several schedules of two small programs. For the 1-D FIR, the presented methods perform quite well. However, for the matrix multiplication there seems to be less advantage compared to existing techniques, a.o., the evaluation in each point of the discrete domain. Note that for multidimensional schedules or larger domains the latter may become much slower.

By introducing a simple selection heuristic, selecting the *Exact* method for small domains and another method for large domains, we have demonstrated that the good characteristics of different methods can be combined. A more complex selection heuristic based on other quasi-polynomial properties, in addition to the domain size, may possibly improve on these hybrid methods, but probably with less added value than this simple heuristic.

An advantage of Bernstein expansion is that symbolic maximization is possible as suggested in [56]. This possibility has not yet been thoroughly examined for the quasi-polynomial maximization. A disadvantage is that a large approximation error may be introduced by the Bernstein expansion, much larger than the error introduced by the difference between the discrete-domain and continuous-domain extrema of polynomials. The accuracy may be improved by subdivision of the domains. Doing this in an efficient way for arbitrary polyhedral domains is not trivial and may be a topic for future research.

Bounds on quasi-polynomials can result in, e.g., memory size estimates of an implementation. Such estimates are needed to guide loop transformations (Chapter 3). In the experiments in this chapter sets of schedules were scanned exhaustively. This is similar to the way Pouchet et al. [137] deal with small examples. The decoupling heuristic they present decreases the number of schedules to evaluate. Since this heuristic does not depend on the evaluation method (compilation and running the program in their case, i.e. iterative compilation), the heuristic can probably also be applied when evaluating a schedule with bounds on quasi-polynomials.

Chapter 5

Hardware generation from the polyhedral model

The previous chapters dealt with the search for and the application of loop transformations to improve the locality of an algorithm. This was done in a platform independent way. In this chapter the path towards a hardware implementation is considered, starting from the polyhedral model.
A hardware architecture with correspondence to the polyhedral model is presented. By automating the generation of control hardware, the influence of loop transformations on the hardware can be investigated by exploring (part of) the design space. One or more variants can then be selected for further refinement, optimizations and system integration.
The implementation of a 2D-IDWT was taken as a case study. Several variants without memory hierarchy were generated and compared with each other and with designs made with the high-level synthesis tools Impulse C and the Celoxica Handel-C compiler. One variant was further refined and integrated with other components on a FPGA to measure performance and power dissipation.

5.1 Introduction

As mentioned in Section 3.1.1 the manually designed RC-IDWT cannot reach its maximal execution speed due to its high bandwidth requirements. A new version with lower bandwidth requirements is needed. Loop transformations can improve the locality and thus reduce the data traffic. After these transformations, the whole design trajectory from

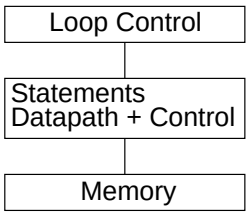


Figure 5.1: Top level representation of the hardware architecture.

high-level software description until hardware implementation has to be redone. As will be demonstrated in this chapter loop transformations not only influence the data access pattern but also have an impact on the control complexity and thus hardware implementation. Iterating over the design cycle, including both loop transformations and refinement to hardware, may be needed until the requirements are met. Therefore, this work should be automated and integrated as much as possible.

The class of applications that benefit from an implementation on FPGAs or dedicated hardware typically share the following characteristics: a low control complexity with little data dependence, much low-level parallelism, computation and data intensive operations. The restrictions required by the polyhedral model, static control, affine expressions, ..., correspond well with the first characteristic. The other two characteristics are not limited by the model. Therefore, most applications that may benefit from a hardware implementation, may fit into the polyhedral model.

A shorter path from loop transformation to hardware allows design space exploration and creation of a library of variants of a design. The latter is useful in systems that can be reconfigured dynamically. Several variants of a design are then stored and the one that best fits the working conditions at a certain time is loaded into the FPGA. Also when an application has to run on different platforms with, e.g., different computational power or memory bandwidth, generating variants of a design may be beneficial.

In this chapter a hardware architecture is presented which corresponds to the polyhedral representation. It is composed of two entities, one with the statement implementations (both datapath and control) and one with a controller that drives the iterators and triggers the statements (Figure 5.1). Since loop transformations change the loop control

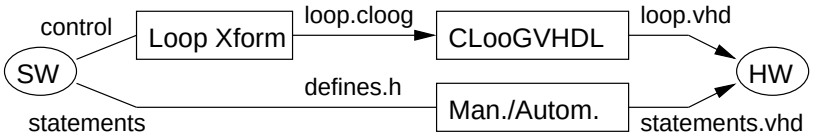


Figure 5.2: CLooGVHDL extends the CLooG software generation process (Figure 3.16) to create hardware.

and structure of an algorithm but leave the statement definitions unchanged (cf. Figure 3.16), the focus is on the automatic generation of the loop (nest) control hardware. The statement implementations can be reused among different loop transformation results. On the other hand several variants of the statements, e.g., trading-off area and number of clock cycles, can be connected to the same loop control entity. We have written a back-end to the CLooG code generator, called CLooG-VHDL (Figure 5.2), which fully automates the generation of a controller implementation, consisting of communicating automata. The generation of the data path and statement control (control which does not depend on the loop nest) is only partially automated, but as mentioned above this is less problematic.[1]

5.2 Hardware architecture

Since in the polyhedral model the control and statements are separated and recombined in the end, an architecture is proposed that has the same composition. At a high level this looks like Figure 5.1. The memory is shown as one entity but may be split into several banks or extended to a full memory hierarchy as in Figure 1.6. This will be demonstrated in Section 5.6.

In this section the architecture will be presented at a high level. A more detailed view is given in Section 5.3.

[1]Techniques that could automate this part do already exist and are used in high-level synthesis tools. Therefore, this part would imply a lot of implementation work or integration with existing tools (not necessarily more feasible) before further research is possible.

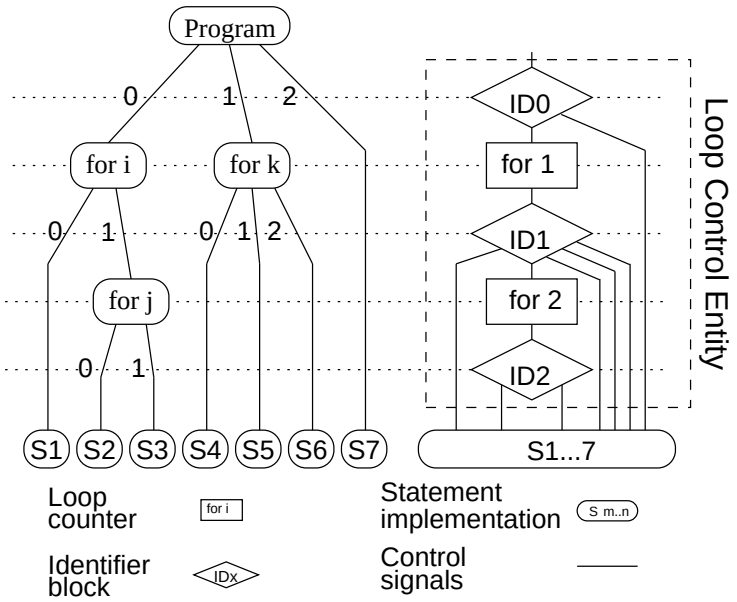


Figure 5.3: Example of a simplified abstract syntax tree (left) and hardware architecture (right) of the corresponding loop controller. A single hardware block implements all statements S1 . . . 7. The block for 1 implements the loops at depth 0: for i and for k. Which of the two is executed depends on the value received from ID0

5.2.1 Loop control architecture

Sequential execution

First, a sequential execution of the statement instances, equivalent to a software execution, is considered. In the abstract syntax tree in Figure 5.3, the numbers and iterators on the path between the top-node *Program* and a statement node correspond to the elements of the schedule vector of that statement. The controller is composed of a set of communicating automata, a so-called factorized implementation [19]. Each automaton corresponds to one dimension of the schedule vector. This results in two types of automata. A first type, the loop counter blocks, is responsible for the iterators. for 1, e.g., drives i and k. The other type, the identifier blocks, e.g., ID 1, corresponds to the elements of the ordering vectors. The loop counter blocks calculate the loop bounds and stride in function of the parameters, the iterators of surrounding loops and the more significant elements of the ordering vector. The identifier

Table 5.1: Comparison of implementation variants of the example in Figure 5.5 with dummy statements and fixed loop bounds (cols=8, rows=5). *Word width* denotes the number of bits used to represent the iterator values and some internal signals.

(a) Monolithic implementation

Word width	LE	f_{max}(MHz)
32 bit	269	116.90
4 bit	55	219.93

(b) Factorized implementation

Word width	LE	f_{max}(MHz)
32 bit	121	167.17
4 bit	47	232.29

blocks count from zero onwards to enumerate the different statements and loops at the level below. ID 1, e.g., counts from 0 to 1 if the value received from ID 0 is 0, and from 0 to 2 if that value is 1. When a statement or loop at the level below has finished its execution, the counter is incremented and the next statement or loop is triggered.

This architecture has, a.o., the following interesting properties.

- Experiments showed that the proposed factorized implementation consumes less area and reaches a higher clock frequency than a monolithic control block. Table 5.1 gives an example.

- The higher in the hierarchy, the lower the switching activity of the blocks. This allows clock gating [35] or other techniques for power reduction.

- For deep and large loop nests some control blocks might become very complex as they implement different behavior for a lot of different identifier values (components of ordering vectors). In that case it may be beneficial to regard subtrees as programs in their own right and give them their own control hardware. This results in more, but smaller and faster automata and a trade-off between clock speed and area. For example, in Figure 5.4(a) the for i and for k loop get their own controllers. In principle the two loops can now also operate in parallel. How the clock speed can be increased by duplicating hardware will be shown in Section 5.5.

- Pipelining is possible by inserting registers between the automata, which may increase the maximal clock frequency (Section 5.5).

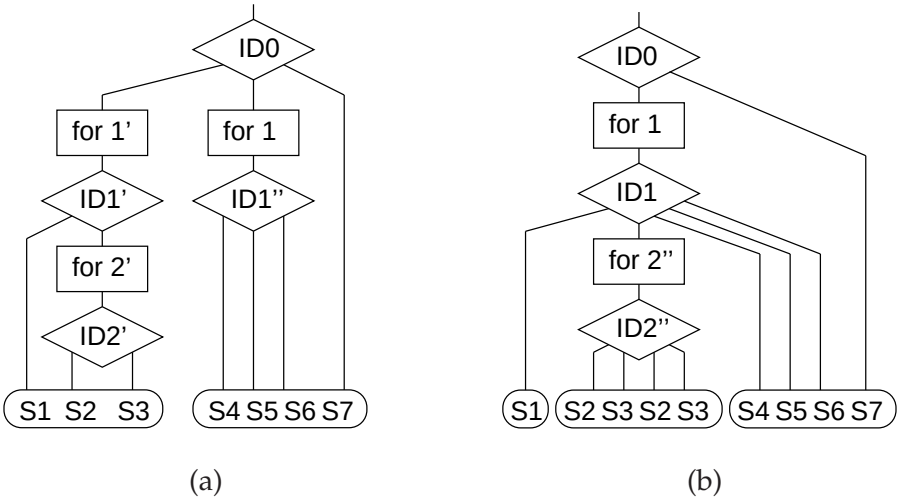


Figure 5.4: Duplicating hardware for reduction of complexity or introduction of parallelism

For the case of sequential execution, CLooGVHDL can generate a loop controller from a CLooG [27] input file.

Parallel execution

By creating separate hardware for different loops and the statements they control, it is possible to execute them in parallel. Dependence analysis using the polyhedral model allows to check which statement instances can run in parallel. Thanks to the factorized implementation, parallelization, if legal, is made straightforward by duplication of subtrees or statements. For example, in Figure 5.4(b) duplication of `S2` and `S3` makes it possible to run several iterations of the `j` loop in parallel or in a pipeline.

The detailed elaboration of code generation for parallel execution is left as future work. However, to prove that the proposed architecture indeed allows parallel execution an example has been adjusted manually from a sequential to a parallel execution as will be shown in Section 5.3.1.

5.2.2 Statement control and data path

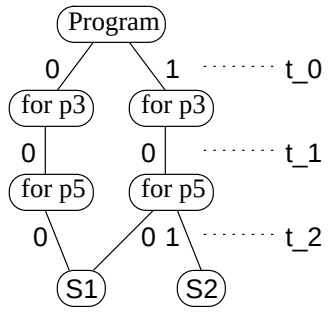
Since in the case of sequential execution no two statements operate at the same time, it is possible to share hardware among statements. Therefore, in Figure 5.3 all statements are implemented as a single VHDL process. The generation of the statements entity is not fully automated yet. This is not a large problem since a loop transformation only influences the controller entity and the statements entity can be reused for several loop transformation variants.

Splitting the statements entity, as in Figure 5.4, may be beneficial for similar reasons as for the control entity, i.e. parallelism and clock frequency.

5.3 Implementation details

```
1  for (p3=0; p3<=1; p3++){
2    for (p5=1; p5<=cols−2; p5++){
3      S1(0,0,p3,0,p5,0);
4    } /*end for p5*/
5  } /*end for p3*/
6  for (p3=2; p3<=rows−1; p3++){
7    for (p5=1; p5<=cols−2; p5++){
8      S1(0,0,p3,0,p5,0);
9      S2(0,1,p5,0,p3−1,0);
10   } /*end for p5*/
11 } /*end for p3*/
```

(a)



(b)

Figure 5.5: Example C output of CLooG (a) and corresponding AST (b). Since `S1` occurs at two places the structure in (b) is in fact not a tree anymore. Nevertheless, we continue to use the name AST.

This section refines the architecture presented in the previous section. To construct instances of the architecture presented here, several tools have been built or used to partially automate the work. The VHDL code generated by the tools contains a lot of redundancy (an example is given in Appendix F). A lot of signals or ports may be unneeded because their value is never read. Conditional assignments may have tens of cases from which only a few or even none differ. Modern

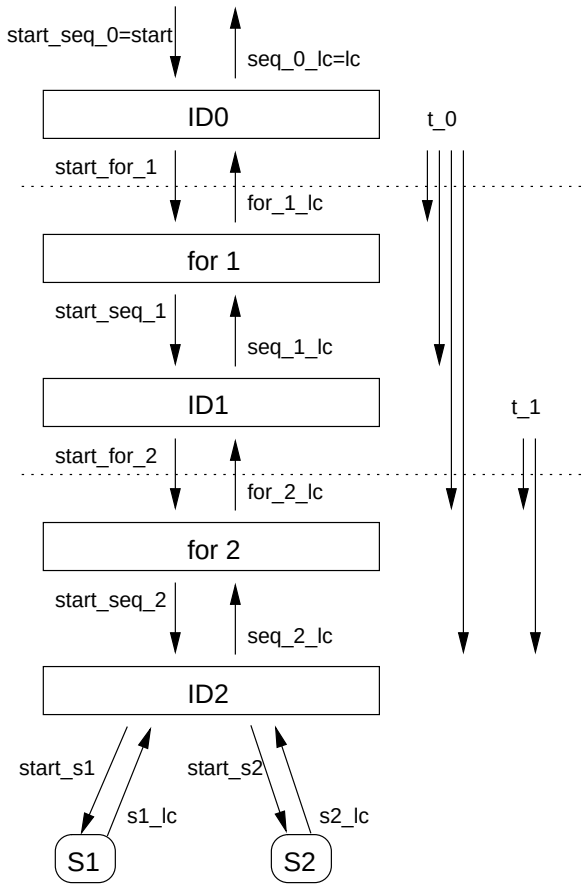


Figure 5.6: Detailed view of the factorized hardware architecture, corresponding to the code in Figure 5.5. Control signals between the automata are indicated. The loop iterators $p3$ and $p5$, and the statement arguments are omitted. The dotted lines indicate two possible positions to insert registers for pipelining.

synthesis tools have no problems with this redundancy and will optimize away all unnecessary signals, simplify combinational logic and propagate constants. Therefore, no attempt was made to generate *more optimal* code, since we believe that FPGA vendor synthesis tools are better in optimizing logic for the target FPGA. The code instead tries to give full optimization freedom to the synthesis tools.

5.3.1 Loop controller generation

Sequential execution

A more detailed view of the control architecture that corresponds to the code in Figure 5.5 is shown in Figure 5.6. The full VHDL code is listed in Appendix F).

The execution time of a statement invocation does not have to be known at compile time and may vary during execution or between variants of the statement implementations. A simple handshake between controller and statements makes this possible without losing clock cycles. A *start* signal triggers the execution of a statement. The *last cycle* signal becomes active one cycle before the statement finishes its execution. In the case of a one cycle execution time the *last cycle* output of a statement is thus simply connected to the incoming *start* signal. The same protocol is used between the different control automata of the loop controller.

This is demonstrated in Figure 5.7 for the example shown in Figure 5.5 and 5.6. The pulse on the *start* signal (= *start_seq_0*) (cycle 5) is directly passed to *start_for_1*, *start_seq_1*, *start_for_2* and *start_seq_2* (indicated with arrows). Since $t\_0 = t\_1 = t\_2 = 0$ (see Figure 5.5(b)) the identifier block ID2 triggers statement `S1` by raising *start_s1*. Since in this example the statement `S1` has an execution time of one clock cycle, the first cycle is also the last cycle, which means that the value of *start_s1* is passed to *s1_lc*. The inner loop for $t\_0 = t\_1 = 0$ only contains the single statement `S1`. Therefore the last cycle of `S1` is also the last cycle of the inner of the loop with iterator $p5$ and *seq_2_lc* becomes high. As a result, $p5$ is incremented at the next clock cycle and the inner of this loop is started by setting (i.e. holding) *start_seq_2* to '1', which is again passed to *start_s1*...*seq_2_lc*. Now $p5$ has reached its end value (*for_2_ev* = 2, not shown in the diagram), and *for_2_lc* and *seq_1_lc* (since there is only one loop at this level) are also raised. At the next clock cycle $p3$ is incremented and the inner of this loop is executed as before. Two cycles later, the first loop nest (line 1–5 in Figure 5.5(a)) has been executed and t_0 is incremented to start the execution of the second loop nest (line 6–11). Pulses on start signals are passed until *start_s1*. Here `S1` is followed by `S2`. *start_s2* is thus raised one cycle after *s1_lc* becomes high. At the same time t_2 is incremented. The execution time of `S2` is two cycles. Therefore, *s2_lc* becomes high one cycle later. When `S2` has finished its execution t_2 is reset, $p5$ is incremented and `S1` is started again. This

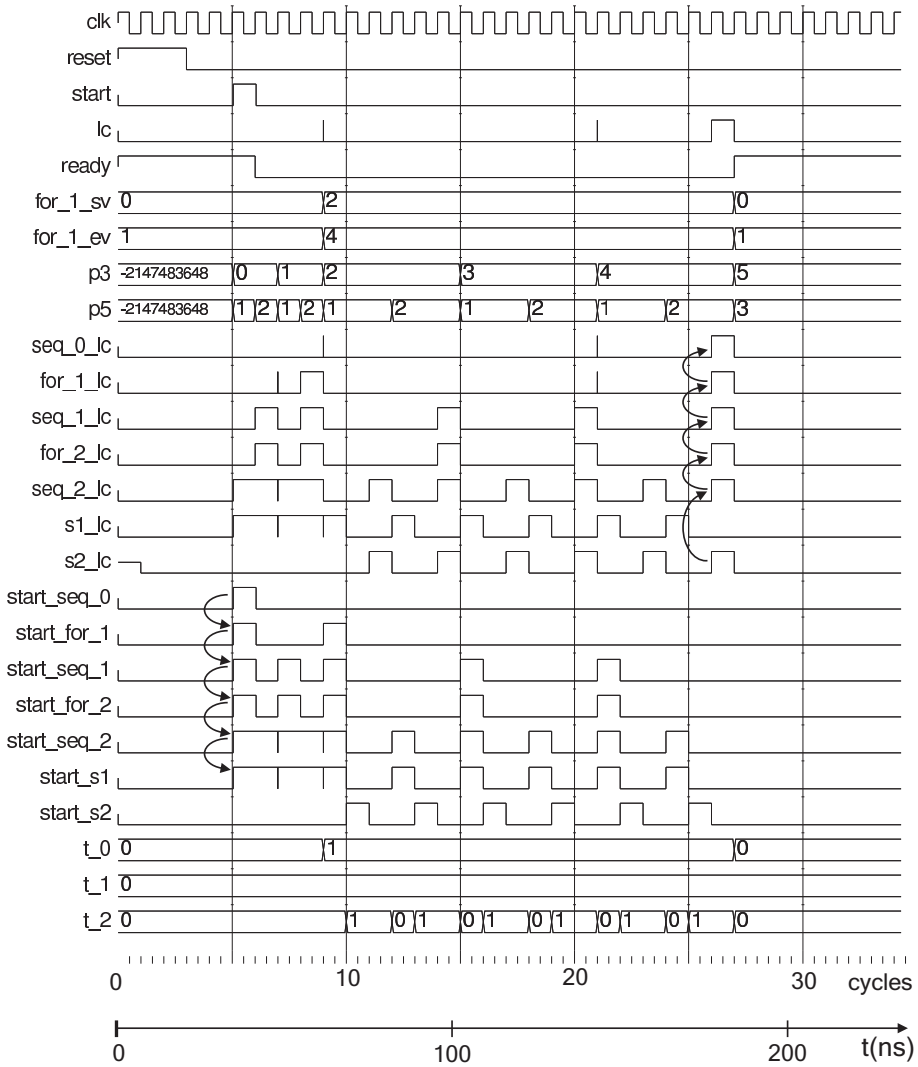


Figure 5.7: Timing diagram of the signals indicated in Figure 5.6 without extra registers between the automata. $f_{\text{clk}} = 146$ MHz.

continues until the last statement invocation. The last pulse on *s2_lc* is passed to *lc* as indicated with arrows. One can notice that at each clock cycle one statement is executing. No cycles are lost in the modules that pass the control signals.

A disadvantage of this protocol is that long combinational paths may occur in deeply nested loops. Therefore, an option is provided to

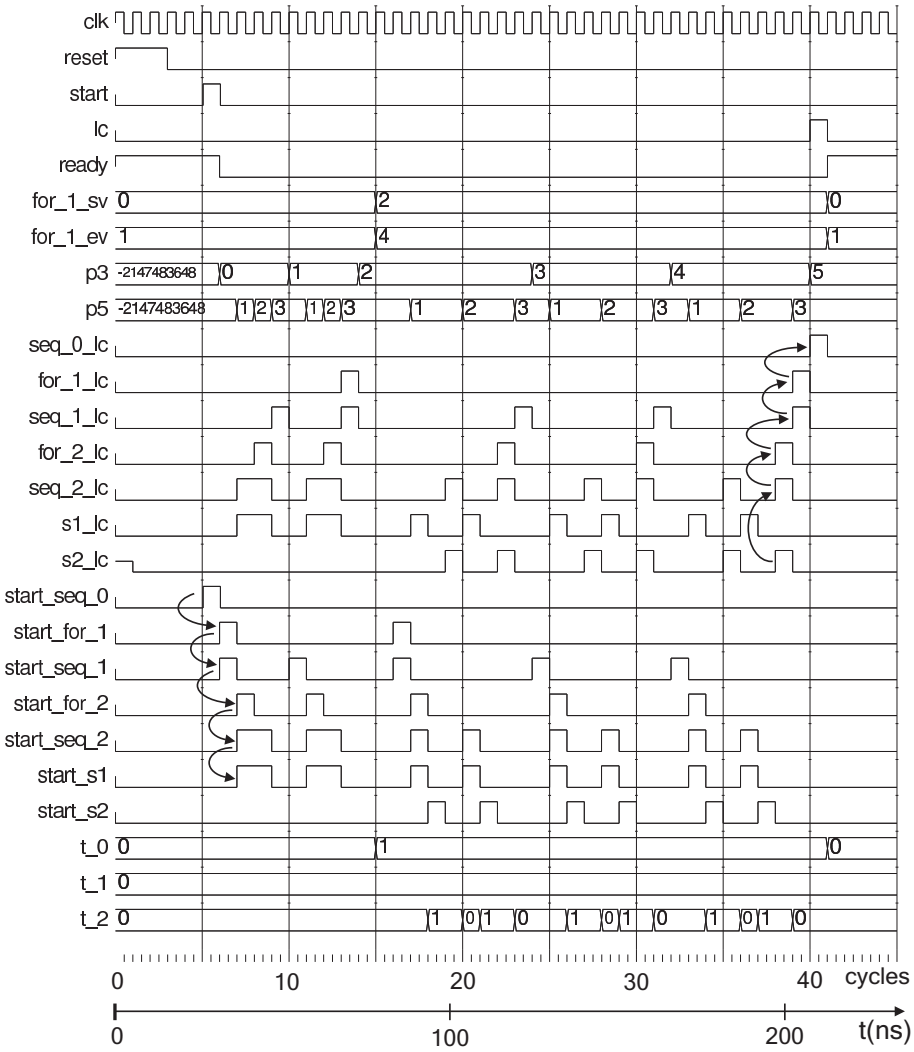


Figure 5.8: Timing diagram of the signals indicated in Figure 5.6 with extra registers between each identifier block and the loop counter below (indicated with dotted lines in Figure 5.6). $f_{\mathsf{clk}} = 192$ MHz.

include registers between the different levels of the control automata, leading to a pipelined implementation. These registers introduce extra clock cycles but increase the achievable clock speed by dividing combinational paths into smaller parts. At high levels the switching activity is low and the introduced clock cycle cost is fairly small. Experiments in Section 5.5 and 5.6.1 (Table 5.10) examine the influence on area, clock

frequency and execution time. Note that registers on signals that go up in the hierarchy can be inserted independently from buffers on signals that go down in the hierarchy.

A timing diagram of the example studied above with registers (also called buffers) inserted between each identifier block and the loop counter block below (dotted lines in Figure 5.6) is shown in Figure 5.8. The start signals now need two cycles to descend from the top to the bottom of the hierarchy (see arrows). For example, *start_seq_1* rises one cycle after *start_seq_0* and one cycle before *start_seq_2* and *start_s1*. In the given example the effect of shortening the clock cycle is outweighed by the increase in the number of cycles. This is due to the small number of iterations in the inner loop (only two). For a larger number of iterations the total execution time will be shorter than without extra registers.

Also the signals *t_0* and *t_1* can be buffered. As a result shifted versions of these signals are used in the lower levels of the controller hierarchy (not shown in the figure). However, using an unbuffered version will not violate the correct execution, since the value is only used after a start pulse arrives and the value cannot change while blocks lower in the hierarchy are active. The opposite is not allowed, e.g., buffering *t_0* without buffering the start signals.

The generated VHDL expressions for the loop bounds are equivalent to the expressions generated by CLooG. As a result, operators as `mod` and `div` are only synthesizable (in an efficient way) for powers of 2. Techniques as those presented in [171] could extend this synthesizable subset. The *Method of Differences* technique (MoD) exploits the repetitive evaluation of expressions in loops and uses differences of the terms of an expression to calculate the next value. An integer division by a constant can be converted into a multiplication with a constant and a shift, i.e. using so-called scaled reciprocals. Also [149] presents techniques to eliminate division and modulo operations, by inserting conditionals and using algebraic axioms and loop transformations.

The interface between the controller and the statements consists of *start* and *last cycle* signals together with the iterator values or possibly the values of the original iterators, i.e. those before the application of loop transformations. This corresponds with the statement arguments generated by CLooG with the option "`-cpp 1`". The interface to the outer world (top of `ID0`) consists of a *start*, *ready* and *last cycle* port and ports for all parameters. Input and output of data is done by statements and not directly visible to the loop controller.

Table 5.2: Synthesis results of control hardware of the IDWT variants listed in Table 3.6. RC HVs and LB VHs are variants with a smaller tile size, edge 1 instead of 4 in the smallest subband. The influence of the control optimizations is clearly visible (command line options "`-f 1`" (a) and "`-f -1`" (b)). Dummy statements are connected to the generated loop controller. No optimizations on the loop bound expressions have been applied (cf. Section 5.4). Results obtained with Altera QuartusII 6.1. for the Stratix EP1S25F1020C5. Integer data types of 10 bit used for iterators and parameters. (LB VH (a) uses 244,736 bits of memory to reduce the number of LEs used).

	#lines C		LE		f_{max}(MHz)	
	(a)	(b)	(a)	(b)	(a)	(b)
RC	32	32	223	223	185.49	185.49
RC H	166	55	1041	787	80.90	85.46
RC HV	456	69	2449	2533	55.08	11.23
RC HVs	246	59	1768	1820	64.21	16.53
RC V	98	46	668	632	87.90	90.64
RC VH	400	57	2318	2515	54.78	11.58
VH LB	1887	75	6563	3218	14.29	9.69
LB	83	56	468	521	119.65	54.35
LB H	926	71	3410	866	40.48	72.53
LB HV	2765	73	8737	2003	29.71	13.62
LB V	425	64	1188	734	73.34	42.86
LB VH	5639	75	21820	3195	4.52	10.73
LB VHs	716	69	3869	2710	13.45	11.62

The code optimization options of CLooG have a large impact on the generated VHDL. The synthesis results of the variants listed in Table 3.6 are shown in Table 5.2. All elements of the schedule vector are passed as arguments to the statements. As a result, only synthesizing the loop control entity would lead to unrealistic numbers of pins (from 141 to 477) and timing results that are not representative for a design with statement implementations on-chip. Therefore, dummy statements are connected to the controller and only 15 pins remain. The dummy statements write their arguments to registers (one set of registers for all statements). The logic AND of the output of these registers is connected to output pins to avoid them to be optimized away. Variants with a large number of lines of C code have a low clock frequency and a high LE usage. However, the block-based variants generated with the option "`-f -1`" (b) also appear to have a low clock frequency and

relatively large numbers of LEs. Section 5.4 investigates the origins of these bad synthesis results and presents methods to improve on these.

Parallel execution

Figure 5.9 shows a timing diagram corresponding to Figure 5.8 but with parallel execution of `S1` and `S2`. This is achieved by using the `par_sync` entity shown in Figure 5.10 and 5.11. This block has on its top side an interface that looks like that of any other block or statement considered before: a *start* and a *last cycle* (*lc*) signal. At the bottom side it drives two blocks *B1* and *B2* that could be statements, identifier blocks or loop counter blocks. The *lc* signal becomes high one cycle before both blocks have finished execution. Extension to more than two blocks is straight-forward. Extra code is needed to drive the input arguments of the blocks.

5.3.2 Statement control and data path generation

To automate the generation of the statement implementations, a tool would be needed that parses the statement definitions to construct an AST representation, does cycle and hardware assignment of the operations and generates synthesizable VHDL. To avoid the effort of building or customizing such a tool, an alternative was found in building and using a code generator, called *Steps2Process* (see below), and writing ad-hoc Vim scripts. Still, user interaction is needed but much less than in a fully manual design flow. In addition, a lot of work can be reused when iterating over a design.

The control hardware inside the statements entity has to take care of the timing of the operations on the data path and the accesses to the memories. *Steps2Process* was written to generate a statements entity starting from a file that contains the schedule information for each statement. Here *schedule* means the operations that have to be performed in consecutive cycles of a statement invocation. This differs from but is similar to the usage of schedule until now, which meant the ordering of statement invocations. Example input files are shown in Figure 5.12(b) and (c). Each line (not starting with "`--`") contains operations that should be executed in one clock cycle. Lines starting with "`-- #`", are used to separate the different statement descriptions and to direct conditional execution.

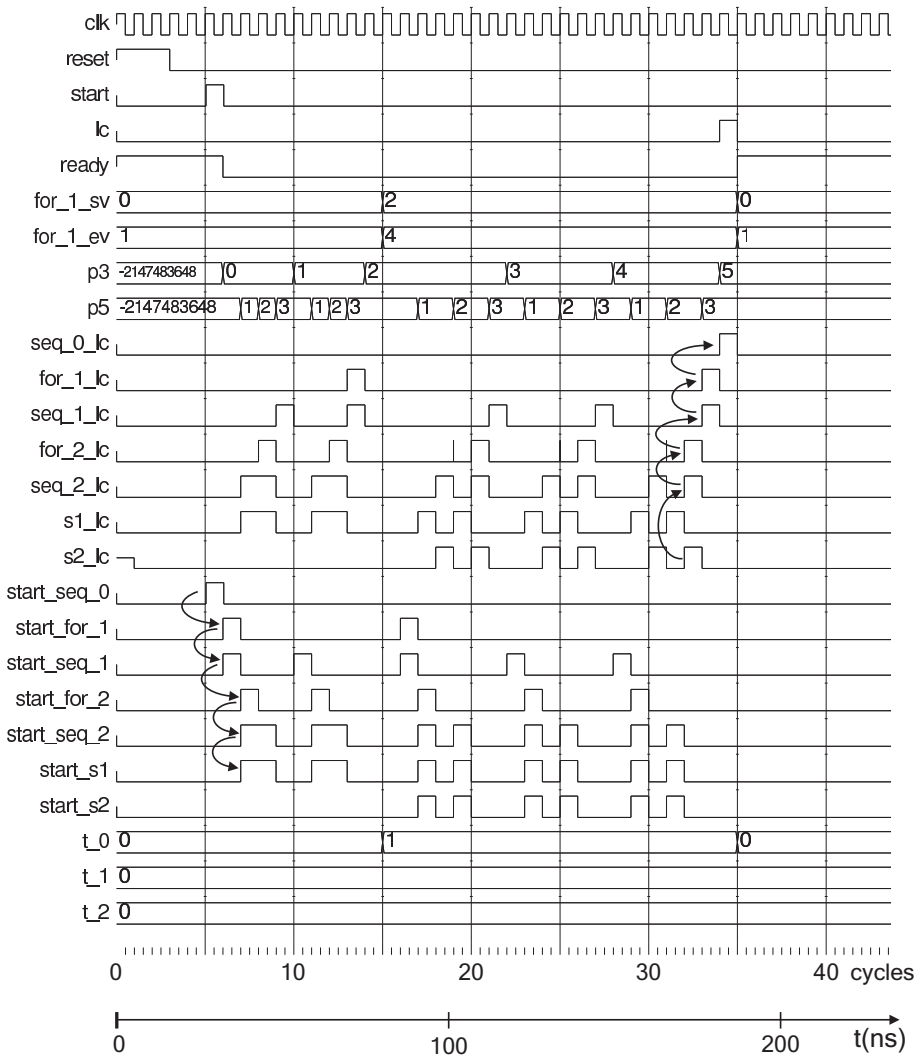


Figure 5.9: Timing diagram of the signals indicated in Figure 5.6 with extra registers and parallel execution of S1 and S2. $f_{\text{clk}} = 186$ MHz.

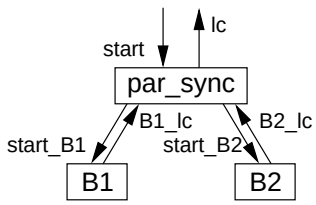


Figure 5.10: Entity to drive two statements or parts of loop nests in parallel.

```
1  LIBRARY ieee, work;
2  USE ieee.std_logic_1164. all ;
3
4  entity par_sync is
5  port(
6    clk , reset          : in std_logic ;
7    start                : in std_logic ;
8    lc                   : out std_logic ;
9    B1_lc , B2_lc        : in std_logic ;
10   start_B1 , start_B2  : out std_logic
11   );
12 end par_sync;
13
14 architecture rtl of par_sync is
15   signal B1_ready, B2_ready : std_logic ;
16 begin
17   start_B1 <= start;
18   start_B2 <= start;
19   lc <= (start and B1_lc and B2_lc) or
20     (not(start ) and (B1_lc or B2_lc) and
21     (B1_ready or B1_lc) and (B2_ready or B2_lc)) ;
22
23   ready: process(clk)
24   begin
25     if (clk='1' and clk'event) then
26       if (reset = '1') then
27         B1_ready <= '1';
28         B2_ready <= '1';
29       elsif start = '1' then
30       -- Set to '0' unless one cycle execution time
31         B1_ready <= B1_lc;
32         B2_ready <= B2_lc;
33       else
34       -- Set to '1' when block has finished
35         B1_ready <= B1_ready or B1_lc;
36         B2_ready <= B2_ready or B2_lc;
37       end if; --else reset / start / lc
38     end if; --clk
39   end process;
40 end architecture rtl ;
```

Figure 5.11: VHDL code of an entity that drives two blocks (statements or loop nests) in parallel.

```
#define S1(i)          \
  A[i]=                \
    (B[i]+B[i-1]+C[i]) \
    *B[i*2];

for(i=1;i<=N;i++){
  S1(i)
}
```

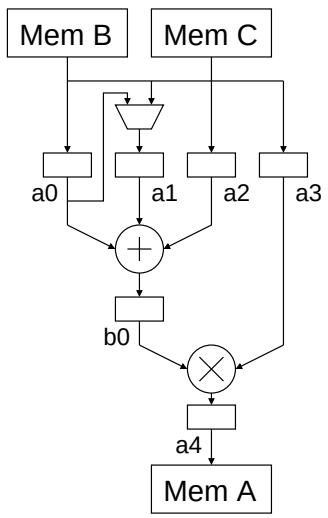
(a) C code

```
-- #define S1(i)
i:=S1_arg_0;
B_ad<=i;
B_ad<=i-1;
a0:=B_q; C_ad<=i;
a1:=B_q; B_ad<=i*2;
a2:=C_q;
a3:=B_q;
a4:=(a0+a1+a2)*a3;
A_ad<=i;A_we<='1';A_d<=a4;
```

(b) Statement execution steps

```
-- #define S1(i)
i:=S1_arg_0;
-- #if (i=1)
B_ad<=i; C_ad<=i;
B_ad<=i-1;
a0:=B_q;a2:=C_q;B_ad<=i*2;
a1:=B_q;
a3:=B_q; b0:=(a0+a1+a2);
a4:=b0*a3;
A_ad<=i;A_we<='1';A_d<=a4;
-- #else
B_ad<=i; C_ad<=i;
a1:=a0; B_ad<=i*2;
a0:=B_q; a2:=C_q;
a3:=B_q; b0:=(a0+a1+a2);
a4:=b0*a3;
A_ad<=i;A_we<='1';A_d<=a4;
```

(c) After optimizations



(d) Resulting data path

Figure 5.12: Example of optimizations on an intermediate file describing statement execution steps. Each line corresponds to one clock cycle. Memory is pipelined and a data element (X_q) is read two cycles after the address (X_ad) assignment. Arrays B and C are put in a separate memory and can thus be read in parallel. $B[i-1]$ can be reused from the previous iteration if $i > 1$. The calculation is split into two cycles, shortening the critical path. The number of cycles is reduced from 9 to 7 (8 for $i = 1$).

Consider the code in Figure 5.12(a). A memory access time of 1 cycle and a separate memory for each array is assumed (Figure 5.13). By

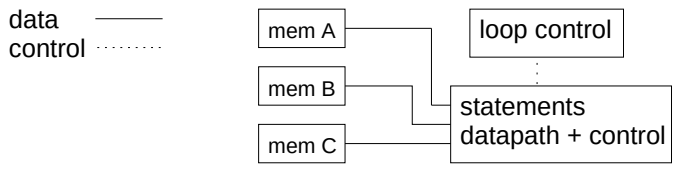


Figure 5.13: Architecture with separate memory for each array.

Table 5.3: Synthesis results of the design in Figure 5.12.

	Pins	LE	DSP blocks	f_{max} (MHz)
Loop Control	48	152	0	153.56
Statements (b)	98	196	2	123.11
Statements (c)	98	186	2	212.45
Loop Control + Statements (b)	96	324	2	123.17
Loop Control + Statements (c)	96	323	2	140.53

translating the C syntax of the operations in the statement definition in Figure 5.12(a) to a VHDL syntax, and writing the array accesses as consecutive memory accesses the code in (b) is obtained. Here several optimizations are possible. Memory B and C can be read in parallel and the element $B[i-1]$ should not be fetched from memory after the first iteration. The addition and multiplication are spread over two cycles to shorten the critical path. This results in the code in (c) with corresponding data path in (d). Synthesis results for the different entities and their compositions are shown in Table 5.3.

To synchronize the execution with external events the statements "`-- #waitdo`" (execute next line as soon as condition becomes true) and "`-- #waitndo`" (execute next line one cycle after condition becomes true) can be used. These wait statements will be used in Section 5.6.3 to ensure that the needed data is available in the buffer memories (after a read transaction from the main memory to the on-chip buffers) or data in buffers is not overwritten before it is written to the main memory.

1: **for** $i = 0, \left\lfloor \frac{8N+39}{8} \right\rfloor$
2: **for** $j = \max\left(0, \left\lceil \frac{-i-2}{7} \right\rceil, \left\lceil \frac{-i}{3} \right\rceil\right),$
$\min\left(\left\lfloor \frac{i+7M-5}{7} \right\rfloor, \left\lfloor \frac{4M-1}{4} \right\rfloor, \left\lfloor \frac{i+3M-3}{3} \right\rfloor\right)$
3: ...

(a)

1: **for** $i = 0, N + 4$
2: **for** $j = 0, M - 1$
3: ...

(b)

Figure 5.14: Part of the code of RC HVs, generated with the CLooG option "`-f -1`", before (a) and after (b) algebraic simplifications of the loop bounds.

5.4 Simplification of loop bound expressions

As observed in Table 5.2 using the CLooG option "`-f -1`" reduces the size of the generated software code. However, in several cases the generated hardware becomes slower or even larger when using this option. This deserves a closer investigation.

The two outer most loops in RC HVs generated with the option "`-f -1`" are shown in Figure 5.14(a). In the strip-mine transformations that are used in the loop transformation process only factors that are a power of 2 are used. Nevertheless, the generated code shows divisions by 3 and 7. These divisions are also present in the generated VHDL code, which causes the bad synthesis results.[2]

Fortunately, these divisions can be eliminated using the equations listed in Table 5.4. We will demonstrate this on the example in Figure 5.14(a). Using (5.1) the upper bound on i can be simplified:

$$\left\lfloor \frac{8N+39}{8} \right\rfloor = \left\lfloor \frac{8N+8\cdot 4}{8} \right\rfloor = \left\lfloor \frac{N+4}{1} \right\rfloor = N+4\,.$$

Since $i \geq 0$ we know that $-i \leq 0$ and $-i-2 \leq 0$ and thus (5.5) can be

[2]Divisions by integers that are not a power of 2 used to be unsynthesizable. However, the QuartusII tool has an integer divider in its library, which is used during technology mapping.

Table 5.4: Equations which can be used to simplify algebraic expressions.

Let $a, d \in \mathbb{N},\ b, c \in \mathbb{Z},\ x, y \in \mathbb{Q}$ then:

$$\left\lfloor \frac{ab+x}{ad} \right\rfloor = \left\lfloor \frac{ab+ax'}{ad} \right\rfloor = \left\lfloor \frac{b+x'}{d} \right\rfloor, \quad \text{with } x' = \left\lfloor \frac{x}{a} \right\rfloor \tag{5.1}$$

$$\left\lceil \frac{ab+x}{ad} \right\rceil = \left\lceil \frac{ab+ax'}{ad} \right\rceil = \left\lceil \frac{b+x'}{d} \right\rceil, \quad \text{with } x' = \left\lceil \frac{x}{a} \right\rceil \tag{5.2}$$

$$\max\left(\lceil x \rceil, \lceil y \rceil\right) = \lceil \max(x, y) \rceil \tag{5.3}$$

$$\min\left(\lfloor x \rfloor, \lfloor y \rfloor\right) = \lfloor \min(x, y) \rfloor \tag{5.4}$$

$$\max\left(0, \left\lceil \frac{ab-x}{d} \right\rceil\right) = 0 \quad \Leftrightarrow \quad ab - x \leq 0 \tag{5.5}$$

$$\Leftrightarrow \quad ab \leq x$$

$$\Leftrightarrow \quad b \leq x', \quad \text{with } x' = \left\lceil \frac{x}{a} \right\rceil \tag{5.6}$$

$$\left\lceil \frac{x}{d} \right\rceil > c \Leftrightarrow x > cd \tag{5.7}$$

$$\left\lfloor \frac{x}{d} \right\rfloor < c \Leftrightarrow x < cd \tag{5.8}$$

$$b < c \Leftrightarrow b \leq c + 1 \tag{5.9}$$

used to prove

$$\max\left(0, \left\lceil \frac{-i-2}{7} \right\rceil, \left\lceil \frac{-i}{3} \right\rceil\right) = 0\,.$$

In a similar way, using $i \geq 0$ and $M \in \mathbb{Z}$, the upper bound on j can be simplified:

$$\begin{aligned} & \min\left(\left\lfloor \frac{i+7M-5}{7} \right\rfloor, \left\lfloor \frac{4M-1}{4} \right\rfloor, \left\lfloor \frac{i+3M-3}{3} \right\rfloor\right) \\ = \quad & M + \min\left(\left\lfloor \frac{i-5}{7} \right\rfloor, \left\lfloor \frac{-1}{4} \right\rfloor, \left\lfloor \frac{i-3}{3} \right\rfloor\right) \\ = \quad & M - 1\,. \end{aligned}$$

A similar but more complex case is found in the outer loops of RC HV (Figure 5.15(a)). A study on the ranges of the different terms inside $\max$ and $\min$, using (5.7) and (5.8), leads to different cases depending

1: **for** $i = 0, \left\lfloor \frac{8N+63}{32} \right\rfloor$
2: **for** $j = \max\left(0, \left\lceil \frac{60i-15N-44}{68} \right\rceil, \left\lceil \frac{512i-128N-975}{16} \right\rceil, \left\lceil \frac{-32i-27}{4} \right\rceil, \left\lceil \frac{28i-7N-20}{36} \right\rceil\right),$
$\min\left(\left\lfloor \frac{24i+5M+36}{20} \right\rfloor, \left\lfloor \frac{136i+31M+224}{124} \right\rfloor, \left\lfloor \frac{4M+47}{16} \right\rfloor\right)$
3: ...

(a)

1: **for** $i = 0, \left\lfloor \frac{N-1}{4} \right\rfloor + 2$
2: **for** $j = (4i - N < 3 \,?\, 0 \,:\, (4i - N > 7 \,?\, 2 \,:\, 1)),$
$\left(i == 0 \,?\, 2 + \left\lfloor \frac{M-1}{4} \right\rfloor \,:\, 3 + \left\lfloor \frac{M-1}{4} \right\rfloor\right)$
3: ...

(b)

Figure 5.15: Part of the code of RC HV, generated with the CLooG option "`-f -1`", before (a) and after (b) algebraic simplifications of the loop bounds.

Table 5.5: The bad synthesis results in the block-based variants are improved after algebraic simplification of the loop bounds in the code generated with the option "`-f -1`" (c). (a) and (b) are taken from Table 5.2.

	LE			f_{max}(MHz)		
	(a)	(b)	(c)	(a)	(b)	(c)
RC HV	2449	2533	1021	55.08	11.23	63.01
RC HVs	1768	1820	996	64.21	16.53	78.83
RC VH	2318	2515	1024	54.78	11.58	61.08
VH LB	6563	3218	1784	14.29	9.69	36.81
LB HV	8737	2003	1130	29.71	13.62	57.99
LB VH	21820	3195	2148	4.52	10.73	28.92
LB VHs	3869	2710	1781	13.45	11.62	33.85

on the value of i, three and two cases respectively, as shown in Figure 5.15(b).

Using the equations in Table 5.4. all variants listed in Table 5.2 can be simplified to only contain divisions that can be implemented as shifts, i.e. divisions by powers of 2. The synthesis results after these optimizations are shown in Table 5.5. Now the results are better than those obtained with the option "`-f 1`".

The techniques presented in this section to simplify or eliminate integer divisions are an extension to the techniques presented in [149]. That the divisions with dividers that are not a power of 2 can be eliminated corresponds to the fact that all strip-mine transformations were done with factors that are a power of 2. Of course, in the general case it is not always possible to eliminate such divisions using the techniques presented here, e.g., when strip-mining is done with a factor 3. In these and similar cases other methods, such as the *Method of Differences*, should be used [171, 149].

5.5 Optimizing the clock frequency

5.5.1 Loop control

We have mentioned two methods to increase the clock frequency of the loop control entity: splitting a program into partitions each with their own loop controller (Section 5.2.1) and inserting extra pipeline registers between the automata (Section 5.3.1). One of the optimizations done by synthesis tools is retiming [130, Ch.5], i.e. moving registers across combinational blocks to minimize the clock cycle (or to minimize the number of registers). Inserting registers not only splits one path into two shorter parts, but also gives more retiming opportunities.

To test and compare the two options the example in Figure 5.16 is considered. The program in (a) is split into four parts in (b), one top-level loop and three subtree (ST) loop nests. The corresponding hardware architectures are shown in Figure 5.17. The Greek symbols denote places where optional registers can be inserted. Synthesis results are found in Table 5.6. In (a) and (b) no statements are connected to the loop controller. As a result, 103 pins are used, the major part for the statement arguments. In (c) and (d) the simple statements defined in Figure 5.16(c) are connected to the controller and the number of pins drops to 18. The unused statement arguments are then optimized away. In some cases this results in higher clock speeds.

It is clear that inserting registers increases the reachable clock speeds, although, not in all cases. Inserting registers at the highest level (α) almost always results in a lower frequency. The presence of extra registers here hinders some optimizations, as can be seen in the number of logic elements, which raises more than the number of registers. γ clearly is the best location for optional registers. Therefore, the

Table 5.6: Synthesis results of the architectures in Figure 5.17 for a Stratix EP1S25F1020C5. Figures without statements (a and b) and with statements (c and d) for different choices of optional register insertion. (a) and (c) correspond to Figure 5.17(a). (b) and (d) correspond to Figure 5.17(b). The registers (last column) are contained in the LEs.

(a)

Opt. Regs	f_{max} (MHz)	LE	Regs
None	59.79	286	42
α	57.39	297	46
β	69.18	295	47
γ	82.19	289	47
$\alpha+\beta$	68.55	303	51
$\alpha+\beta+\gamma$	106.12	304	56
$\alpha+\gamma$	74.86	298	51
$\beta+\gamma$	109.63	296	52
2γ	83.21	292	52
$\beta+2\gamma$	109.10	299	57

(b)

Opt. Regs	f_{max} (MHz)	LE	Regs
None	68.34	338	65
α	66.45	350	69
β	71.77	348	72
γ	102.30	343	67
$\alpha+\beta$	72.46	360	76
$\alpha+\beta+\gamma$	104.30	362	78
$\alpha+\gamma$	93.77	355	71
$\beta+\gamma$	109.57	350	74
2γ	99.71	343	69
$\beta+2\gamma$	107.28	350	76

(c)

Opt. Regs	f_{max} (MHz)	LE	Regs
None	55.69	308	46
α	58.05	317	50
β	68.67	324	51
γ	77.36	317	51
$\alpha+\beta$	67.35	333	55
$\alpha+\beta+\gamma$	105.65	333	60
$\alpha+\gamma$	75.84	326	55
$\beta+\gamma$	109.94	325	56
2γ	78.76	320	56
$\beta+2\gamma$	105.98	328	61

(d)

Opt. Regs	f_{max} (MHz)	LE	Regs
None	74.81	384	69
α	72.58	387	73
β	75.77	380	76
γ	116.08	383	71
$\alpha+\beta$	74.07	392	80
$\alpha+\beta+\gamma$	130.45	393	82
$\alpha+\gamma$	95.17	386	75
$\beta+\gamma$	118.04	381	78
2γ	114.14	383	73
$\beta+2\gamma$	130.01	381	80

```
for(i=0;i<=N;i++){
  for(j=i;j<=N;j++){
    S1(i,j) ;
  }
  for(j=0;j<=2*N-1;j++){
    S2(i,j) ;
    S3(i,j) ;
  }
  for(j=-N;j<=0;j++){
    for(k=i-j;k<=2*N;k++){
      S4(i,-j,k) ;
    }
  }
}
```

(a)

```
// Subtrees:
#define S1st(i)    \
for(j=i;j<=N;j++){ \
  S1(i,j) ;        \
}
#define S2st(i)        \
for(j=0;j<=2*N-1;j++){ \
  S2(i,j) ;            \
  S3(i,j) ;            \
}
#define S3st(i)          \
for(j=-N;j<=0;j++){      \
  for(k=-j+i;k<=2*N;k++){\
    S4(i,-j,k) ;         \
  }                      \
}

// Top:
for(i=0;i<=N;i++){
  S1st(i) ;
  S2st(i) ;
  S3st(i) ;
}
```

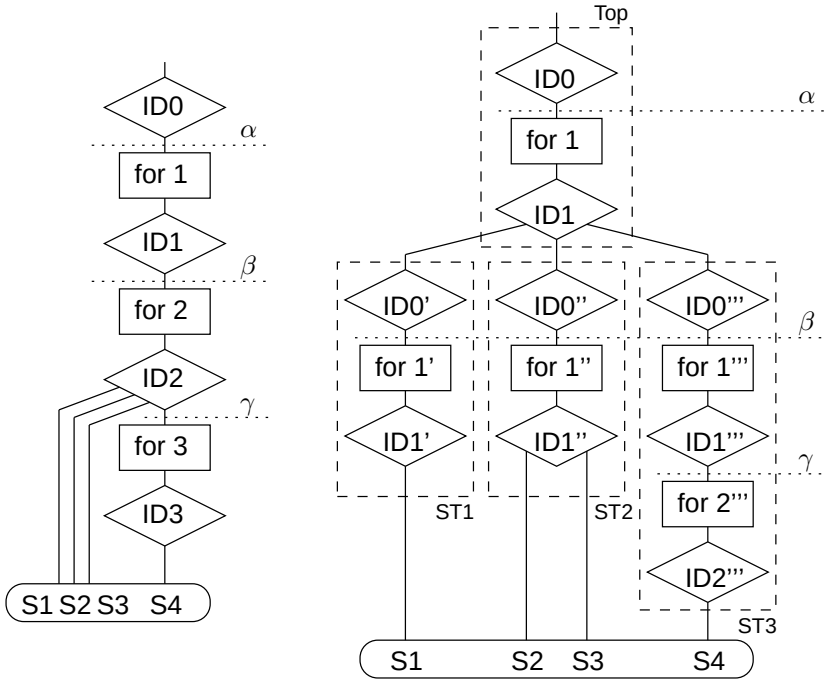
(b)

```
-- #define S1(i,j)
i:=S1_arg_0; j:=S1_arg_1;
-- #if (j>=i+2)
S1_out <= '1';
-- #else
S1_out <= '0';
-- #define S2(i,j)
i:=S2_arg_0; j:=S2_arg_1;
-- #if (i>=j)
S2_out <= '1';
-- #else
S2_out <= '0';
```

```
-- #define S3(i,j)
i:=S3_arg_0; j:=S3_arg_1;
-- #if (i>=j)
S3_out <= '0';
-- #else
S3_out <= '1';
-- #define S4(i,j,k)
k:=S4_arg_2;
-- #if (k>=9)
S4_out <= '0';
-- #else
S4_out <= '1';
```

(c)

Figure 5.16: Program example written as a single loop nest (a) and split into one top-level and three sub-level loop nests (b). Statements are defined as *Steps2Process* input (c).



(a) (b)

Figure 5.17: Hardware architectures for the example in Figure 5.16 (a) and (b). α, β and γ denote places where optional registers can be inserted.

option to place double registers at that location was tried (2γ, $\beta + 2\gamma$). Note that retiming is applied after placing the registers, which may insert combinational logic between the registers of such a pair.

In the synthesis results of designs generated with CLooGVHDL we observed that in the logic elements the utilization of LUTs is much higher than the usage of registers. This means that registers may be inserted without extra hardware cost and without the need to rerun technology mapping tools. This makes it possible to integrate the optional register insertion with the retiming optimizations done during place and route. In this thesis the register insertion options are explored exhaustively. After integration with retiming and place and route tools, timing information becomes available that may be used to guide the insertion process and avoid an exhaustive scan.

Composing the loop controller out of four subcontrollers increases the area usage in a larger amount than placing registers, but also leads to higher clock frequencies. The effect is larger with the statements connected and depends heavily on the inserted registers.

The influence on the number of clock cycles is not shown here since the relative impact depends on the value of N, which can be chosen arbitrarily in this artificial example. The number of cycles will be studied in the IDWT case study (Table 5.10).

5.5.2 Data path

Since the user has more control over the construction of the data path all classical methods can be used here. This has been demonstrated in Figure 5.12 by spreading the computations over two cycles to shorten the critical path.

5.6 Case study: exploration of IDWT variants

In this section hardware implementation variants of a 2-D IDWT are generated and compared. A simple memory architecture assuming one memory for each array and short fixed access times, as if it were on-chip memories, are considered (Figure 5.13). In a final implementation an external memory will be needed as the used data sets are too large to fit in the FPGA memory blocks. Nevertheless, the performance of these systems without memory hierarchy are a good measure for the perfor-

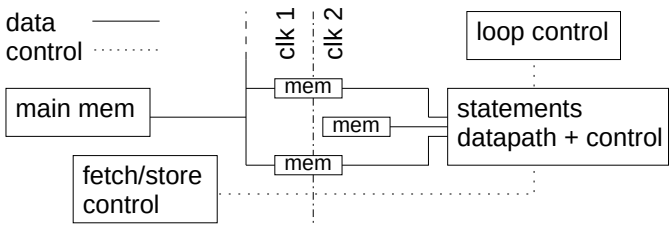


Figure 5.18: Architecture with memory hierarchy. Fetching and storing data is done in parallel with the processing. The execution speed is determined by the slowest of these two processes. Multiple local buffers allow parallel memory accesses.

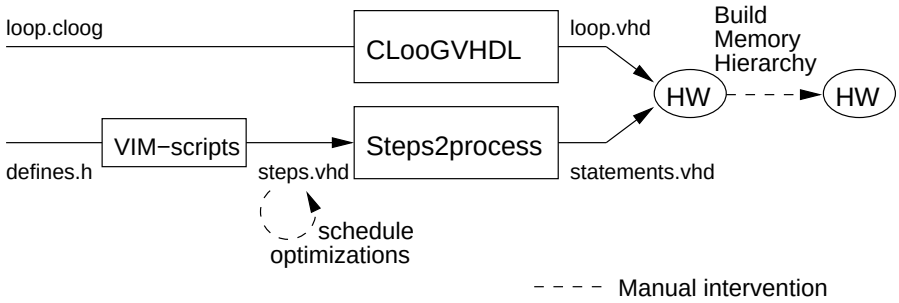


Figure 5.19: Design flow used to generate hardware implementation variants of the 2-D IDWT.

mance of a final implementation with memory hierarchy as long as the system is not bandwidth limited. By replacing the memories by dual-port memories, using the other port to transfer data to and from the external memory (Figure 5.18) such that at each time the needed data is available in the on-chip memories, the execution times of the statement invocations stay the same as without memory hierarchy (expressed as the number of clock cycles since the clock frequency will probably drop when the design is extended and more FPGA area is used). Some extra cycles will be needed for the first fetch operations and the last store operations. All other data transfers may be done in parallel with the statement executions. If the design is bandwidth limited the performance will be assumed to be proportional with the available bandwidth.

Table 5.7: Amount of data exchange (in pixels) to/from the main memory (left side of Figure 5.18) for different IDWT variants. k=number of transformation levels, R/C=number of rows/columns of image.

Variant	Amount of data exchange		Burst usage
	$k \in \mathbb{N}$	$k = 3$	
RC	$\frac{16}{3}RC(1 - 1/4^k)$	5.25RC	50%
LB	$\frac{8}{3}RC(1 - 1/4^k)$	2.625RC	100%
LB V	2RC	2RC	100%

5.6.1 Generating hardware variants

After some preprocessing steps (conversion to dynamic single assignment to eliminate false dependences and unrolling of the outer loop to remove the exponential expressions) the code can be represented in the polyhedral model (Section 3.7).

URUK scripts (Section 3.5.2) were written for different loop transformation sequences (Section 3.7 and 3.8), to generate IDWT variants: untransformed (RC), fused (LB) and fused-and-tiled (LB V).[3] How the bandwidth requirements change by doing loop transformations is shown in Table 5.7. Here, it is assumed that small data sets, in the order of lines of an image, can be stored on-chip, but large sets, in the order of a frame or a subband, have to be stored in external memory, a.o., the input and final output (both with size RC). LB V reaches the minimal possible bandwidth requirements, i.e. reading the input and storing the output. The burst usage indicates the amount of data that can be transferred in bursts. A row-major order is assumed. Only the Row-Column-wise IDWT that alternately accesses the data in row and in column wise has a bad burst mode usage. A bad burst mode usage reduces the available bandwidth.

For each variant a hardware controller was generated by CLooG-VHDL and combined with a statements entity, constructed using Vim scripts [7, 133] and *Steps2Process*, with or without manual optimiza-

[3]The used input code is not exactly the same as in Chapter 3. The number of statements differs. 1 statement in Chapter 3 corresponds with 2 consecutive statements in the code used here. However, the transformation sequences are equivalent.

Table 5.8: Implementations of the IDWT, without borders. Synthesis results without memories (Figure 5.13 or the right side (clk 2) of Figure 5.18) obtained with Altera QuartusII v6.1 for the Stratix EP1S25F1020C5 (S25 C5) and EP1S60F1020C6 (S60 C6).

(a) Designs made with CLooGVHDL. Manual optimizations on the statements were done in designs 4–6.

Design	Var.	LE	DSP bl. (#Mul)	Cycles (72×88)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
1	RC	3244	18 (9)	187389	56.37	46.20	19.05	15.61
2	LB	3390	18 (9)	200459	52.80	45.60	16.68	14.41
3	LB V	4031	18 (9)	200497	53.67	47.74	16.95	15.08
4	RC	2925	18 (9)	129821	57.67	56.60	28.13	27.61
5	LB	3091	18 (9)	142891	62.56	46.84	27.73	20.76
6	LB V	3155	18 (9)	142929	62.12	56.21	27.53	24.91

(b) Impulse C

Design	Var.	LE	DSP bl. (#Mul)	Cycles (72×88)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
7	RC	16087	80 (10)	605588	34.39	-	3.60	-
8	RC	8006	128 (16)	605588	-	32.93	-	3.44

(c) Designs generated by Handel-C. In designs 13 and 14 an EDIF netlist was generated. In the other designs (9–12) VHDL was generated.

Design	Var.	LE	DSP bl. (#Mul)	Cycles (72×88)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
9	RC	7632	80 (10)	108792	47.02	-	27.37	-
10	RC	1730	144 (18)	108792	-	45.10	-	26.25
11	LB	15179	80 (10)	100356	44.87	-	28.32	-
12	LB	9296	144 (18)	100356	-	41.27	-	26.04
13	RC	3941	0 (0)	108792	70.96	64.30	41.31	37.43
14	LB	5762	0 (0)	100356	66.44	63.65	41.93	40.17

tions. The design flow is shown in Figure 5.19 and synthesis results of all the designs are listed in Tables 5.8 and 5.9. The frame rates assume a computation limited design.

To investigate the influence of the complexity due to the mirroring at the borders, two versions of the code were made. The first does not calculate the pixels near the borders. The second uses predicated statements to distinguish between pixels at the center and near the border. This distinction is done inside the statements (cf. Figure 5.12(c)) and

Table 5.9: Implementations of the IDWT, with borders. Synthesis results obtained with Altera QuartusII v6.1 for the Stratix EP1S25F1020C5 (S25 C5) and EP1S60F1020C6 (S60 C6). (* = Number of cycles for CIF resolution instead of 72×88 pixels)

(a) Designs made with CLooGVHDL. Manual optimizations on the statements were done in design 18, which was extended with a memory hierarchy to obtain design 19.

Design	Var.	LE	DSP bl. (#Mul)	Cycles (72×88)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
15	RC	7278	18 (9)	214269	54.71	50.17	16.17	14.83
16	LB	5589	18 (9)	215848	54.82	49.21	16.08	14.44
17	LB V	5743	18 (9)	215881	53.64	55.88	15.74	16.39
18	LB	10836	18 (9)	161037	50.12	43.40	19.71	17.07
19	LB	17350	18 (9)	*934962	47.22	45.60	50.50	48.77

(b) Impulse C

Design	Var.	LE	DSP bl. (#Mul)	Cycles (72×88)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
20	RC	31288	144 (18)	697431	-	30.60	-	2.78
21	LB	23116	144 (18)	508116	-	38.01	-	4.74

(c) Designs generated by Handel-C. In designs 24 and 25 an EDIF netlist was generated. In the other designs (22 and 23) VHDL was generated.

Design	Var.	LE	DSP bl. (#Mul)	Cycles (72×88)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
22	RC	45722	144 (18)	119712	-	30.81	-	16.30
23	LB	49705	144 (18)	119124	-	31.16	-	16.57
24	RC	11855	0 (0)	119712	53.17	49.90	28.13	26.40
25	LB	12303	0 (0)	119124	52.66	49.90	28.00	26.53

(d) Manual design without (design 26) and with (design 27) memory hierarchy.

Design	Var.	LE	DSP bl. (#Mul)	Cycles (CIF)	f_{max} (MHz) S25 C5	f_{max} (MHz) S60 C6	Frames/s (CIF) S25 C5	Frames/s (CIF) S60 C6
26	RC	1823	10 (5)	*869530	72.54	68.60	83.42	78.89
27	RC	2202	13 (8)	*869530	72.54	68.60	83.42	78.89

not inside the loop control structure. This makes the loop transformations and control complexity similar for both cases and will only have an impact on the statement implementations. Design 15 (Table 5.9(a)) is an exception, built from 80 statements instead of 16, and illustrates

Table 5.10: Insertion of registers between control automata increases the clock speed at the cost of extra clock cycles. Results for the line-based IDWT with borders on a Stratix EP1S25F1020C5. α: buffer signals *t_0*, *for_1_lc*, *start_for_1* between ID0 and for 1. β: buffer signals *t_0*, *t_1*, *for_2_lc*, *start_for_2* between ID1 and for 2. $\alpha + \beta$ corresponds to design 16 in Table 5.9. Number of cycles and execution time T for transforming a frame of 72×88 pixels over 3 levels.

Opt. Regs	LE	Regs	Cycles	f_{max}(MHz)	T(ms)
None	5587	336 (= +0)	215366 (= +0)	53.42	4.03
α	5546	342 (= +6)	215388 (= +22)	57.20	3.77
β	5603	344 (= +8)	215826 (=+460)	53.99	4.00
$\alpha + \beta$	5589	350 (=+14)	215848 (=+482)	54.82	3.94

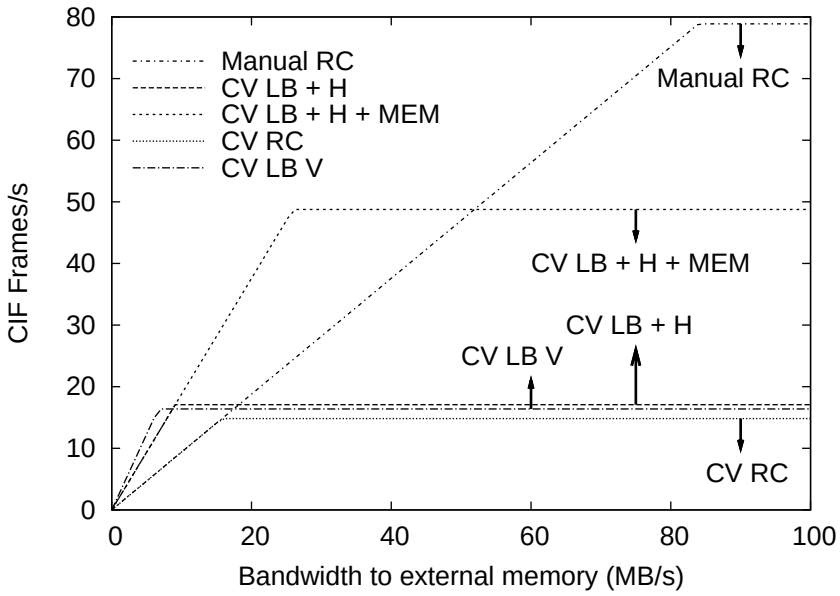
the area advantage of using fewer statements with predicates.

In Section 5.5 the option to insert registers between control automata has already been studied. Experiments on the line-based IDWT (Table 5.10) show that inserting registers between high levels is more beneficial than between low levels. Placing registers at both places results in a summation of the extra clock cycles and registers. However, the clock frequency lies somewhere in between. The designs in Tables 5.8 and 5.9 all have registers at both places (cf. $\alpha + \beta$).

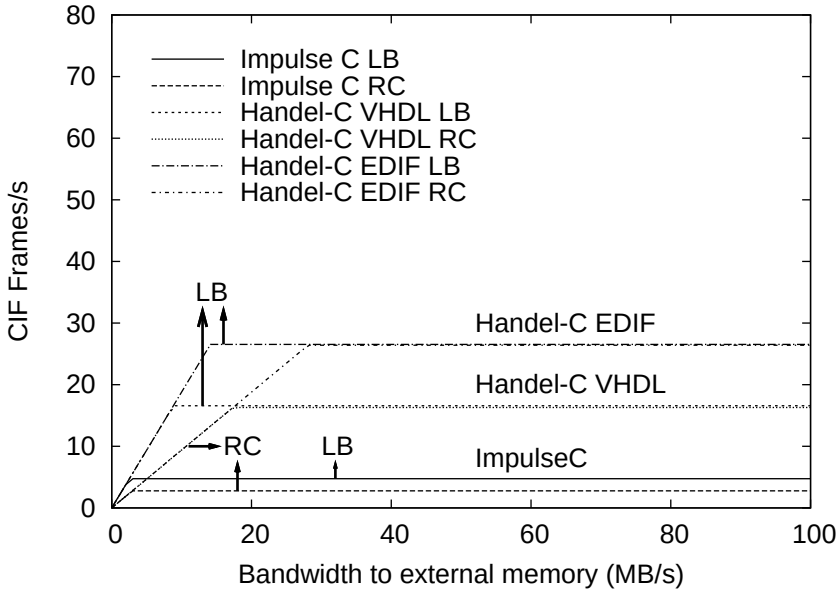
Figure 5.20(a) plots the frame rate as a function of the available bandwidth. When sufficient bandwidth is available, a design is computation limited and the frame rate does not depend on the available bandwidth. For lower bandwidths a design is bandwidth limited and the frame rate is proportional with the bandwidth. For different available bandwidths different designs have the highest performance. The manual design of Section A.3 (design 26-27) reaches a much higher computation limited frame rate, but suffers from its bad spatial and temporal locality. Note that the available bandwidth depends on the burst mode usage. For the RC-wise IDWT this is the harmonic mean of the bandwidths of the horizontal and vertical accesses, and thus dominated by the lowest of the two.

Manual optimizations

As can be seen in Figure 3.22 some elements of array A used on line 7 are already used on line 6 and similarly for array B on line 11 and 10. These non-loop-carried reuses can be exploited by storing the reused



(a) Manual and CLooGVHDL (CV), (H=hand-optimized statements)



(b) Impulse C and Celoxica Handel-C

Figure 5.20: Frame rate as a function of the available memory bandwidth. Designs with borders on the Altera Stratix EP1S60F1020C6.

elements in registers after the first read and using this value in the next statement within that iteration. This reduces the number of memory reads and thus the cycle count.

In the designs described above two memories are used, one for the input frame consisting of the LL_0, HL, LH and HH bands and one for intermediate results (array B and frames LL_l for $l > 0$). These two memories can be accessed in parallel which is only done after optimizing the schedules of the statement implementations.

To shorten the critical path, the computation of the sum of products is spread over several clock cycles, similar to the optimization of the computation in Figure 5.12(c).

These three kinds of optimizations were done manually on the statements of designs 4–6 and design 18.

Only when constructing a memory hierarchy (Section 5.6.3, design 19) the loop-carried reuses will be exploited and the number of external memory reads is further reduced. This is done by introducing registers to exploit reuses in the horizontal filtering, FIFOs to exploit reuses between vertical and horizontal filtering and line buffers to exploit reuses within the vertical filtering.

5.6.2 Comparison of designs generated with CLooGVHDL, Impulse C and Handel-C

The implementations made manually and with CLooGVHDL are compared with designs generated by Impulse C (v. 1.22) [4], a commercial tool for automatic synthesis of stream-based applications (Section 2.2.2), and the Handel-C compiler (v. 3.5.768.63181) of Celoxica DK Design Suite (v. 4.0 SP1) [3] (Section 2.2.4)[4].

In each input language 18-bit data types are used in the datapath. However, the VHDL code generated by Impulse C and Handel-C results in 32-bit multipliers (8 DSP blocks each), while with CLooGVHDL 18-bit multipliers (2 DSP blocks each) are used. It seems that the VHDL code generators of Impulse C and Handel-C just neglect the data type information specified in the input code. The Handel-C compiler can directly generate EDIF-netlists for a target FPGA instead of VHDL. In this case no embedded multipliers but LEs (Logic Elements) are used to implement the multiplication with constants. This has a large impact on

[4]The author would like to thank Tom Degryse for writing the Handel-C variants.

the performance. Two FPGAs were targeted: the EP1S60F1020C6 has 144 DSP blocks, the EP1S25F1020C5 only 80 but it has a higher speed grade. This results in a shift between used LEs and DSP blocks (designs 7–12).

The coding style of Handel-C has a large influence on the resulting hardware. To force the reuse of multipliers a function had to be defined that is called from different parts of the code, e.g., within the predicates to distinguish between center and borders in designs 22–25. Also the reuse of variables and the choice of putting directives for sequential and parallel execution has a large impact and should be done with the resulting hardware in mind. For each loop transformation this work has to be redone, while with CLooGVHDL the statements block can be reused, with the exception of transformation specific manual optimizations. The designs made with Handel-C have the lowest clock cycle count. They exploit parallelism between different statements (and invocations), which is not possible (until now) with CLooGVHDL.

The inclusion of the calculations near the borders clearly adds a lot of complexity to the design. If a processor is available in the system it may be beneficial to put the border calculations on the processor and run only the regular non-border operations on the FPGA. However, adding a processor only for processing the borders could be a large overkill.

Impulse C gives the worst synthesis results. A single large automaton is generated instead of a factorization into smaller automata, leading to a low clock frequency. Also the number of clock cycles is much higher than with the other tools. The states of the control automaton directly correspond to the macro blocks of the input code. Separate hardware for each macro block is generated, which results in a bad area efficiency.

5.6.3 Building a memory hierarchy

From the designs generated in Section 5.6.1 design 18 was selected for a real hardware implementation.[5] To this end, a memory hierarchy with prefetch/store system was built (Figure 5.21, design 19) to replace the simple construct of the two memories with fixed access time.

[5]The bandwidth requirements of RC are too high. LB V can only be used if the number of transformation levels k is fixed at design time. For the scalable video decoder (Section A.3.1) the design was made for, k had to be a run-time parameter.

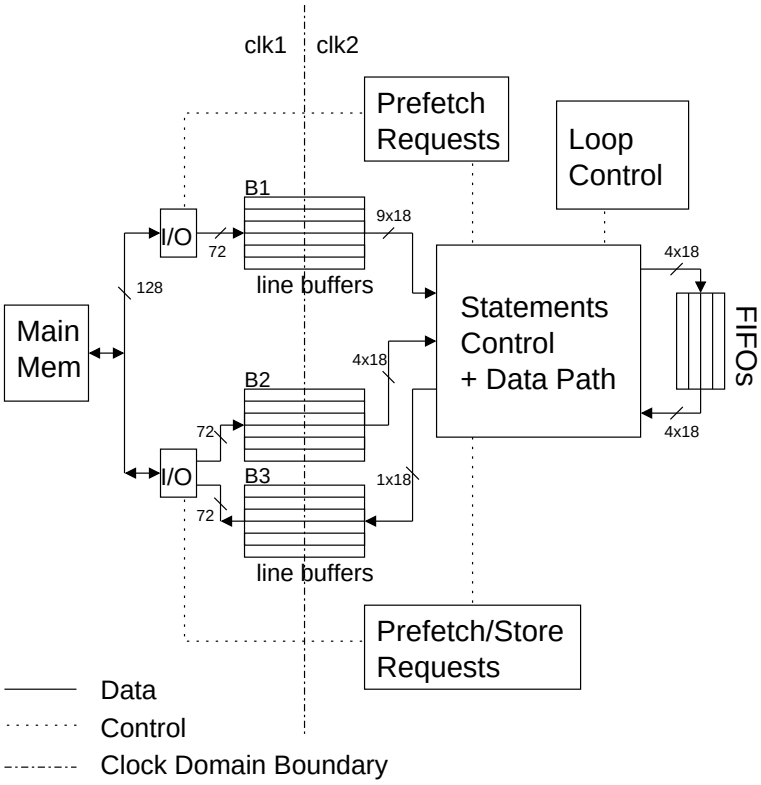


Figure 5.21: Line-Based IDWT with memory hierarchy (design 19).

To exploit the improved data locality several buffers are inserted.

- Registers are used to exploit the loop-carried reuses in the horizontal filtering. Register promotion techniques [51] can be used to do this in a systematic way.
- Four FIFO buffers are inserted to transfer data from the vertical to the horizontal wavelet transformation.
- Line buffers (B1, B2 and B3) are inserted between the main memory and computation elements.
- A block transfer system copies data from the main memories to and from the buffers. A queue of prefetch and store requests is kept in the Prefetch(/Store) Requests entities. A new fetch request is added each time space becomes available in the buffers and not just before the data is needed. Therefore, if the system

is not bandwidth limited, only in the beginning time is wasted waiting for data. A block transfer is specified by the source and target address and the amount of data to be copied (a power of two times the line length in the smallest subband). A DMA controller (Direct Memory Access) is used to drive the block (=burst) transfers [78].

- Synchronization points (using "`-- #waitndo`") are used to ensure that when a line in one of the buffers is accessed, all transfers between that line and the main memory are finished.

- Some line buffers are split into parallel accessible buffers of one line to increase the on-chip bandwidth. This results in a large reduction in the number of clock cycles.

More information on the memory hierarchy construction process with some considerations influencing implementation choices are found in [71].

5.6.4 Performance, power and energy analysis

This design (19) and the manual design (27) were integrated on an Altera PCI Development Board with a Stratix EP1S60F1020C6 FPGA and 256 MiB of DDR SDRAM memory, as part of the RESUME scalable wavelet-based video decoder [70]. An Avalon switch fabric is used to connect the hardware blocks to the external memory, running at 50 MHz.

The partially generated design transforms up to 53 (gray-scale) CIF frames/s (higher than in Table 5.9, due to other tool settings). The manually made design reaches only 11.3 CIF frames/s due to the double amount of transfers to/from the external memory and the large amount of transactions that cannot be done in bursts.

A line-based software implementation reaches 43.5 CIF frames/s on an AMD Athlon XP 2500+ running at 1830 MHz. This corresponds to 42 million cycles for one CIF frame, 50 times more than the optimized FPGA implementations. Note that on the FPGA different tasks can run in parallel with the IDWT while this is not possible on the CPU.

Off-chip memory accesses may not only decrease the performance but may also increase the energy dissipation. Table 5.11 illustrates this by comparing the energy and mean power dissipation of two video

Table 5.11: Comparison of two decoders, each using one variant of the IDWT. Sequence of 10 GOPs (161 CIF color frames YUV 4:2:0) of Foreman at full quality.

Variant	Time (s)	E (J)	P_{mean} (W)
RC IDWT (Manual)	40	25.4	0.64
LB IDWT (CLooGVHDL + ManOpt)	10	11.6	1.16

decoders, each using one of the two IDWT variants mentioned above. More information on the measurement setup and detailed results can be found in [70, 81].

5.7 Discussion and conclusions

In this chapter a novel hardware architecture is presented that fits the polyhedral model. Its modular composition allows to make trade-offs between area and clock speed and supports extensions towards parallel execution. Integrating the application of loop transformations and hardware generation makes design space exploration more feasible.

Experiments with two variants of the IDWT showed that a semi-automatically generated (over-sized) implementation can outperform a manual design in speed and energy dissipation if bandwidth is a limiting factor. Obviously, doing loop transformations at the start of a manual design process would have led to better results. However, in this case doing a search space exploration would have been much harder.

We have shown that loop transformations not only influence the data locality but also the control complexity. In some cases optimizations on the loop bound expressions generated by CLooG are possible and needed. For this we have presented techniques that are an extension of existing loop bound simplification methods.

Is it a good choice to divorce the loop control from the statement implementations? An argument pro is the orthogonal development of both entities, which allows to combine variants of both and explore the design space when the generation of one of both is not fully automated yet. An argument contra is that no combined optimizations are possible. Certainly in the case considered until now, sequential execution of statement invocations, this is a large drawback. Only the parallelism

within statement invocations can be exploited (cf. the lower cycle count reached with Handel-C). This only gives good results for applications with *large statements*. Hence, extending the presented work to include parallelism among statement invocations is a must.

An option not considered here is to use one of the C-based hardware description languages, described in Chapter 2, as output. A C to Handel-C converter has already been described in [134]. Since a lot of user directives are needed and coding style has a large influence on the final result, it is not sure if this would have been much easier.

A restriction of many high-level synthesis tools is that they can deal well with the macros and corresponding hardware constructs they offer in their library, but the inclusion of user made blocks has to be specified as communication with an external block. Such a block then has to fit a certain interface or the communication should be described at a lower level. An ideal high-level synthesis tool should offer the option to extend the vendor library with user constructs. Such new constructs should then be specified as

- a C-like construct and behavior,
- a VHDL or hard core implementation,
- schedule or synchronization information,

such that it can be optimally integrated in the schedule process.

Vim scripts, although primitive by nature, offer this extendability in a simple way. The path to a first implementation is not much smoother, but the scripts work as a kind of memory for manual optimizations. They make iterations more feasible where tool support is lacking.

Synthesis tools that try to optimize the clock speed are conservative in their approach. Retiming and other techniques like duplication of registers will never change the *behavior* of a circuit. Here, behavior is defined in a strict sense including clock cycle count. In the proposed factorized architecture the insertion of extra registers at certain places is allowed, even though this changes the cycle count. This could be used in synthesis tools to extend the current clock optimization methods.

In this chapter we have presented methods to more easily explore the design space by shortening the path to a certain point (implementation) in that space. Besides this, methods are needed that aid in predicting the results of design choices, such that less points in the design space really have to be visited.

Chapter 6

Conclusions and future work

In this chapter the conclusions drawn from this dissertation are summarized. Directions for future research based on the contributions of this work are highlighted.

6.1 Conclusions

Many methods and tools have emerged that automate part of the tasks done by a hardware designer. Nowadays, register-transfer-level specifications are considered as synthesizable. High-level synthesis tools try to further raise the abstraction level of hardware descriptions. Typically, to have good results, the user still has to be aware of the architectural choices made by such tools. However, if we want to fully benefit from the computational power that future technologies will offer, there is no alternative to raising the abstraction level to avoid a design bottleneck.

A high computational power is useless if time is wasted waiting for data transfers. Loop transformations are indispensable to tackle the memory bottleneck, both for software and hardware implementations. The polyhedral model offers a perfect means to automate the application of loop transformations. However, the choice of which transformations to apply is much less straightforward to automate. Two trends exist. The first tries to iterate over a large number of possible transformations, and evaluates the result of each transformation to guide the

further exploration. Heuristics are needed to prune the search space to be scanned. The second uses directives delivered by the user. These may be pragmas or scripts composed of loop transformation steps. In the context of the latter, this dissertation presented the option to alleviate the task of constructing long transformation sequences by combining primary (sub)sequences.

Bounds on (piecewise) (quasi-)polynomials over discrete domains are needed to statically evaluate properties of a program obtained after loop transformations. To this end, several novel methods to find such bounds are presented in Chapter 4 and compared with existing methods. The presented methods take advantage of the fact that for large domains or small degrees the continuous-domain extrema of polynomials are a good approximation of the discrete-domain extrema. For small domains the straightforward method of evaluating the (quasi-) polynomial in each point is still faster, but this solution does not scale for larger domains. We have proved that with a simple selection mechanism, which introduces almost no overhead, hybrid methods can be constructed that combine the strengths of different methods. With more complex selection mechanisms less additional benefit is expected. Experiments demonstrate that the techniques can be used to evaluate properties of algorithm implementations with different schedules and thus can be used to guide the loop transformation process.

The class of applications that benefit from implementation on a FPGA corresponds well with the class of applications that can be described within the polyhedral model. Since loop transformations not only influence the data access pattern but also the control complexity, integration of loop transformations and hardware generation is needed to speed up the design space exploration. Therefore, we have presented a novel hardware architecture with close correspondence to the polyhedral representation. The architecture consists of a loop controller entity, composed of different automata, and a statements entity. With CLooGVHDL a hardware description of such a loop controller is generated directly after applying loop transformations. The factorized implementation offers several options for area-speed trade-offs, allows independent optimization of both entities, and supports the extension towards a parallel implementation. The IDWT case study has proved that the integration of loop transformations and hardware generation indeed eases the design space exploration.

6.2 Directions for future research

Every answer to a question raises ten new questions to answer. As a consequence, the conclusions have to be followed by new research questions.

A lot of work is still needed before loop transformations will be fully automated. It is not clear if the solution will exist of a set of transformations applied in sequence or of more abstract methods that try to find the optimal schedule functions. Trying to automate the combination of primary sequences already seems a challenging task.

The arise of divisions with a high implementation cost in code generated from the polyhedral model points to the fact that the current code generation tools are not optimal yet. In this thesis some methods to simplify loop bounds were presented and applied manually. Automation of these methods and integration with code generators is not done yet. However, probably it is possible to avoid the generation of such expressions (at least in cases similar to those that have arisen in this thesis) by adapting the way in which the code is generated, instead of adding a postprocessing step. This would be a cleaner solution.

The proposed methods to find bounds on quasi-polynomials only aid in evaluating a design property. A next step is the integration in methods to guide a search space exploration.

Bernstein expansion offers the possibility to deal with parametric polynomials. This option has not been used yet in the context of the presented work. For nonparametric problems Bernstein expansion may lead to large overestimates. Subdivision of the domains may improve the accuracy but does not look trivial for arbitrary polyhedral domains. Alternatives for the Bernstein expansion may also be considered.

The tools described in Chapter 5 are only a small step towards fully automated hardware generation. Parallelism between statement invocations has to be supported. For the construction of the data path a real input parser and scheduler are needed, together with methods to construct a memory hierarchy that supports the available parallelism at the lower levels. The impact of the choice to split loop control from data path has to be investigated further. Possibly, hybrid architectures may be constructed.

Retiming methods may be extended to allow the insertion of registers at some predefined places. This would relieve the constraint of preserving clock cycle true behavior.

When using a high-level synthesis tool, interfaces are mostly restricted to those offered by the vendor, or have to be implemented at a lower level, which removes the advantage of using such tools. The development of an extendible high-level synthesis tool would be an option. In such a tool user built constructs and vendor library blocks would be dealt with on equal terms.

A lot of the options mentioned above try to shorten the time to reach a certain point in the design space. However, complementary methods are needed that estimate the impact of design choices without actual implementation. As a result, fewer points in the design spaces would have to be visited.

Appendix A

The discrete wavelet transform and its inverse

This appendix provides more details on the Discrete Wavelet Transform (DWT) and the Inverse Discrete Wavelet Transform (IDWT). The focus is on the implementation aspects. For more information on where, why and how wavelets are used the reader is referred to [48] and [121].

A.1 The 1-D discrete wavelet transform

A.1.1 Filters

A *filter* F is a linear time invariant operator, which is completely determined by its impulse response $\{f_k \in \mathbb{R} \mid k \in \mathbb{Z}\}$. The output signal b of a filter is obtained as the convolution of the input signal a and the impulse response f:

$$b_n = (f * a)_n = \sum_{k=k_b}^{k_e} a_{n-k} f_k \ , \tag{A.1}$$

with k_b and k_e the smallest and largest integer number for which f_k is non-zero[1]. F is causal if $k_b \geq 0$. The filter is a Finite Impulse Response (FIR) filter in case only a finite number of filter coefficients f_k is non-zero. The Z-transform of a FIR filter F is a Laurent polynomial $F(z)$

[1] k_b may be smaller than zero (non-causal filter). This avoids the use of an arbitrary time shift (factor z^i) in Equation A.5 and A.6. In a physical implementation all filters are made causal by a time shift of the impulse response.

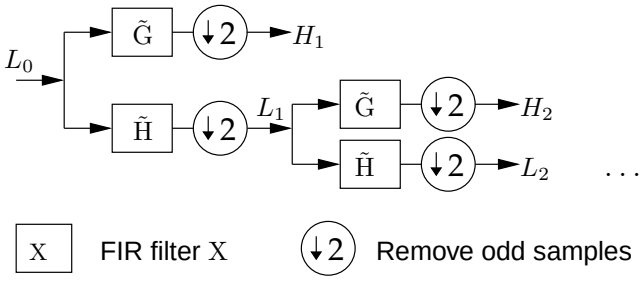


Figure A.1: Block diagram of the 1-D Discrete Wavelet Transform (1-D DWT).

given by

$$F(z) = \sum_{k=k_b}^{k_e} f_k z^{-k} \quad . \tag{A.2}$$

The notations F, F and $F(z)$ are often used to denote both the filter and the associated polynomial.

If l denotes the length of a filter ($l = k_e - k_b + 1$), then the computation of one element of the filter output requires l multiplications and $l - 1$ additions. We will further assume $k_b = 0$ and $k_e = l - 1$. For a symmetric causal filter ($f_k = f_{l-1-k}$, $k = 0, 1, \ldots, l-1$) formula A.1 can be rewritten as

$$b_n = \sum_{k=0}^{l-1} a_{n-k} f_k \tag{A.3}$$

$$= \begin{cases} \sum_{k=0}^{\frac{l}{2}-1} (a_{n-k} + a_{n-l+1+k}) f_k & \text{if } l \text{ is even} \\ \sum_{k=0}^{\frac{l-1}{2}-1} (a_{n-k} + a_{n-l+1+k}) f_k + a_{n-\frac{l-1}{2}} f_{\frac{l-1}{2}} & \text{if } l \text{ is odd} \end{cases} \tag{A.4}$$

which uses only $\lceil \frac{l}{2} \rceil$ multiplications and $l - 1$ additions. For antisymmetric filters ($f_k = -f_{l-1-k}$) the same number of operations applies. Symmetric and antisymmetric filters are *linear phase* filters.

A.1.2 The convolution-based 1-D discrete wavelet transform

The general block diagram of a forward wavelet transform is shown in Figure A.1. The input signal, L_0, is sent through two FIR filters, a low-pass filter $\tilde{H}$ and a high-pass filter $\tilde{G}$, called the analysis filters. The

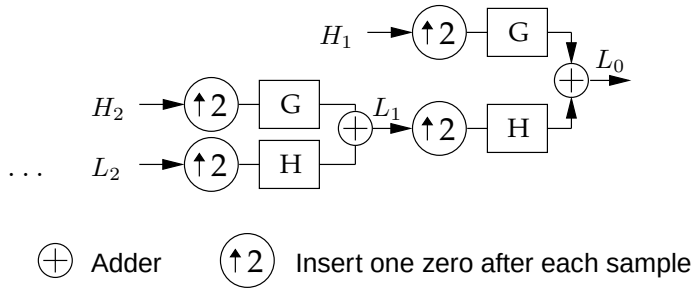


Figure A.2: Block diagram of the 1-D Inverse Discrete Wavelet Transform (1-D IDWT).

results are subsampled by a factor two (removing the odd samples), resulting in the signals, H_1 and L_1, the first level subbands of the wavelet transform. This process is repeated on the L subband of each level until the kth level is reached. The Inverse Discrete Wavelet Transform (IDWT) uses the synthesis filters H and G, preceded by upsampling with a factor two (inserting a zero after each sample) (Figure A.2). The conditions (necessary and sufficient) for perfect reconstruction of the original input signal are given by

$$H(z)\tilde{H}(z^{-1}) + G(z)\tilde{G}(z^{-1}) = 2 \tag{A.5}$$

$$H(z)\tilde{H}(-z^{-1}) + G(z)\tilde{G}(-z^{-1}) = 0 \ . \tag{A.6}$$

The symbols A, a or s (approximation, signal) and D or d (detail, difference) are sometimes used instead of L and H. These names refer to an important property of the wavelet subbands. The L subbands form a low-resolution approximation of the original signal. This approximation can be made more accurate by adding information from the detail signals.

Example 12. *The Haar Wavelet is the simplest example of a wavelet transform. The analysis filters are given by*

$$\begin{aligned} \tilde{H}(z) &= \frac{1}{2} + \frac{1}{2}z^{-1} \\ \tilde{G}(z) &= -1 + z^{-1} \ . \end{aligned}$$

Let $L_0 = (a_0, a_1, a_2, \ldots)$*, then*

$$L_1 = \left(\frac{a_1 + a_0}{2}, \frac{a_3 + a_2}{2}, \ldots \right)$$

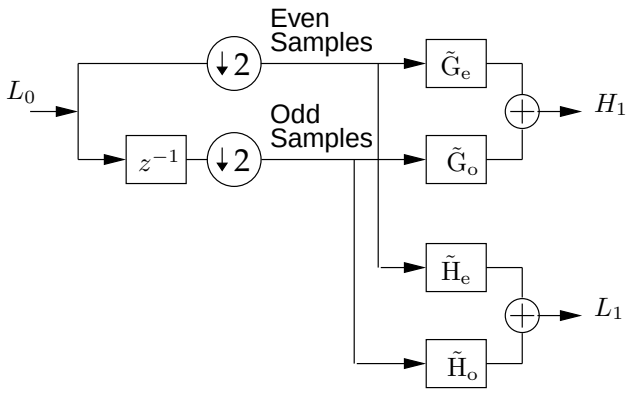


Figure A.3: Polyphase decomposition of the analysis filters to avoid the computation of results, removed by subsampling.

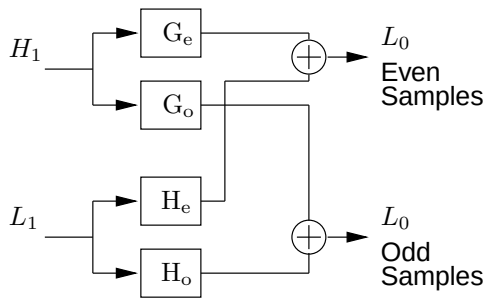


Figure A.4: Polyphase decomposition of the synthesis filters to avoid multiplications with zero.

and

$$H_1 = (a_1 - a_0, a_3 - a_2, \ldots)$$

are simply the means and the differences of each pair of input samples. It is easy to see that L_0 can be reconstructed using the synthesis filters

$$\begin{aligned} H(z) &= 1 + z^{-1} \\ G(z) &= -\frac{1}{2} + \frac{1}{2}z^{-1} \quad . \end{aligned}$$

A straightforward implementation of the diagram in Figure A.1 and A.2 leads to superfluous operations. Half of the filtering results are thrown away by subsampling, and half of the multiplications in the filters after upsampling are multiplications with zero. A filter can be

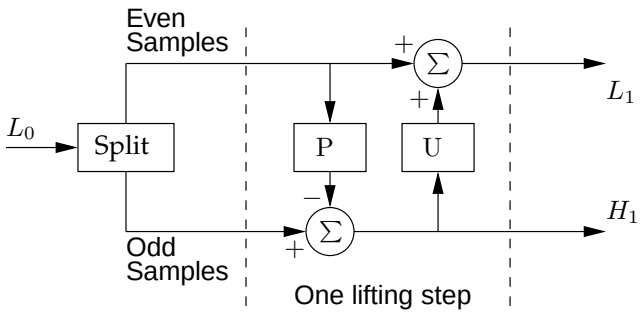


Figure A.5: Lifting scheme of the DWT with one lifting step.

decomposed into an even and an odd phase as follows:

$$F(z) \quad = \quad F_e(z^2) + z^{-1}F_o(z^2) \ . \tag{A.7}$$

Using the even and odd phases of the wavelet filters and splitting the signals into even and odd samples, the number of operations can be reduced. This results in the diagrams of Figure A.3 and A.4.

For the Haar filter (Example 12) this becomes:

$$\tilde{H}_e(z) = \tilde{H}_o(z) = \frac{1}{2} \ , \quad \tilde{G}_e(z) = -1 \ , \quad \tilde{G}_o(z) = 1$$
$$H_e(z) = H_o(z) = 1 \ , \quad G_e(z) = -\frac{1}{2} \ , \quad G_o(z) = \frac{1}{2} \ .$$

A.1.3 The lifting scheme

The number of operations needed to perform a DWT can in many cases be reduced by reusing subexpressions. This is done in the *lifting scheme*. A simple instance is shown in Figure A.5. First, the even and odd samples are split, as in the left side of Figure A.3. This splitting is called the *lazy wavelet transform*. If the input signal has a local correlation structure, the even and odd subsets will be highly correlated. Given one of the two sets, it should be possible to predict the other one with reasonable accuracy. The odd samples are predicted by filtering the even samples with a predict filter P. In the case of the Haar wavelet $P(z) = 1$. The L-subbands have the property that their average is the same as for the original signal ($\sum_k h_k = H(1) = 1$). This is ensured by the update step with the update filter U. For the Haar wavelet $U(z) = \frac{1}{2}$. For longer wavelet filters the lifting scheme consists of multiple lifting steps. A detailed explanation of the lifting scheme can be found in [60, 154]. Note

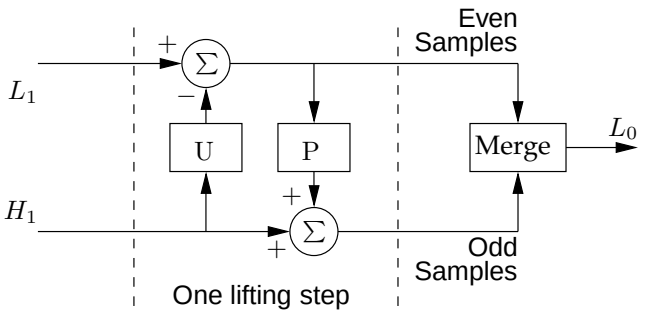


Figure A.6: Lifting scheme of the IDWT with one lifting step.

that the decomposition of a wavelet transform into lifting steps is not unique.

A lifting scheme can always be inverted. It suffices to undo the update and predict steps and merge the even and odd samples (Figure A.6).

A.1.4 Computational complexity

For the convolution-based DWT the number of operations is easily derived by counting the operations of the FIR filters as shown in Table A.1. The notations l_G and l_H are used for the lengths of the (synthesis) wavelet filters. After adding tildes, the equivalent expressions in $l_{\tilde{G}}$ and $l_{\tilde{H}}$ for the analysis filters are obtained. For Daubechies filters $l_G + l_H \in 4\mathbb{Z}$. Therefore, the filters have both even or both odd length.

A.2 The 2-D discrete wavelet transform

The 2-dimensional DWT (Figure A.7) consists of consecutively applying the 1-D DWT in the horizontal and in the vertical direction. First, each row of the original image (LL_0) is transformed horizontally resulting in the subbands L_1 and H_1. Next, this result is filtered vertically in a similar way to generate the four subbands LL_1, LH_1, HL_1 and HH_1. The same steps are repeated recursively on each LL-subband until level k. This variant of the DWT is called level-by-level, row-column-wise (RC-wise). In a similar way the DWT can be defined for higher dimensions [112, 146]. Analogously, the inverse transformation performs a 1-D IDWT along the columns and the rows of the different subbands.

Table A.1: Number of operations of the 1-D DWT or IDWT for each 2 samples. The formulas assume lifting steps of degree one, i.e., predict and update filters with length two. In particular cases, the numbers can differ slightly as shown for the (9,7) filter pair. l_H and l_G denote the lengths of the analysis or synthesis wavelet filters. Formulas and numbers based on [60, 106].

		Convolution-based		Lifting-based	
		Mul	Add	Mul	Add
FIR Filter	General	l	$l-1$		
	(Anti-)symmetric	$\lceil \frac{l}{2} \rceil$			
Daubechies Wavelets	General	$l_H + l_G$	$l_H + l_G - 2$	$\left\lceil \frac{l_H+1}{2} \right\rceil + \left\lceil \frac{l_G+1}{2} \right\rceil + 3$	$\left\lceil \frac{l_H+1}{2} \right\rceil + \left\lceil \frac{l_G+1}{2} \right\rceil + 1$
	Odd, Linear phase	$\frac{l_H+l_G}{2} + 1$		$\frac{l_H+l_G}{4} + 1$	
	Even, Linear phase	$\frac{l_H+l_G}{2}$		$\left\lceil \frac{l_H+1}{2} \right\rceil + \left\lceil \frac{l_G+1}{2} \right\rceil + 3$	
Haar		1	2	1	2
Daubechies	(9,7)	9	14	6	8

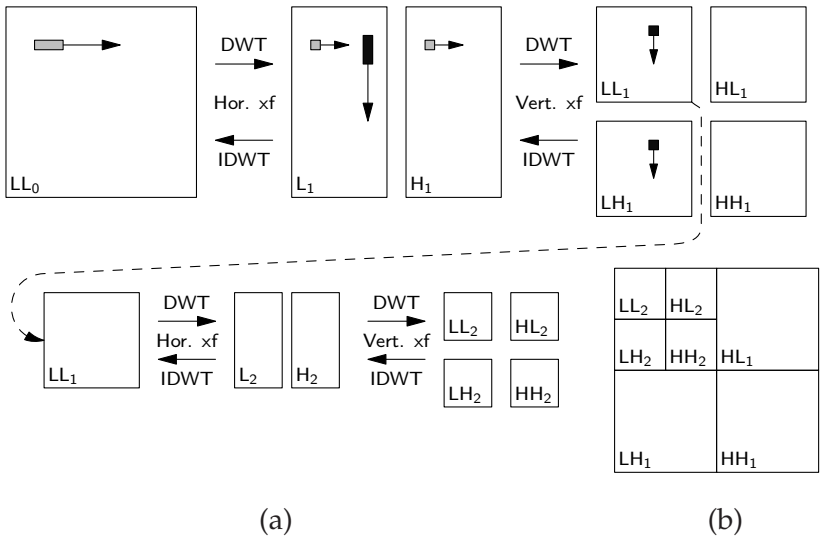


Figure A.7: Graphical representation of the 2-D (I)DWT over 2 levels. The 2-D DWT first transforms the original image horizontally, line by line. The gray box in LL_0 represents the pixels that are read to calculate the elements indicated by the gray boxes in L_1 and H_1. The arrows indicate the scanning direction and their sizes indicate the scanning rate. This rate differs by a factor 2 due to the subsampling. Next, the results are filtered vertically, column by column as indicated by the black boxes. These two steps are repeated on the LL_1 subband. The resulting subbands can be stored together within one data set with the same dimensions as the original frame (b). The 2-D IDWT consists of doing the inverse of each step.

Figure A.8 shows the DWT applied on the Lena image, a standard reference image for image processing. To remove the negative values in the LH, HL and HH subbands the absolute value is taken. Vertical lines are visible in the HL subbands and horizontal lines in the LH subbands corresponding with spatial high-frequent components in the horizontal and vertical direction, respectively.

Code for the 2-D inverse transformation in the specific case of a (9,7) filter pair ($l_G = 9$, $l_H = 7$) is shown in Figure 3.22. Corresponding to Figure A.4, the code in Figure 3.22 contains lines for the even and lines for the odd elements.

Some indices get values outside the image boundaries. A way to extend the image is needed. This can be done by adding zeros, mir-

(a) (b)

Figure A.8: The Lena reference image before (a) and after (b) the discrete wavelet transform over 2 levels. In (b) the absolute value of the wavelet coefficients is used.

roring the pixels near the border or extending the image with a linear combination of the pixels near the border [113].

A.2.1 Computational complexity

The number of operations needed to transform an image with size $R \times C$ over k levels is given by

$$\frac{\#oper.}{frame} = \sum_{i=1}^{k} \frac{1}{4^i} \left(AR\frac{C}{2} + AC\frac{R}{2} \right) = ARC\frac{1-(\frac{1}{4})^k}{\frac{3}{4}} \quad , \tag{A.8}$$

with A the number of operations in the 1-D DWT needed for each two samples, depending on the filter lengths and the kind of operation (Table A.1).

When the (I)DWT is included in a multimedia system and has to transform a stream of images, the required number of operations per second is calculated by multiplication with the framerate, i.e. the number of frames per second. If color images represented in the YUV 4:2:0 format are used a factor 3/2 is added since the pixel luminance (Y frames) is given at full resolution while the chrominance information (U and V frames) is only available at a quarter of the resolution (1 +

1/4 + 1/4 = 3/2). This results in

$$\frac{\#oper.}{s} = \frac{3}{2}\frac{\#oper.}{frame} \times framerate \ . \tag{A.9}$$

A.3 Manual design of a RC-wise IDWT

A.3.1 Specifications

Part of the work presented in this thesis has been done within the scope of the RESUME project (Reconfigurable Embedded Systems for Use in scalable Multimedia Environments, 2003–2006, IWT grant 020174, [9]). An overview of the project achievements is found in [80].

This project investigated, a.o., the usefulness of reconfigurable hardware (FPGAs) for decoding scalable video. *Scalable video* means that the Quality of Service (QoS), i.e. the image quality, frame rate, resolution and color depth of the decoded video, can be freely adapted without having to re-encode the video stream or having to decode the whole video stream if only a lower quality version is required. As a result, the video server only has to encode and store a single video stream for all different kinds of decoding platforms which can have varying computational power, network bandwidth or screen types.

Within the RESUME project a scalable wavelet-based video decoder has been built with the requirements to decode a video with CIF resolution (288×352 pixels, YUV 4:2:0 format) in real-time, i.e. 30 frames/s (originally, later lowered to 25 frames/s). An overview of the used codec algorithm is found in [77, 80]. This led to the following design specifications for the IDWT: an IDWT with a 9/7 filter pair, able to transform 45 gray-scale CIF frames/s over 3 levels (more or less equivalent to 30 color frames/s using YUV 4:2:0). Within these constraints minimal area is pursued.

The target platform was an Altera PCI development board with a Stratix S25F1020C5 FPGA, which contains roughly 25000 LEs and 80 DSP blocks (9-bit equivalents).

A.3.2 Modular design concept

To make the design reusable on different HW platforms a modular design concept is used (Figure A.9). The assumption is made that the used

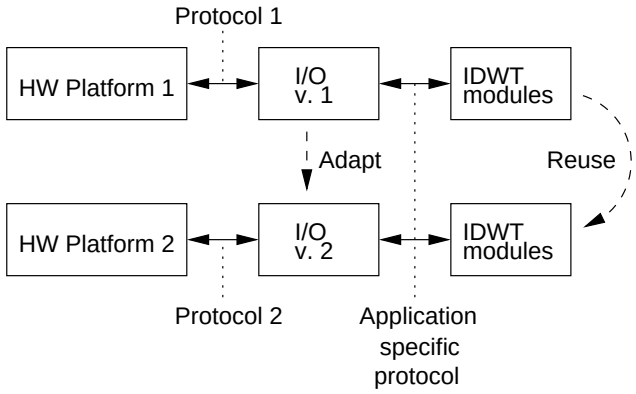


Figure A.9: Using separate I/O-modules makes it easy to put a design on different platforms with different interfaces.

HW platform (or a potential future platform) offers a way to plug-in IP-cores or other HW modules when they satisfy a given interface (IF). Examples are the open standard AMBA® protocol [17] and the Avalon® Memory-Mapped interface used by the Altera SOPC builder [15].

This interface is platform dependent. Therefore, a core has to be adjusted to the platform it is plugged into. With the introduction of platform specific I/O-modules that translate a platform independent, but application specific protocol into the platform specific protocol, the needed adjustments are localized and the functional modules, here the IDWT, can be reused without changes.

More information on the used design flow is found in [69].

A.3.3 Architecture

The architecture is shown in Figure A.10. Some small control ports and parameter inputs, such as the start and ready signal, R, C, k and the filter coefficients, are not shown.

From the design specifications the required number of operations per second is estimated using formula A.9 and is slightly less than 54 million multiplications and 84 million additions per second. Assuming a clock frequency of 50 MHz (proved feasible for other designs) this should be reachable with two multipliers. In the final design five multipliers are used as explained below.

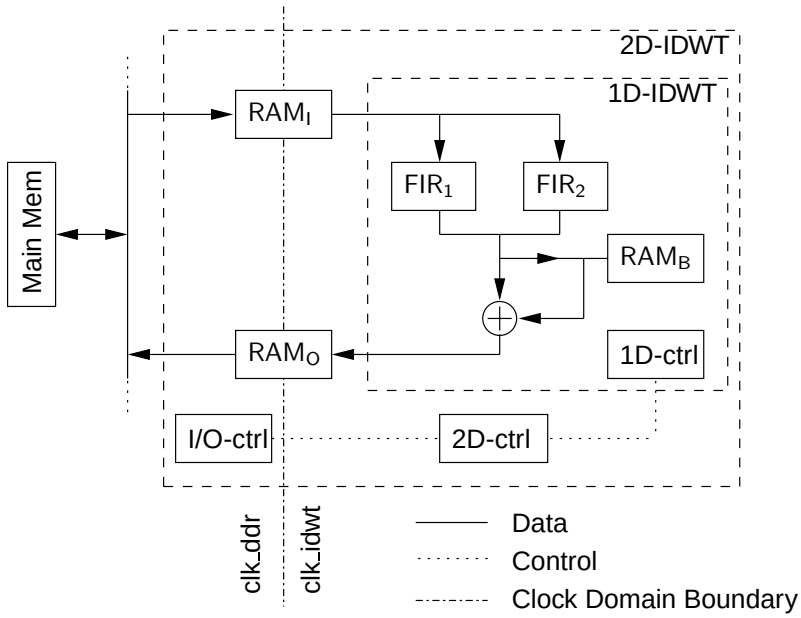


Figure A.10: Architecture of the manually designed IDWT.

The used 9/7 filter pair leads to the following filter lengths:

$$l_G = 9,\ l_H = 7\ ,$$

$$l_{G_e} = 5,\ l_{G_o} = 4,\ l_{H_e} = 4,\ l_{H_o} = 3\ .$$

Two FIR filters (HW modules) are used. FIR_1 has length 4 and implements G_o and H_e, successively. FIR_2 has length 5 and implements G_e and H_o (extended with a 0 in front and at the end to have length 5). Both FIR filters are symmetric and thus take 2 and 3 coefficients, respectively. The filter coefficients can be altered at run-time. In a first execution phase, the L samples are filtered with H_e and H_o and the results are stored in RAM_B. In a second phase, the H samples are filtered with G_e and G_o and the results are added to the results stored in RAM_B. The filter coefficients of G and H are thus loaded alternately into the two FIR filters. This scheme (a 1-D IDWT) is repeated for every column and then for every row and this for all transformation levels.

The in- and output buffers, RAM_I and RAM_O, can store 2 rows or columns of a frame or subband (Figure A.11(a)). These buffers are two-port memories. Therefore, transactions to the main memory can take place in parallel with accesses of the calculation units. After the first

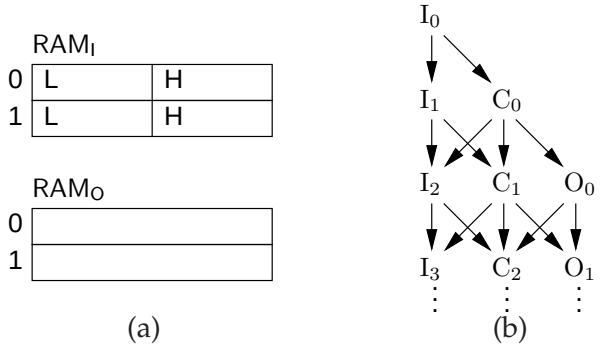


Figure A.11: I/O buffers and partial ordering of memory transactions (I and O) and calculations (C).

column (row) (numbered 0) is fetched into line 0 of $\mathsf{RAM_I}$ (transaction I_0), the computations can start (C_0) while the second column (row) is fetched into line 1 (I_1). The next column (row) can only be fetched into line 0 (I_2), if this line does no longer contain needed data, i.e., if C_0 is finished. For the output a similar dependence exists. This leads to a partial ordering as shown in Figure A.11(b).

To implement the mirroring at the borders, needed to have enough input samples as mentioned on page 173, the address counters first count down a few samples, then up from the beginning to the end of the buffered column (row) and then a few samples down again.

Synthesis results are found in Table 5.9, design 26 (without I/O modules) and design 27 (with I/O modules for direct connection to a single memory with a one-cycle access time).

A.4 Image quality

If data types with infinite accuracy are used the result of doing an IDWT after a DWT is the original image. However, when the word length is limited differences may arise which lower the *quality* of the reconstructed image. A very common quantity to measure image quality is the PSNR (Peak Signal to Noise Ratio).

A.4.1 PSNR

For an image with $R \times C$ pixels, denoted as a_{ij}, the Signal to Noise Ratio (SNR) is expressed as

$$SNR = 10 \log \left(\frac{\frac{1}{RC} \sum_{R,C} a_{ij}^2}{\frac{1}{RC} \sum_{R,C} (a_{ij} - b_{ij})^2} \right) (dB) \ , \tag{A.10}$$

with b_{ij} the pixels of the original (error-free) image. The denominator is the mean quadratic error and the numerator the mean signal energy. This is a measure for the relative error. In most cases a measure for the absolute noise value or error is needed. If the signal value is replaced by its maximal possible value (peak), 255 if the range is $[0, 255]$ or $[-255, 255]$, then the definition of PSNR is obtained:

$$PSNR = 10 \log \left(\frac{255^2}{\frac{1}{RC} \sum_{R,C} (a_{ij} - b_{ij})^2} \right) (dB) \ . \tag{A.11}$$

The PSNR is an objective measure for image quality but does not tell anything about the subjective quality as perceived by a human observer.

A.4.2 From floating-point to fixed-point

The wavelet transform used in this dissertation was originally written in software using floating point data types. For a hardware implementation a conversion to fixed-point was needed. Using 18 bit for the wavelet coefficients (4 decimals) resulted in perfect reconstruction.

A.4.3 Quality scalability

One of the QoS scalability axes of the RESUME codec is the number of bits used to represent wavelet coefficients. This allows to make a trade-off between the computation power needed, the used network bandwidth and the image quality, expressed as PSNR.

A.5 Related work on DWT variants

Most IDWT variants discussed in Section 3.8 are similar to hardware architectures reported in the literature. Often different names for the

Table A.2: An overview of DWT variants found in the literature

Variant	Specification	Reference
RC		[106, 16, 116]
LB	level-by-level multi-level	[116] [54, 167, 106, 53, 16, 116]
BB	BB_{RC}	[167, 16, 116]
Stripe-based	LB V (overlap)	[106]
Others	Pseudo-fractal	[115]

same variant are found. In [106] an overlapped and non-overlapped stripe-based scan method are proposed. This corresponds to an (I)DWT tiled along one dimension. Line-based and block-based architectures have been the subject of intensive research [54, 167, 106, 53, 16, 116]. The reports illustrate how the buffer size, throughput, latency, energy, ... are determined by the data access pattern and thus by what is called above *variant*.

The usefulness of a variant does also depend on other blocks in the system. It is usually beneficial to adjust the data production and consumption pattern of successive modules to each other. By doing this Lafruit et al. [115] create a 2-D pseudo-fractal space-filling scan curve. This schedule is not linear and cannot be reached by doing loop transformations. This illustrates one of the limitations of the polyhedral approach.

To our knowledge none of the articles cited above use high-level loop transformation tools in their implementation path. There exist tools [73, 151] that apply a limited set of loop transformations to explore the area-speed trade-off curve of hardware implementations. However, these transformations work at a lower level and can only explore different implementations of one variant.

Appendix B

Reuse distances of IDWT variants

This appendix contains the results of the reuse distance measurements of the IDWT variants created in Section 3.8 and listed in Table 3.6.

SLO [39] (Section 3.2.2) was used for the reuse distance measurements. By default SLO measures the reference distance, the number of accesses between two accesses of a reuse pair, instead of the number of different elements accessed.[1] Therefore, the environment variable `RDLIB_MEASURE_TRUE_REUSE_DISTANCE` was set. The measurements are performed while transforming an image of 72×88 pixels over 3 ($= k$) levels.

By accumulating the reuse distances larger than a certain cache size one gets the number of misses of a cache with that size using the LRU replacement policy. This is done to generate Table B.1, from which Figure 3.26 is derived.

Note that SLO counts multiple references for multidimensional arrays. E.g., the access to `A[i][j]`, is counted as a read from `A[i]` (pointer to `A[i][0]`) followed by the access to `A[i][j]` itself, leading to a count of two accesses. This can be avoided by mapping multidimensional arrays to one-dimensional arrays.

[1] Sometimes the term *stack distance* is used to denote the true reuse distance, while *reuse distance* is then used to denote the reference distance [38].

Table B.1: Cache misses based on reuse distances of IDWT variants measured by SLO, expressed as the number of misses assuming the LRU cache policy for a cache with size $2^i - 1$.

i	RC	RC H	RC HV	RC V	RC VH	VH LB
0	931998	931998	931998	931998	931998	931998
1	852582	852262	849903	851486	847388	854033
2	726595	722092	721674	721320	728181	730556
3	291168	288294	287064	290335	284615	284972
4	230320	233585	233728	232979	233377	233468
5	26613	29010	32336	29935	32911	50745
6	26298	28706	29602	27187	30176	42393
7	25671	28110	25423	22903	26069	41241
8	22579	25103	14992	12322	14965	36362
9	10202	12731	14213	11845	14267	15660
10	9690	12079	12991	11032	13006	12595
11	8954	10981	10695	9106	10801	10876
12	7060	9222	7152	6913	7236	6758
13	4868	6035	2114	2103	2120	2008
14	489	0	0	0	0	0

i	LB	LB H	LB HV	LB V	LB VH
0	931998	931998	931998	931998	931998
1	853202	848196	846701	851340	853636
2	727883	725639	728090	725780	725317
3	291010	287006	285132	291080	288962
4	238930	232089	234891	239041	231493
5	90660	67602	66120	90650	64154
6	38244	43059	43230	38262	41407
7	37907	41176	41506	37955	40493
8	36895	36064	37134	36982	35944
9	33646	14871	16617	34242	16685
10	25102	14409	14521	26942	15571
11	1881	13124	12456	4678	11899
12	1642	8812	8831	2754	5571
13	1119	6508	6748	144	774
14	211	0	0	0	0

Appendix C

Bernstein expansion

C.1 Parametric Bernstein expansion

In the experiments in Chapter 4, Bernstein expansion is used to find bounds on polynomials over continuous domains. A tutorial on how Bernstein expansion works and what it is used for is given by [88]. Also [56] contains a clear explanation with examples. In this paper Clauss et al. have shown how to extend Bernstein expansion to arbitrary linearly parameterized polytopes. In this section parametric Bernstein expansion is explained with the notation and naming conventions used in this thesis and illustrated on a small example.

Given a parameterized polytope

$$P_p = \{\, x \in \mathbb{Q}^n \mid Ax \geq Bp + c \,\} \ ,$$

with $p \in \mathbb{Z}^m$, and a (parametric) polynomial

$$f_p(x) \in (\mathbb{Q}[p])[x]$$

of degree d (in the variables x)[1], an upper bound for

$$\overline{F}_{\mathrm{c}}(p) = \max_{x \in P_p} f_p(x)$$

is computed as follows.

[1] $\mathbb{Q}[x]$ is the set of the polynomials in x over $\mathbb{Q}$. $(\mathbb{Q}[p])[x]$ thus means the set of the polynomials in x with coefficients that are polynomials in p.

Table C.1: Polyhedral chambers of the parameterized polytope defined in example 3.

Chamber	N	Parametric vertices
$p \in [0,5]$	5	$(0,0)$, $(0,6)$, $(5-p,6)$, $(5,6-p)$, $(5,0)$
$p \in [5,6]$	4	$(0,0)$, $(0,11-p)$, $(5,6-p)$, $(5,0)$
$p \in [6,11]$	3	$(0,0)$, $(0,11-p)$, $(11-p,0)$

- Recall that the parametric vertices of a parameterized polytope may only exist for a subset of the parameter values (Section 3.3.3) [119]. Therefore, the parameter domain, $\mathcal{D} = \{p \mid P_p \neq \emptyset\}$, is divided into chambers such that within each chamber C, P_p has a fixed set of N parametric vertices $v_i(p)$. Note that N may be different in different chambers.

  For the polytope defined in example 3 on page 36, there are three chambers as listed in Table C.1.

- In each chamber, the polynomial is written in terms of the barycentric coordinates of the polytope ($x = \sum_i \alpha_i v_i(p)$, $0 \leq \alpha_i \leq 1$, $\sum_i \alpha_i = 1$),

$$\tilde{f}_p(\alpha) = f_p\left(\sum_i \alpha_i v_i(p)\right) .$$

  For example, in the third chamber listed in Table C.1, i.e. $p \in [6,11]$, one can write

$$\begin{aligned}(x,y) &= \alpha_1(0,0) + \alpha_2(0,11-p) + \alpha_3(11-p,0) \\ &= (11-p)(\alpha_3,\alpha_2)\end{aligned}$$

  with $\sum_i \alpha_i = 1$ and $0 \leq \alpha_i \leq 1$. In terms of these barycentric coordinates the polynomial $f_p(x,y) = pxy$ becomes

$$\tilde{f}_p(\alpha) = p(11-p)^2\alpha_2\alpha_3 \ .$$

- This polynomial is rewritten in terms of generalized Bernstein basis polynomials

$$\tilde{f}_p(\alpha) = \sum_{\substack{k_1,k_2,\ldots,k_N \geq 0 \\ k_1+k_2+\cdots+k_N=d}} b_k^d(p) B_k^d(\alpha)$$

where the Bernstein polynomials are defined as

$$B_k^d(\alpha) = \binom{d}{k}\alpha^k = \binom{d}{k_1, k_2, \ldots, k_N}\alpha_1^{k_1}\alpha_2^{k_2}\cdots\alpha_N^{k_N} \quad ,$$

with

$$\binom{d}{k_1, k_2, \ldots, k_N} = \frac{d!}{k_1!k_2!\ldots k_N!}$$

the multinomial coefficients.

For the example above this becomes

$$\tilde{f}_p(\alpha) = p(11-p)^2\alpha_2\alpha_3 = \frac{p(11-p)^2}{2}B^2_{(0,1,1)} \quad ,$$

with

$$B^2_{(0,1,1)}(\alpha) = \binom{2}{0, 1, 1}\alpha_2\alpha_3 = 2\alpha_2\alpha_3 \quad ,$$

and thus

$$b^2_{(0,1,1)} = \frac{p(11-p)^2}{2}$$

$$b^2_{(2,0,0)} = b^2_{(0,2,0)} = b^2_{(0,0,2)} = b^2_{(1,1,0)} = b^2_{(1,0,1)} = 0 \quad .$$

- Since

$$\begin{aligned} 1 &= (\alpha_1 + \alpha_2 + \cdots + \alpha_N)^d \\ &= \sum_{\substack{k_1,k_2,\ldots,k_N \geq 0 \\ k_1+k_2+\cdots+k_N=d}} \binom{d}{k_1, k_2, \ldots, k_N}\alpha_1^{k_1}\alpha_2^{k_2}\cdots\alpha_N^{k_N} \\ &= \sum_{\substack{k_1,k_2,\ldots,k_N \geq 0 \\ k_1+k_2+\cdots+k_N=d}} B_k^d(\alpha) \quad , \end{aligned}$$

the Bernstein polynomials sum to one for all α.[2] From this and the fact that they are nonnegative for $0 \leq \alpha_i \leq 1$, the minimal

[2]This can also be derived from the fact that the Bernstein polynomials can be found in the probability mass function of the multinomial distribution. In probability theory, the multinomial distribution is a generalization of the binomial distribution. The multinomial distribution is a discrete distribution which gives the probability of choosing a given collection of n items from a set of k items with repetitions and the probabilities of each choice given by $p_1, \ldots, p_k$. These probabilities are the parameters of the multi-

and maximal resulting *Bernstein coefficients* $b_k^d(p)$ bound the polynomial, i.e.

$$\min_{\substack{k_i \geq 0 \\ \sum k_i = d}} b_k^d(p) \leq f_p(x) \leq \max_{\substack{k_i \geq 0 \\ \sum k_i = d}} b_k^d(p)$$

for all $p \in C$, $x \in P_p$.

For the working example we get

$$0 \leq f_p(x) \leq \frac{p(11-p)^2}{2}$$

for all $p \in [6, 11]$. The exact upper bound is $\frac{p(11-p)^2}{4}$. In this example the Bernstein expansion gives an overestimate of a factor 2.

- Redundant Bernstein coefficients, i.e. those that are (clearly) never larger (max) or smaller (min) than any other Bernstein coefficient, are removed.

C.2 Incremental Bernstein expansion

In an *incremental* Bernstein expansion the Bernstein coefficients are calculated in a recursive way. If part of the variables are considered as parameters the Bernstein coefficients obtained will be functions of these variables and can be expanded in their turn.

For example, we consider all but one variable to be extra parameters and compute the parametric vertices of this parametric interval with the corresponding chamber decomposition. In each of the chambers, we compute the Bernstein coefficients. Note that the single variable that was not considered to be a parameter in this computation no longer appears in these Bernstein coefficients. The above procedure is then

nomial distribution. The probability mass function of the multinomial distribution is:

$$f(x_1, \ldots, x_k; n, p_1, \ldots, p_k) = \Pr(X_1 = x_1 \wedge \cdots \wedge X_k = x_k)$$
$$= \begin{cases} \dfrac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k} = B_x^n(p) \ , & \text{when } \sum_{i=1}^k x_i = n \\ 0 & \text{otherwise} \ , \end{cases}$$

for non-negative integers $x_1, \ldots, x_k$. Since all values of a probability mass function lie in the interval $[0, 1]$ and sum to one, the Bernstein polynomials $B_x^n(p)$ for a given n have a range in $[0, 1]$ and sum to one when $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$

applied recursively on the Bernstein coefficients until all variables have been eliminated.

We will demonstrate this on the example used in the previous section with $p \in [6, 11]$.

- In the first step y is considered as a parameter, which results in a one-dimensional function over a one-dimensional parameterized polytope:

$$\begin{aligned} f_{p,y}(x) &= pyx \\ \mathcal{D} &= \{(p,y) \in \mathbb{Q}^2 \mid p \in [6,11], y \in [0, 11-p]\} \\ P_{p,y} &= \{(x) \in \mathbb{Q}^1 \mid x \in [0, 11-p-y]\} \ . \end{aligned}$$

  The parametric vertices are (0) and $(11-p-y)$ for all parameter values.[3] Now, Bernstein expansion is done:

$$\begin{aligned} (x) &= \alpha_1(0) + \alpha_2(11-p-y) \\ &= \alpha_2(11-p-y) \end{aligned}$$

$$B^1_{(1,0)}(\alpha) = \alpha_1 \qquad B^1_{(0,1)}(\alpha) = \alpha_2$$

$$pyx = py\alpha_2(11-p-y) = py(11-p-y)B^1_{(0,1)}(\alpha) \ .$$

  The Bernstein coefficients are

$$\begin{aligned} b^1_{(1,0)} &= 0 \\ b^1_{(0,1)} &= py(11-p-y) \ . \end{aligned}$$

- In a second step y is considered as a variable and Bernstein expansion is done on the Bernstein coefficients obtained in the previous step. The first coefficient is 0 and does not have to be expanded anymore. For the other coefficient we get:

$$\begin{aligned} f_p(y) &= py(11-p-y) \\ \mathcal{D} &= \{(p) \in \mathbb{Q}^1 \mid p \in [6,11]\} \\ P_p &= \{(y) \in \mathbb{Q}^1 \mid y \in [0, 11-p]\} \ . \end{aligned}$$

[3] If p was not restricted to [6,11], there would have been two chambers:

$$C_1 = \{(p,y) \in \mathbb{Q}^2 \mid 0 \le y \le 6, 0 \le p \le 6-y\}$$

with parametric vertices (0) and (5) and

$$C_2 = \{(p,y) \in \mathbb{Q}^2 \mid 0 \le y \le 6, 6-y \le p \le 11-y\}$$

with parametric vertices (0) and $(11-p-y)$.

The parametric vertices are (0) and $(11-p)$ for all parameter values. No division into chambers is needed.

$$\begin{aligned}(y) &= \alpha_1(0) + \alpha_2(11-p) \\ &= \alpha_2(11-p)\end{aligned}$$

$$B^2_{(2,0)}(\alpha) = \alpha_1^2 \quad B^2_{(0,2)}(\alpha) = \alpha_2^2 \quad B^2_{(1,1)}(\alpha) = 2\alpha_1\alpha_2$$

$$\begin{aligned}f_p(y) &= py(11-p-y) \\ &= p\alpha_2(11-p)(11-p-\alpha_2(11-p)) \\ &= p(11-p)^2\alpha_2(1-\alpha_2) \\ &= p(11-p)^2\alpha_2\alpha_1 \\ &= \frac{p(11-p)^2}{2}B^2_{(1,1)}(\alpha)\end{aligned}$$

The Bernstein coefficients are

$$\begin{aligned}b^2_{(2,0)} &= 0 \\ b^2_{(0,2)} &= 0 \\ b^2_{(1,1)} &= \frac{p(11-p)^2}{2}\ ,\end{aligned}$$

which leads to the same bounds as in the previous section.

The motivation for considering an incremental computation is that in the direct computation, the number of Bernstein coefficients is equal to the number of multinomial coefficients $\binom{d}{k_1,k_2,\ldots,k_N}$, which is

$$\binom{d+N-1}{d} > \frac{N^d}{d!}\ ,$$

where d is the degree of the polynomial and N is the number of vertices.

In the incremental computation, all parametric polytopes are intervals. They therefore have 2 vertices and require only $d+1$ Bernstein coefficients. On the other hand, the interval Bernstein expansion has to be performed in all nodes of a tree with $2n$ levels, where n is the number of variables, which branches alternately over the chambers and the Bernstein coefficients. Furthermore, the final number of chambers may be larger than the number of chambers obtained through a direct application of Bernstein expansion. Note that an advantage of this possibly

increased number of chambers is that the results may be more accurate. Preliminary experiments have shown that the incremental computation performs better than the direct computation, mainly because the removal of redundant Bernstein coefficients can be applied at each level, greatly reducing the total number of Bernstein coefficients considered.

Incremental Bernstein was thought up and implemented in the `barvinok` library by Sven Verdoolaege.

Appendix D

MRE of 3rd degree polynomials

An upper bound on the relative difference (RE) between continuous-domain and discrete-domain extrema of polynomials is found in the proof of Theorem 3 in Section 4.5.1. For second degree polynomials an exact bound is given in (4.16) or Theorem 5. This appendix determines an exact bound (MRE) for third degree polynomials (listed in Table 4.1(a)).

D.1 Problem formulation

Let the RE of a polynomial be defined as

$$RE = \begin{cases} \dfrac{(\overline{F}_{\mathrm{c}} - \underline{F}_{\mathrm{c}}) - (\overline{F}_{\mathrm{d}} - \underline{F}_{\mathrm{d}})}{\overline{F}_{\mathrm{d}} - \underline{F}_{\mathrm{d}}} & \text{when } \overline{F}_{\mathrm{d}} \neq \underline{F}_{\mathrm{d}} \ , \\ 0 & \text{when } \overline{F}_{\mathrm{d}} = \underline{F}_{\mathrm{d}} \wedge \overline{F}_{\mathrm{c}} = \underline{F}_{\mathrm{c}} \ , \\ \infty & \text{when } \overline{F}_{\mathrm{d}} = \underline{F}_{\mathrm{d}} \wedge \overline{F}_{\mathrm{c}} \neq \underline{F}_{\mathrm{c}} \ , \end{cases}$$

with $\overline{F}_{\mathrm{c}}$ and $\underline{F}_{\mathrm{c}}$ the extrema of this polynomial in the continuous interval $[0, N]$, $N \in \mathbb{N}$ ($N \geq 3$), and $\overline{F}_{\mathrm{d}}$ and $\underline{F}_{\mathrm{d}}$ the maximum and minimum in the discrete subdomain $\{0, 1, \ldots, N\} = \mathbb{Z} \cap [0, N]$. Find a third degree polynomial for which the RE becomes maximal.

D.2 Solution

Consider the general third degree polynomial

$$f(x) = ax^3 + bx^2 + cx + d \ . \tag{D.1}$$

Without loss of generality we assume $a > 0$.

The derivative $f'(x) = 3ax^2 + 2bx + c$ has discriminant $D = 4b^2 - 12ac$. If $b^2 - 3ac < 0$ the polynomial has no local extrema and $RE = 0$. When $b^2 - 3ac = 0$ there is one point of inflection with horizontal tangent and thus also $RE = 0$. For $b^2 - 3ac > 0$ there are two local extrema

$$\begin{aligned} x_l &= \frac{-b - \sqrt{b^2 - 3ac}}{3a} \\ x_r &= \frac{-b + \sqrt{b^2 - 3ac}}{3a} \ . \end{aligned}$$

Note that the point $\left(\frac{-b}{3a}, f\left(\frac{-b}{3a}\right)\right)$ is a point of symmetry of the polynomial function.

If none of the extrema lies inside the interval $[0, N]$ RE will be 0. Now, three cases are left, based on the locations of the local extrema:

Case 1: $0 \leq x_l < x_r \leq N$

The extrema can be written as:

$$\begin{aligned} \overline{F}_{\mathrm{c}} &= \max(f(x_l), f(N)) \\ \underline{F}_{\mathrm{c}} &= \min(f(0), f(x_r)) \\ \overline{F}_{\mathrm{d}} &= \max(f(\lfloor x_l \rfloor), f(\lceil x_l \rceil), f(N)) \\ \underline{F}_{\mathrm{d}} &= \min(f(0), f(\lfloor x_r \rfloor), f(\lceil x_r \rceil)) \ . \end{aligned}$$

We will prove that the maximal RE is obtained when

$$\begin{aligned} f(\lfloor x_l \rfloor) &= f(\lceil x_l \rceil) = \overline{F}_{\mathrm{d}} \\ f(\lfloor x_r \rfloor) &= f(\lceil x_r \rceil) = \underline{F}_{\mathrm{d}} \ . \end{aligned}$$

Since the RE is scale and position invariant we can assume $f(x_l) = 1$ and $f(x_r) = -1$. The polynomial is then fully determined by x_l and x_r. We first ignore the influence of $f(0)$ and $f(N)$.

Lemma 2. *Given $f(x_l) = 1$, $f(x_r) = -1$, $x_r = A + B$ and $x_l = A - B$, then*

$$RE_r \triangleq \frac{(1 - \max(f(\lfloor x_l \rfloor), f(\lceil x_l \rceil))) - (-1 - \min(f(\lfloor x_r \rfloor), f(\lceil x_r \rceil)))}{2}$$

can only reach a local maximum in the (A, B)-space when $2A \in \mathbb{Z}$.

Proof. Note that $A = \frac{-b}{3a}$, with a and b defined in (D.1), and $(A, f(A))$ is the symmetry point of $f(x)$. From $f(x_l) = -f(x_r)$ it follows that $f(A) = 0$.

Assume $2A \notin \mathbb{Z}$. Then $f(\lfloor x_l \rfloor) \neq f(\lceil x_l \rceil)$ or $f(\lfloor x_r \rfloor) \neq f(\lceil x_r \rceil)$. When $f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil)$ and $f(\lfloor x_r \rfloor) = f(\lceil x_r \rceil)$ there would exist $i, j \in \mathbb{Z}, i \neq j$ for which $f(i) = f(i+1)$ and $f(j) = f(j+1)$ and thus

$$\begin{aligned} ai^3 + bi^2 + ci + d &= a(i+1)^3 + b(i+1)^2 + c(i+1) + d \\ 0 &= a(3i^2 + 3i + 1) + 2bi + b + c \\ 0 &= a(3j^2 + 3j + 1) + 2bj + b + c \\ 0 &= 3a(i^2 - j^2 + i - j) + 2b(i - j) \ . \end{aligned}$$

Since $i, j \in \mathbb{Z}$ and $i \neq j$ this results in

$$\begin{aligned} 3a(i + j + 1) + 2b &= 0 \\ 2A = \frac{-2b}{3a} &= i + j + 1 \in \mathbb{Z} \ . \end{aligned}$$

1. $f(\lfloor x_l \rfloor) \neq f(\lceil x_l \rceil)$ and $f(\lfloor x_r \rfloor) \neq f(\lceil x_r \rceil)$.
   There are four subcases:

   (a) $f(\lfloor x_l \rfloor) > f(\lceil x_l \rceil)$, $f(\lfloor x_r \rfloor) > f(\lceil x_r \rceil)$.
   If the tangents in $\lfloor x_l \rfloor$ and $\lceil x_r \rceil$ are not parallel then a horizontal shift of the curve away from the point with the steepest tangent increases RE_r since the value of one discrete extremum then varies faster than the other (Figure D.1(a)). If the tangents are parallel, i.e. $f'(\lfloor x_l \rfloor) = f'(\lceil x_r \rceil)$ there exist $i, j \in \mathbb{Z}, i \neq j$ with

$$\begin{aligned} f'(i) &= f'(j) \\ 3ai^2 + 2bi + c &= 3aj^2 + 2bj + c \\ 3a(i + j) + 2b &= 0 \\ 2A = \frac{-2b}{3a} &= i + j \in \mathbb{Z} \ . \end{aligned}$$

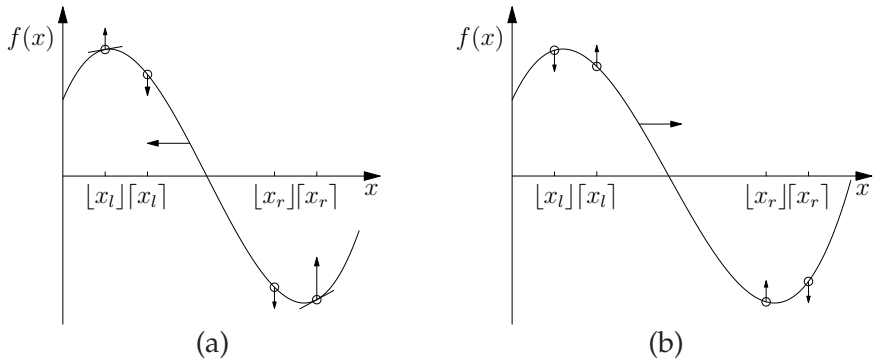


Figure D.1: Examples illustrating the cases 1a and 1b of the relative positions of $f(\lfloor x_l \rfloor)$, $f(\lceil x_l \rceil)$, $f(\lfloor x_r \rfloor)$ and $f(\lceil x_r \rceil)$. A horizontal arrow denotes a horizontal shift of the curve. The resulting vertical shifts of the values in the discrete points are indicated with vertical arrows.

(b) $f(\lfloor x_l \rfloor) > f(\lceil x_l \rceil),\ f(\lfloor x_r \rfloor) < f(\lceil x_r \rceil)$.
Shifting the curve to the right increases RE_r. (Figure D.1(b)).

(c) $f(\lfloor x_l \rfloor) < f(\lceil x_l \rceil),\ f(\lfloor x_r \rfloor) > f(\lceil x_r \rceil)$.
Analogous to 1b.

(d) $f(\lfloor x_l \rfloor) < f(\lceil x_l \rceil),\ f(\lfloor x_r \rfloor) < f(\lceil x_r \rceil)$.
Analogous to 1a.

2. $f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil)$ or $f(\lfloor x_r \rfloor) = f(\lceil x_r \rceil)$ There our four subcases similar to case 1. In some cases it may be needed that B and A vary together such that $f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil)$ or $f(\lfloor x_r \rfloor) = f(\lceil x_r \rceil)$, respectively. The detailed elaboration is omitted for brevity.

□

Lemma 3. *If $2A \in \mathbb{Z}$ then RE_r reaches a local maximum when $f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil)$ and $f(\lfloor x_r \rfloor) = f(\lceil x_r \rceil)$.*

Proof. When varying B, $\max(f(\lfloor x_l \rfloor), f(\lceil x_l \rceil))$ reaches a minimum when $f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil)$ and $\min(f(\lfloor x_r \rfloor), f(\lceil x_r \rceil))$ reaches a maximum when $f(\lfloor x_r \rfloor) = f(\lceil x_r \rceil)$. Since $2A \in \mathbb{Z}$ both equalities are always satisfied or not at the same time. □

Lemma 4. *RE can only reach a local maximum when*

$$\begin{aligned} f(0) &\geq \min(f(\lfloor x_r \rfloor), f(\lceil x_r \rceil)) \\ f(N) &\leq \max(f(\lfloor x_l \rfloor), f(\lceil x_l \rceil)) \ . \end{aligned}$$

Proof. If $f(0) < \min(f(\lfloor x_r \rfloor), f(\lceil x_r \rceil))$ or $f(N) > \max(f(\lfloor x_l \rfloor), f(\lceil x_l \rceil))$ RE can be increased by varying A and B appropriately. The details are omitted for brevity. □

Corollary 1. *If $0 \leq x_l < x_r \leq N$ the maximal RE is reached when*

$$\begin{aligned} \overline{F}_{\mathrm{d}} &= f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil) \geq f(N) \\ \underline{F}_{\mathrm{d}} &= f(\lfloor x_r \rfloor) = f(\lceil x_r \rceil) \leq f(0) \ , \end{aligned}$$

and B is minimal.

Indeed, when $f(x_l) = 1 = -f(x_r)$ and $f(\lfloor x_l \rfloor) = f(\lceil x_l \rceil)$, a smaller value of B leads to a higher value of $f(x_l) - f(\lfloor x_l \rfloor) = f(x_l) - f(\lceil x_l \rceil)$.

This leads to the solutions listed at the end of Section 4.5.1. The corresponding MREs are printed in Table 4.1(a).

Case 2: $x_l < 0 \leq x_r \leq N$

For a given polynomial it is always possible to construct a polynomial with an equal or higher RE for which $0 \leq x_l < x_r \leq N$. For brevity, this is not proved here.

Case 3: $0 \leq x_l \leq N < x_r$

Thanks to the point symmetry of the polynomial around $(-\frac{b}{3a}, f(-\frac{b}{3a}))$ this case can be converted into case 2.

Appendix E

Bounds on quasi-polynomials with the barvinok library

E.1 Command line interface

The `bernstein` library used for Bernstein expansion is included in the `barvinok` library [160]. Most of the methods described in Section 4.6 are implemented in the `barvinok` library and available through the command line application `barvinok_maximize`. This program reads a piecewise (quasi-)polynomial from standard input and converts it to a (set of) polynomial(s) on which Bernstein expansion is applied to find upper and lower bounds. Following command line options are used in the experiments:

Add Var	= Default option
Split Periods	`--split=`$Threshold$

Without the option `--minimize`, the maximum is output. *Mod Classes* and *Exact* are not yet included in the library, at the time of writing, but implemented using the library.

The input format corresponds to the output of the enumeration applications, such as `barvinok_enumerate_e`. Here, the following command line options are used to have a piecewise polynomial as output.

Drop Frac	`--polynomial-approximation=upper` `--approximation-method=drop`

Poly Approx	`--polynomial-approximation=upper`
	`--approximation-method=scale`
	`--scale-options=narrow`

A command line could look like:

```
barvinok_enumerate < polytope.in \
    | barvinok_maximize --split=10 --minimize
```

E.2 Library call

When iterating over a large number of schedule functions, using the command line interface would terribly slow down the execution. Therefore, the counting problems are generated using the internal data types of the `barvinok` library and the enumeration and maximization functions are called directly from within the C++ program that iterates over the schedules.

Appendix F

Example of code generated by CLooGVHDL

This appendix lists the code generated by CLooGVHDL corresponding to the C code listed in Figure 5.5. The different blocks of the control entity can be distinguished in the code:

for 1	line 103–147
for 2	line 149–193
ID 0	line 195–249
ID 1	line 251–314
ID 2	line 316–379

The rest of the code contains glue logic, statement argument assignments (line 391–432) and a process that generates the value of the *ready* output port (line 436–447).

```
1  -- Generated from filter_comp.cloog by (VHDL)CLooG v0.12.2 64 bits in
       0.024001s.
2  LIBRARY ieee;
3  LIBRARY work;
4  --library
5  USE work.test_constants.all;
6  USE ieee.std_logic_1164.all;
7  USE work.cloog_functions.all;
8
9  entity filter_comp is
10 port(
11   clk   : in std_logic;
12   reset : in std_logic;
13   start : in std_logic;
```

```
14      lc      : out std_logic ;
15      ready : out std_logic ;
16      S1_arg_0 : out integer range -512 to 511;
17      S1_arg_1 : out integer range -512 to 511;
18      S1_arg_2 : out integer range -512 to 511;
19      S1_arg_3 : out integer range -512 to 511;
20      S1_arg_4 : out integer range -512 to 511;
21      S1_arg_5 : out integer range -512 to 511;
22      S1_lc : in std_logic ;
23      S2_arg_0 : out integer range -512 to 511;
24      S2_arg_1 : out integer range -512 to 511;
25      S2_arg_2 : out integer range -512 to 511;
26      S2_arg_3 : out integer range -512 to 511;
27      S2_arg_4 : out integer range -512 to 511;
28      S2_arg_5 : out integer range -512 to 511;
29      S2_lc : in std_logic ;
30      start_S1 : out std_logic ;
31      start_S2 : out std_logic ;
32      cols : in integer range -512 to 511;
33      rows : in integer range -512 to 511
34      );
35    end filter_comp;
36    architecture cloog_gen of filter_comp is
37      signal S1_lc_g : std_logic ;
38      signal S2_lc_g : std_logic ;
39      signal for_1_ev : integer range -512 to 511;
40      signal for_1_lc : std_logic ;
41      signal for_1_lc_g : std_logic ;
42      signal for_1_lc_g_unbuf : std_logic ;
43      signal for_1_li : std_logic ;
44      signal for_1_running : std_logic ;
45      signal for_1_sv : integer range -512 to 511;
46      signal for_2_ev : integer range -512 to 511;
47      signal for_2_lc : std_logic ;
48      signal for_2_lc_g : std_logic ;
49      signal for_2_lc_g_unbuf : std_logic ;
50      signal for_2_li : std_logic ;
51      signal for_2_running : std_logic ;
52      signal for_2_sv : integer range -512 to 511;
53      signal guard_S1 : std_logic ;
54      signal guard_S2 : std_logic ;
55      signal guard_for_1 : std_logic ;
56      signal guard_for_2 : std_logic ;
57      signal i_1_ext , i_1_int , inc_1 : integer range -512 to 511;
58      signal i_2_ext , i_2_int , inc_2 : integer range -512 to 511;
```

```
59    signal p3 : integer range −512 to 511;
60    signal p5 : integer range −512 to 511;
61    signal seq_0_lc : std_logic;
62    signal seq_0_running : std_logic;
63    signal seq_1_lc : std_logic;
64    signal seq_1_running : std_logic;
65    signal seq_2_lc : std_logic;
66    signal seq_2_running : std_logic;
67    signal start_S1_g : std_logic;
68    signal start_S1_s : std_logic;
69    signal start_S1_t2 : std_logic;
70    signal start_S1_t2_int : std_logic;
71    signal start_S2_g : std_logic;
72    signal start_S2_s : std_logic;
73    signal start_S2_t2 : std_logic;
74    signal start_S2_t2_int : std_logic;
75    signal start_for_1 : std_logic;
76    signal start_for_1_g : std_logic;
77    signal start_for_1_int : std_logic;
78    signal start_for_2 : std_logic;
79    signal start_for_2_g : std_logic;
80    signal start_for_2_int : std_logic;
81    signal start_inner_for_1 , start_inner_for_2 : std_logic;
82    signal start_seq_0 , start_seq_1 , start_seq_2 : std_logic;
83    signal t_0 , t_0_l0 , t_0_l1 , t_0_l2 : integer range 0 to 1;
84    signal t_1 , t_1_l1 , t_1_l2 : integer range 0 to 1;
85    signal t_2 , t_2_l2 : integer range 0 to 1;
86  begin
87    start_seq_0 <= start;
88    lc <= seq_0_lc;
89
90    t_0_l0 <= t_0;
91    t_0_l1 <= t_0_l0;
92    t_0_l2 <= t_0_l1;
93    t_1_l1 <= t_1;
94    t_1_l2 <= t_1_l1;
95    t_2_l2 <= t_2;
96    start_S1_g <= start_S1_t2;
97    start_S1_s <= start_S1_g when guard_S1 ='1' else '0';
98    start_S1 <= start_S1_s;
99    start_S2_g <= start_S2_t2;
100   start_S2_s <= start_S2_g when guard_S2 ='1' else '0';
101   start_S2 <= start_S2_s;
102
103   start_seq_1 <= start_for_1 or start_inner_for_1 ;
```

```
104     for_1_li  <= '1' when ((i_1_ext+inc_1)> for_1_ev ) else  '0';
105     for_1_lc  <= for_1_li  and seq_1_lc;
106     for_1_lc_g_unbuf  <= for_1_lc  when guard_for_1='1' else
            start_for_1_g ;
107     for_1_lc_g  <= for_1_lc_g_unbuf;
108     i_1_ext  <= for_1_sv when ((start_for_1 ='1')  and (for_1_running='0')
            ) else  i_1_int  ;
109
110     for_1 :  process(clk)
111     begin
112       if  (clk='1'  and clk'event) then
113         if  (reset  =  '1')  then
114           start_inner_for_1  <= '0';
115           for_1_running <= '0';
116         else  --reset
117           if  for_1_running='1' then
118             if  seq_1_lc='1'  then
119               i_1_int  <= i_1_int+inc_1  ;
120               if  for_1_li ='1'  then
121                 for_1_running <= '0';
122                 start_inner_for_1  <= '0';
123               else  --not last  iter
124                 start_inner_for_1  <= '1';
125               end if;
126             else  --seq_lc
127               start_inner_for_1  <= '0';
128             end if;--seq_lc
129           else  --for_running
130             if  start_for_1 ='1'  then
131               if  seq_1_lc  =  '0'  then
132                 i_1_int  <= for_1_sv;
133               else
134                 i_1_int  <= for_1_sv + inc_1;
135               end if;
136               if  for_1_lc_g_unbuf  =  '0'  then
137                 for_1_running <= '1';
138                 start_inner_for_1  <= seq_1_lc;
139               else
140                 for_1_running <= '0';
141                 start_inner_for_1  <= '0';
142               end if;
143             end if;
144           end if;  --for_running
145         end if;  --else  reset
146       end if;  --clk
```

```
147   end process;
148
149   start_seq_2 <= start_for_2 or start_inner_for_2 ;
150   for_2_li <= '1' when ((i_2_ext+inc_2)> for_2_ev ) else '0';
151   for_2_lc <= for_2_li and seq_2_lc;
152   for_2_lc_g_unbuf <= for_2_lc when guard_for_2='1' else
          start_for_2_g ;
153   for_2_lc_g <= for_2_lc_g_unbuf;
154   i_2_ext <= for_2_sv when ((start_for_2 ='1') and (for_2_running='0')
          ) else i_2_int ;
155
156   for_2: process(clk)
157   begin
158     if (clk='1' and clk'event) then
159       if (reset = '1') then
160         start_inner_for_2 <= '0';
161         for_2_running <= '0';
162       else --reset
163         if for_2_running='1' then
164           if seq_2_lc='1' then
165             i_2_int <= i_2_int+inc_2 ;
166             if for_2_li ='1' then
167               for_2_running <= '0';
168               start_inner_for_2 <= '0';
169             else --not last iter
170               start_inner_for_2 <= '1';
171             end if;
172           else --seq_lc
173             start_inner_for_2 <= '0';
174           end if;--seq_lc
175         else --for_running
176           if start_for_2 ='1' then
177             if seq_2_lc = '0' then
178               i_2_int <= for_2_sv;
179             else
180               i_2_int <= for_2_sv + inc_2;
181             end if;
182             if for_2_lc_g_unbuf = '0' then
183               for_2_running <= '1';
184               start_inner_for_2 <= seq_2_lc;
185             else
186               for_2_running <= '0';
187               start_inner_for_2 <= '0';
188             end if;
189           end if;
```

```
190          end if; --for_running
191        end if; --else reset
192      end if; --clk
193    end process;
194
195  -- sequence_0
196    seq_0_lc <= '1' when ((t_0=1)and (for_1_lc_g = '1') ) else '0';
197
198    seq_0_running_p: process(clk)
199    begin
200      if (clk='1' and clk'event) then
201        if (reset = '1') then
202          seq_0_running <= '0';
203        elsif seq_0_lc ='1' then
204          seq_0_running <= '0';
205        elsif start_seq_0 ='1' then
206          seq_0_running <= '1';
207        end if; --else reset
208      end if; --clk
209    end process;
210
211    seq_0: process(clk)
212    begin
213      if (clk='1' and clk'event) then
214         start_for_1_int <= '0';
215        if (reset = '1') then
216          t_0 <= 0;
217        else --reset
218          if (seq_0_running ='1') or (start_seq_0 ='1') then
219            case t_0_l0 is
220              when 0 =>
221                if for_1_lc_g = '1' then
222                  t_0 <= t_0 + 1;
223                   start_for_1_int <= '1';
224                end if;
225              when 1 =>
226                if for_1_lc_g = '1' then
227                  t_0 <= 0;
228                end if;
229              when others => null;
230              end case; -- t_0
231          end if; --running or start
232        end if; --else reset
233      end if; --clk
234    end process;
```

```
235     for_1_sv <=
236       0 when (t_0_l0=0) else
237       2 when (t_0_l0=1) else
238       0;
239     for_1_ev <=
240       1 when (t_0_l0=0) else
241       rows−1 when (t_0_l0=1) else
242       1;
243     inc_1 <=
244       1 when (t_0_l0=0) else
245       1 when (t_0_l0=1) else
246       1;
247     guard_for_1 <= '0' when (for_1_sv > for_1_ev) else '1';
248     start_for_1_g <= '1' when ((start_for_1_int ='1') or (start_seq_0
          ='1' and (((t_0=0)) ) ) ) else '0';
249     start_for_1 <= start_for_1_g and guard_for_1;
250
251   −− sequence_1
252     seq_1_lc <= '1' when
253       (( t_0_l1 =0) and (((t_1=0)and (for_2_lc_g = '1') )))or(( t_0_l1 =1)
            and (((t_1=0)and (for_2_lc_g = '1') )))
254       else '0';
255
256     seq_1_running_p: process(clk)
257     begin
258       if (clk='1' and clk'event) then
259         if (reset = '1') then
260           seq_1_running <= '0';
261         elsif seq_1_lc ='1' then
262           seq_1_running <= '0';
263         elsif start_seq_1 ='1' then
264           seq_1_running <= '1';
265         end if; −−else reset
266       end if; −−clk
267     end process;
268
269     seq_1: process(clk)
270     begin
271       if (clk='1' and clk'event) then
272         start_for_2_int <= '0';
273         if (reset = '1') then
274           t_1 <= 0;
275         else −−reset
276           if (seq_1_running ='1') or (start_seq_1 ='1') then
277             case t_0_l1 is
```

```
278                 when 0 =>
279                   case t_1_l1 is
280                     when 0 =>
281                       if for_2_lc_g = '1' then
282                         t_1 <= 0;
283                       end if;
284                     when others => null;
285                     end case; -- t_1
286                 when 1 =>
287                   case t_1_l1 is
288                     when 0 =>
289                       if for_2_lc_g = '1' then
290                         t_1 <= 0;
291                       end if;
292                     when others => null;
293                     end case; -- t_1
294                 when others => null;
295                 end case; -- t_0
296             end if; --running or start
297           end if; --else reset
298         end if; --clk
299       end process;
300       for_2_sv <=
301         1 when (t_0_l1=0) and (t_1_l1=0) else
302         1 when (t_0_l1=1) and (t_1_l1=0) else
303         1;
304       for_2_ev <=
305         cols-2 when (t_0_l1=0) and (t_1_l1=0) else
306         cols-2 when (t_0_l1=1) and (t_1_l1=0) else
307         cols-2;
308       inc_2 <=
309         1 when (t_0_l1=0) and (t_1_l1=0) else
310         1 when (t_0_l1=1) and (t_1_l1=0) else
311         1;
312       guard_for_2 <= '0' when (for_2_sv > for_2_ev) else '1';
313       start_for_2_g <= '1' when ((start_for_2_int ='1') or (start_seq_1
              ='1' and (((t_0_l1=0) and (t_1=0)) or ((t_0_l1=1) and (t_1=0))
              ))))) else '0';
314       start_for_2 <= start_for_2_g and guard_for_2;
315
316     -- sequence_2
317       seq_2_lc <= '1' when ((t_0_l2=0) and (((t_1_l2=0) and (((t_2=0)and
              (S1_lc_g = '1')))))or((t_0_l2=1) and (((t_1_l2=0) and (((t_2
              =1)and (S2_lc_g = '1')))))
318         else '0';
```

```
319
320     seq_2_running_p: process(clk)
321     begin
322       if (clk='1' and clk'event) then
323         if (reset = '1') then
324           seq_2_running <= '0';
325         elsif seq_2_lc ='1' then
326           seq_2_running <= '0';
327         elsif start_seq_2 ='1' then
328           seq_2_running <= '1';
329         end if; --else reset
330       end if; --clk
331     end process;
332
333     seq_2: process(clk)
334     begin
335       if (clk='1' and clk'event) then
336         start_S1_t2_int <= '0';
337         start_S2_t2_int <= '0';
338         if (reset = '1') then
339           t_2 <= 0;
340         else --reset
341           if (seq_2_running ='1') or (start_seq_2 ='1') then
342             case t_0_l2 is
343               when 0 =>
344                 case t_1_l2 is
345                   when 0 =>
346                     case t_2_l2 is
347                       when 0 =>
348                         if S1_lc_g = '1' then
349                           t_2 <= 0;
350                         end if;
351                       when others => null;
352                       end case; -- t_2
353                   when others => null;
354                   end case; -- t_1
355               when 1 =>
356                 case t_1_l2 is
357                   when 0 =>
358                     case t_2_l2 is
359                       when 0 =>
360                         if S1_lc_g = '1' then
361                           t_2 <= t_2 + 1;
362                           start_S2_t2_int <= '1';
363                         end if;
```

```
364                         when 1 =>
365                           if S2_lc_g = '1' then
366                             t_2 <= 0;
367                           end if;
368                         when others => null;
369                         end case; -- t_2
370                     when others => null;
371                     end case; -- t_1
372                 when others => null;
373                 end case; -- t_0
374             end if; --running or start
375         end if; --else reset
376     end if; --clk
377 end process;
378 start_S1_t2 <= '1' when ((start_S1_t2_int ='1') or (start_seq_2 ='1'
        and (((t_0_l2 =0) and (t_1_l2 =0) and (t_2=0)) or ((t_0_l2 =1)
        and (t_1_l2 =0) and (t_2=0)) ) ) ) else '0';
379 start_S2_t2 <= start_S2_t2_int ;
380
381 p3 <=
382     i_1_ext when (t_0_l0=0) else
383     i_1_ext when (t_0_l0=1) else
384     i_1_ext ;
385 p5 <=
386     i_2_ext when (t_0_l1=0) and (t_1_l1 =0) else
387     i_2_ext when (t_0_l1=1) and (t_1_l1 =0) else
388     i_2_ext ;
389 guard_S1 <= '1';
390 guard_S2 <= '1';
391 S1_arg_0 <=
392     0 when ((t_0_l2=0) and (t_1_l2 =0) and (t_2_l2 =0)) else
393     0 when ((t_0_l2=1) and (t_1_l2 =0) and (t_2_l2 =0)) else
394     0;
395 S1_arg_1 <=
396     0 when ((t_0_l2=0) and (t_1_l2 =0) and (t_2_l2 =0)) else
397     0 when ((t_0_l2=1) and (t_1_l2 =0) and (t_2_l2 =0)) else
398     0;
399 S1_arg_2 <=
400     p3 when ((t_0_l2=0) and (t_1_l2 =0) and (t_2_l2 =0)) else
401     p3 when ((t_0_l2=1) and (t_1_l2 =0) and (t_2_l2 =0)) else
402     p3;
403 S1_arg_3 <=
404     0 when ((t_0_l2=0) and (t_1_l2 =0) and (t_2_l2 =0)) else
405     0 when ((t_0_l2=1) and (t_1_l2 =0) and (t_2_l2 =0)) else
406     0;
```

```
407     S1_arg_4 <=
408       p5 when ((t_0_l2=0) and (t_1_l2=0) and (t_2_l2=0)) else
409       p5 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=0)) else
410       p5;
411     S1_arg_5 <=
412       0 when ((t_0_l2=0) and (t_1_l2=0) and (t_2_l2=0)) else
413       0 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=0)) else
414       0;
415     S2_arg_0 <=
416       0 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=1)) else
417       0;
418     S2_arg_1 <=
419       1 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=1)) else
420       1;
421     S2_arg_2 <=
422       p5 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=1)) else
423       p5;
424     S2_arg_3 <=
425       0 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=1)) else
426       0;
427     S2_arg_4 <=
428       p3-1 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=1)) else
429       p3-1;
430     S2_arg_5 <=
431       0 when ((t_0_l2=1) and (t_1_l2=0) and (t_2_l2=1)) else
432       0;
433     S1_lc_g <= S1_lc when guard_S1='1' else start_S1_g;
434     S2_lc_g <= S2_lc when guard_S2='1' else start_S2_g;
435
436     ready_p: process(clk)
437     begin
438       if (clk='1' and clk'event) then
439         if (reset = '1') then
440           ready <= '1';
441         elsif start = '1' and seq_0_lc = '0' then
442           ready <= '0';
443         elsif seq_0_lc = '1' then
444           ready <= '1';
445         end if; --else reset
446       end if; --clk
447     end process;
448   end architecture cloog_gen;
```

Index

List of publications

Journal papers

[66] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, Erik H. D'Hollander and Dirk Stroobandt. Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations. *Transactions on High Performance Embedded Architectures and Compilers I*, *Lecture Notes in Computer Science*, Vol. 4050, 2007, pp. 159–178.

[80] Hendrik Eeckhaut, Harald Devos, Peter Lambert, Davy De Schrijver, Wim Van Lancker, Vincent Nollet, Prabhat Avasare, Tom Clerckx, Fabio Verdicchio, Mark Christiaens, Peter Schelkens, Rik Van de Walle, Dirk Stroobandt. Scalable, Wavelet-Based Video: from Server to Hardware-Accelerated Client. *IEEE Transactions on Multimedia*, Vol. 9 (7), 2007, pp. 1508-1519

[82] Hendrik Eeckhaut, Dirk Stroobandt, Harald Devos and Mark Christiaens. Improving the Hardware Friendliness of a Wavelet Based Scalable Video Codec. *WSEAS Transactions on Systems*, Vol. 4 (5), 2005, pp. 625–634. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P105_073.pdf`.

[152] Dirk Stroobandt, Hendrik Eeckhaut, Harald Devos, Mark Christiaens, Fabio Verdicchio and Peter Schelkens. Reconfigurable Hardware for a Scalable Wavelet Video Decoder and Its Performance Requirements. *Computer Systems: Architectures, Modeling, and Simulation*, *Lecture Notes in Computer Science*, Vol. 3133, 2004, pp. 203–212.

[68] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout and Dirk Stroobandt. Systematic Design Space Exploration of the Discrete Wavelet Transform. Submitted to *Elsevier Digital Signal Processing*. In review since December 2006.

[72] Harald Devos, Sven Verdoolaege, Jan Van Campenhout, and Dirk Stroobandt. Bounds on polynomials and quasi-polynomials over discrete domains. Submitted to *Computing*. Preparing revised version.

Conference papers with international reviewers

[71] Harald Devos, Jan Van Campenhout, and Dirk Stroobandt. Building an Application-specific Memory Hierarchy on FPGA. In *2nd HiPEAC Workshop on Reconfigurable Computing*, January 2008, Göteborg, Sweden, pp. 53–62. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P108_009.pdf`.

- Fabian Diet, Erik H. D'Hollander, Kristof Beyls and Harald Devos. Embedding Smart Buffers for Window Operations in a Stream-Oriented C-to-VHDL Compiler. In *4th IEEE International Symposium on Electronic Design, Test & Applications (DELTA'08)*, January 2008, Hong Kong.

[81] Hendrik Eeckhaut, Harald Devos and Dirk Stroobandt. The Energy Scalability of Wavelet-based, Scalable Video Decoding. In *17th International Workshop on Power And Timing Modeling, Optimization and Simulation*, September 2007, Göteborg, Sweden. *Lecture Notes in Computer Science*, Vol. 4644, pp. 363–372.

[79] Hendrik Eeckhaut, Harald Devos, Philippe Faes, Mark Christiaens, Dirk Stroobandt. FPGA design Methodology for a Wavelet-Based Scalable Video Decoder. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 7th International Workshop, SAMOS 2007, Greece*, *Lecture Notes in Computer Science*, Vol. 4599, pp. 169–178.

[70] Harald Devos, Hendrik Eeckhaut, Mark Christiaens, Dirk Stroobandt. Energy Scalability and the RESUME Scalable Video Codec. In *Power-aware Computing Systems*, *Dagstuhl Seminar Proceedings* 07041, January 2007, Schloss Dagstuhl, Germany. Published on-line: `http://drops.dagstuhl.de/opus/volltexte/2007/1112/`.

[67] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout and Dirk Stroobandt. From Loop Transformation to Hardware Generation. In *Proceedings of the 17th ProRISC Workshop*, November 2006, Veldhoven, The Netherlands, pp. 249–255. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P106_224.pdf`.

[77] Hendrik Eeckhaut, Mark Christiaens, Harald Devos, Dirk Stroobandt. Implementing a Hardware-Friendly Wavelet Entropy Codec for Scalable Video. In *Proceedings of SPIE: Wavelet Applications in Industrial Processing III*, Vol. 6001, October 2005, pp. 169–179.

- Hendrik Eeckhaut, Dirk Stroobandt, Harald Devos and Mark Christiaens. Modeling Subbands of a Wavelet Based Scalable Video Codec. In *Proceedings of the 9th WSEAS International CSCC Multiconference: Circuits '05, Systems '05, Computers '05, Communications '05*, July 2005. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P105_074.pdf`.

- Hendrik Eeckhaut, Mark Christiaens, Benjamin Schrauwen, Harald Devos, Dirk Stroobandt. Improving Parallelism of a Hardware Friendly, Scalable Wavelet Entropy codec. In *Proceedings of SPS-DARTS 2005 (CD-ROM)*, April 2005, pp. 159–162. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P105_024.pdf`.

- Hendrik Eeckhaut, Harald Devos, Benjamin Schrauwen, Mark Christiaens, Dirk Stroobandt. A Hardware-Friendly Wavelet Entropy Codec for Scalable Video. In *Proceedings of Design, Automation and Test in Europe (DATE05)*, March 2005, Munich, Germany, pp. 14–19.

- Tom Clerckx, Fabio Verdicchio, Adrian Munteanu, Yiannis Andreopoulos, Harald Devos, Hendrik Eeckhaut, Mark Christiaens, Dirk Stroobandt, Diederik Verkest and Peter Schelkens. Complexity Scalable Motion Compensated Temporal Filtering. In *Proceedings of SPIE*, Vol. 5558, November 2004, pp. 116–127.

[69] Harald Devos, Hendrik Eeckhaut, Benjamin Schrauwen, Mark Christiaens, Dirk Stroobandt. Ever considered SystemC? In *Proceedings of the 15th ProRISC Workshop*, November 2004, Veldhoven, The Netherlands, pp. 358–363. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P104_120.pdf`.

- Hendrik Eeckhaut, Mark Christiaens, Harald Devos, Benjamin Schrauwen, and Dirk Stroobandt, Scalable and Hardware-friendly Wavelet Entropy Coding. In *Proceedings of the 15th ProRISC Workshop*, November 2004, Veldhoven, The Netherlands, pp. 369–376. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P104_117.pdf`.

- Harald Devos, Hendrik Eeckhaut, Mark Christiaens, Fabio Verdicchio, Dirk Stroobandt and Peter Schelkens. Performance Requirements for Reconfigurable Hardware for a Scalable Wavelet Video Decoder. In *Proceedings of ProRISC*, November 2003, pp. 56–63. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P103_123.pdf`.

- Jeroen De Maeyer, Harald Devos, Wim Meeus, Peter Verplaetse and Dirk Stroobandt. Hardware Implementation of an EAN-13 Bar Code Decoder. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*. January 2003, Kitakyushu, Japan, pp. 583–584. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P103_007.pdf`.

- Harald Devos, Joni Dambre, Wim Meeus, Dirk Stroobandt and Jan Van Campenhout. An exploration of synchronization solutions for parallel short-range optical interconnect in mesochronous systems. In *Proceedings of SPIE: VCSELs and Optical Interconnects*, Vol. 4942, October 2002, Brugge, pp. 258–268. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P102_164.pdf`.

Other publications

- Tom Degryse, Harald Devos and Dirk Stroobandt. Controller Area Estimation for Automatic Design Space Exploration. *Proceedings of the seventh ACES Symposium*, Edegem, September 2007, pp. 31–34. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P107_170.pdf`.
- Harald Devos, Jan Van Campenhout and Dirk Stroobandt. Finding Bounds on Ehrhart Quasi-Polynomials. *Proceedings of the seventh ACES Symposium*, Edegem, September 2007, pp. 35–38. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P107_168.pdf`.
- Hendrik Eeckhaut, Mark Christiaens, Harald Devos, Philippe Faes, Dirk Stroobandt. RESUME's wavelet-based scalable video decoder. *DATE 2007 University Booth*, April 2007, Nice, France, pp. S42. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P107_043.pdf`.
- Hendrik Eeckhaut, Mark Christiaens, Harald Devos, Philippe Faes, Dirk Stroobandt. UGent delivers first real-time wavelet-based scalable video decoder on FPGA. *DSP valley newsletter*, February 2007, pp. 5–6. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P107_018.pdf`.
- Chris Bleakley, Tom Clerckx, Harald Devos, Matthias Grumer, Alex Janek, Ulrich Kremer, Christian W. Probst, Phillip Stanley-Marbell, Christian Steger, Vasanth Venkatachalam and Manuel Wendt. 07041 Working Group – Towards Interfaces for Integrated Performance and Power Analysis and Simulation. In *Power-aware Computing Systems, Dagstuhl Seminar Proceedings* 07041, January 2007, Schloss Dagstuhl, Germany. Published on-line: `http://drops.dagstuhl.de/opus/volltexte/2007/1107/`.
- Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout and Dirk Stroobandt. Hardware Generation from the Polyhedral Model. *Proceedings of the sixth ACES Symposium*, Edegem, October 2006, pp. 23–26. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P106_165.pdf`.
- Harald Devos, Dirk Stroobandt and Jan Van Campenhout. Loop transformations for generating scalable hardware. In *Design, Automation and Test in Europe; Fourth EDAA Ph.D. Forum*, München, March 2006, on CD. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P106_081.pdf`.
- Harald Devos. Loop transformations for generating scalable hardware. In *6th FirW PhD Symposium Gent*, November 2005, on CD. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P105_184.pdf`.

- Harald Devos and Hendrik Eeckhaut. Performance Requirements for Reconfigurable Hardware for a Scalable Video Decoder. In *Fourth FTW PhD Symposium Gent*, December 2003, on CD. Available from `http://escher.elis.ugent.be/publ/Edocs/DOC/P103_139.pdf`.

Bibliography

[1] Altera website. http://www.altera.com/.

[2] Bluespec. http://www.bluespec.com/.

[3] Celoxica. http://www.celoxica.com/.

[4] Impulse C. http://www.impulsec.com/.

[5] Open Research Compiler, Open64. http://ipf-orc.sourceforge.net/, http://www.open64.net/.

[6] SUIF Compiler System. http://suif.stanford.edu/.

[7] Vim, Vi IMproved. http://www.vim.org/.

[8] Xilinx website. http://www.xilinx.com/.

[9] The RESUME project: Reconfigurable Embedded Systems for Use in Scalable Multimedia Environments. http://www.elis.UGent.be/resume/, 2003–2006.

[10] *Nios II C2H Compiler User Guide (ver 1.1)*, March 2007.

[11] Marleen Adé. *Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets.* PhD thesis, Katholieke Universiteit Leuven, October 1996.

[12] Altera. Application note 420: Optimizing Nios II C2H compiler results (ver 1.0). http://www.altera.com/literature/an/AN420.pdf, July 2006.

[13] Altera. *Stratix Device Handbook*, January 2006.

[14] Altera. White paper: Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions (ver 1.0). `http://www.altera.com/literature/wp/wp-aghrdwr.pdf`, May 2006.

[15] Altera. *System interconnect fabric for memory-mapped interfaces, Quartus II Handbook, Volume 4, Chapter 3*, March 2007.

[16] Yiannis Andreopoulos, Peter Schelkens, Gauthier Lafruit, Konstantinos Masselos, and Jan Cornelis. High-level cache modeling for 2-D discrete wavelet transform implementations. *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology*, 34(3):209–226, 2003.

[17] ARM. AMBA specification (Advanced Microcontroller Bus Architecture). `http://www.arm.com/`.

[18] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level synthesis: an essential ingredient for designing complex ASICs. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD-2004*, pages 775–782, November 2004.

[19] Pranav Ashar, Srinivas Devadas, and Arthur Richard Newton. *Sequential logic synthesis*. Kluwer Academic Publishers, 1992.

[20] Christos A. Athanasiadis. Ehrhart polynomials, simplicial polytopes, magic squares and a conjecture of Stanley. *Journal für die reine und angewandte Mathematik (Crelles Journal)*, 2005(583):163–174, June 2005.

[21] Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-based design from parallel C specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1811–1826, December 2005.

[22] Florin Balasa, Hongwei Zhu, and Ilie I. Luican. Computation of storage requirements for multi-dimensional signal processing applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(4):447–460, April 2007.

[23] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for

cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[24] Alexander Barvinok and James E. Pommersheim. *New Perspectives in Algebraic Combinatorics*, volume 38 of *MSRI Publications*, chapter An Algorithmic Theory of Lattice Points in Polyhedra, pages 91–147. Cambridge University Press, 1999.

[25] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, November 1994.

[26] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, October 2003.

[27] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, September 2004.

[28] Cédric Bastoul. *CLooG: A Loop Generator for Scanning Polyhedra, User's Guide, Edition 2.0 for CLooG 0.14.0*, November 2005.

[29] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computing, LNCS 2958*, pages 209–225, College Station, October 2003.

[30] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 320–335, 2003.

[31] Cédric. Bastoul and Paul Feautrier. More legal transformations for locality. In *EURO-PAR Parallel Processing, Lecture Notes in Computer Science*, volume 3149, pages 272–283. Springer-Verlag Berlin, 2004.

[32] Marcus Bednara, Frank Hannig, and Jürgen Teich. Generation of Distributed Loop Control. In Ed F. Deprettere, Jürgen Teich,

and Stamatis Vassiliadis, editors, *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation – SAMOS*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, pages 154–170, Heidelberg, Germany, 2002. Springer.

[33] Marcus Bednara and Jürgen Teich. Automatic synthesis of FPGA processor arrays from loop algorithms. *Journal of Supercomputing*, 26(2):149–165, September 2003.

[34] L. A. Belady. A study of replacement algorithms for a virtualstorage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[35] Luca Benini, Polly Siegel, and Giovanni De Micheli. Saving power by synthesizing gated clocks for sequential circuits. *IEEE Design & Test of Computers*, 11(4):32–41, 1994.

[36] Domingo Benitez. Performance of reconfigurable architectures for image-processing applications. *Journal of Systems Architecture*, 49(4-6):193–210, September 2003.

[37] K. Benkrid, D. Crookes, and A. Benkrid. Towards a general framework for FPGA based image processing using hardware skeletons. *Parallel Computing*, 28(7-8):1141–1154, August 2002.

[38] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *SIGMETRICS*, pages 169–180, 2005.

[39] Kristof Beyls. SLO – Suggestions for Locality Optimizations. `http://slo.sourceforge.net/`.

[40] Kristof Beyls. *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis, Ghent University, June 2004.

[41] Kristof Beyls and Erik D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. In *HPCC*, volume 4208 of *Lecture Notes in Computer Science*, pages 220–229, 2006.

[42] Kristof Beyls and Erik H. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[43] Kristof Beyls and Erik H. D'Hollander. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *CF '06: Proceedings of the 3rd conference on Computing*

Frontiers, pages 373–382, New York, NY, USA, May 2006. ACM Press.

[44] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[45] Inc. Bluespec. Groundbreaking technology from MIT fundamentally alters approach to ASIC/FPGA creation by attacking root of design issues. Press Release, March 2004. – Term Rewriting Systems Technology Licensed to EDA Tools Developer Bluespec as Core of Flagship Product –.

[46] W. Bohm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *Journal of Supercomputing*, 21(2):117–130, February 2002.

[47] J. Bormans, K. Denolf, S. Wuytack, L. Nachtergaele, and I. Bolsens. Integrating system-level low power methodologies into a real-life design flow. In *Ninth International Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 19–28, Kos, Greece, October 1999.

[48] Andrew Bruce, David Donoho, and Hong-Ye Gao. Wavelet analysis [for signal processing]. *IEEE Spectrum*, 33(10):26–35, October 1996.

[49] Betul Buyukkurt, Zhi Guo, and Walid Najjar. Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGA. In Koen Bertels, Joao M.P. Cardoso, and Stamatis Vassiliadis, editors, *Reconfigurable computing: architectures and applications: second international workshop, ARC 2006, Delft, The Netherlands: revised selected papers*, volume 3985 of *LNCS*, pages 401–412, March 2006.

[50] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, April 2000.

[51] Patrick Carribault and Albert Cohen. Applications of storage mapping optimization to register promotion. In *ICS '04: Proceedings of the 18th annual International Conference on Supercomputing*, pages 247–256. ACM Press, 2004.

[52] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Boston, USA, 1998.

[53] Chaitali Chakrabarti and Clint Mumford. Efficient realizations of encoders and decoders based on the 2-D discrete wavelet transform. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(3):289–298, September 1999.

[54] Christos Chrysafis and Antonio Ortega. Line-based, reduced memory, wavelet image compression. *IEEE Transactions on Image Processing*, 9(3):378–389, March 2000.

[55] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal processing systems for signal image and video technology*, 19(2):179–194, July 1998.

[56] Philippe Clauss, Javier Federico Fernandez, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. Technical Report 06-04, Université Louis Pasteur, October 2006. Available from `http://icps.u-strasbg.fr/upload/icps-2006-173.pdf`.

[57] Albert Cohen, Sylvain Girbal, David Parello, Marc Sigler, Olivier Temam, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05), Boston, Massachusetts*, June 2005.

[58] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002.

[59] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol, and Scott Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):209–220, June 2002.

[60] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, 4(3):245–267, 1998.

[61] Philip J. Davis. *Interpolation and Approximation*. Dover Publications, Inc., New York, 1975.

[62] Jesus A. De Loera, Raymond Hemmecke, Matthias Koppe, and Robert Weismantel. Integer polynomial optimization in fixed dimension. *Mathematics of operations research*, 31(1):147–153, February 2006.

[63] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, and M. Wrighton. Design patterns for reconfigurable computing. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, pages 13–23, April 2004.

[64] André DeHon. The density advantage of configurable computing. *IEEE Computer*, 33(4):41–49, April 2000.

[65] Steven Derrien and Tanguy Risset. Interfacing compiled FPGA programs: the MMAlpha approach. In *Int. conf. on Parallel and Distributed Processing Techniques and Applications*, Rennes Cedex, June 2000.

[66] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, Erik H. D'Hollander, and Dirk Stroobandt. Finding and applying loop transformations for generating optimized FPGA implementations. *Transactions on High Performance Embedded Architectures and Compilers I, LNCS*, 4050:159–178, 2007.

[67] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, and Dirk Stroobandt. From loop transformation to hardware generation. In *Proceedings of the 17th ProRISC Workshop*, pages 249–255, Veldhoven, November 2006.

[68] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, and Dirk Stroobandt. Systematic design space exploration of the discrete wavelet transform. *Sumbitted to Elsevier Digital Signal Processing*, 2006.

[69] Harald Devos, Hendrik Eeckhaut, Benjamin Schrauwen, Mark Christiaens, and Dirk Stroobandt. Ever considered SystemC? In *Proceedings of the 15th ProRISC Workshop*, pages 358–363, Veldhoven, November 2004.

[70] Harald Devos, Eeckhaut Hendrik, Mark Christiaens, and Dirk Stroobandt. Energy scalability and the RESUME scalable video codec. In Luca Benini, Naehyuck Chang, Ulrich Kremer, and Christian W. Probst, editors, *Power-aware Computing Systems*, number 07041 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, January 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. `http://drops.dagstuhl.de/opus/volltexte/2007/1112`.

[71] Harald Devos, Dirk Stroobandt, and Jan Van Campenhout. Building an application-specific memory hierarchy on FPGA. In *2nd HiPEAC Workshop on Reconfigurable Computing*, pages 53–62, Göteborg, Sweden, January 2008.

[72] Harald Devos, Sven Verdoolaege, Jan Van Campenhout, and Dirk Stroobandt. Bounds on polynomials and quasi-polynomials over discrete domains. *Computing*. Preparing revised version.

[73] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. Automatic mapping of C to FPGAs with the DEFACTO compilation and synthesis system. *Microprocessors and Microsystems*, 29(2-3):51–62, April 2005.

[74] Florian Dittmann, Achim Rettberg, and Fabian Schulte. A Y-chart based tool for reconfigurable system design. In *Workshop on Dynamically Reconfigurable Systems (DRS), Innsbruck (Austria)*, pages 67–73. VDE Verlag, March 2005.

[75] Yazhuo Dong, Yong Dou, and Jie Zhou. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In *International Workshop on Applied Reconfigurable Computing (ARC)*, volume 4419 of *Lecture Notes in Computer Science*, pages 110–121, 2007.

[76] François Donnet. *Synthèse de Haut Niveau Contrôlée par l'Utilisateur*. PhD thesis, Université Pierre et Marie Curie, Paris, France, January 2004.

[77] Hendrik Eeckhaut, Mark Christiaens, Harald Devos, and Dirk Stroobandt. Implementing a hardware-friendly wavelet entropy codec for scalable video. In F. Truchetet and O. Laligant, editors, *Proceedings of SPIE: Wavelet Applications in Industrial Processing III*,

volume 6001, pages 169–179, Boston, Massachusetts, USA, October 2005. SPIE.

[78] Hendrik Eeckhaut, Mark Christiaens, and Dirk Stroobandt. Improving external memory access for Avalon systems on programmable chips. In *FPL'07, 17th International Conference on Field Programmable Logic and Applications*, August 2007.

[79] Hendrik Eeckhaut, Harald Devos, Philippe Faes, Mark Christiaens, and Dirk Stroobandt. FPGA design methodology for a wavelet-based scalable video decoder. In Stamatis Vassiliadis, Mladen Berekovic, and Timo Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 169–178, Samos, Greece, July 2007. Springer.

[80] Hendrik Eeckhaut, Harald Devos, Peter Lambert, Davy De Schrijver, Wim Van Lancker, Vincent Nollet, Prabhat Avasare, Tom Clerckx, Fabio Verdicchio, Mark Christiaens, Peter Schelkens, Rik Van de Walle, and Dirk Stroobandt. Scalable, wavelet-based video: from server to hardware-accelerated client. *IEEE Transactions on Multimedia*, 9(7):1508–1519, November 2007.

[81] Hendrik Eeckhaut, Harald Devos, and Dirk Stroobandt. The energy scalability of wavelet-based, scalable video decoding. In *17th International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, volume 4644 of *Lecture Notes in Computer Science*, pages 363–372, Göteborg, Sweden, September 2007. Springer-Verlag.

[82] Hendrik Eeckhaut, Dirk Stroobandt, Harald Devos, and Mark Christiaens. Improving the hardware friendliness of a wavelet based scalable video codec. *WSEAS Transactions on Systems*, 4(5):625–634, May 2005.

[83] Paul Feautrier. Array expansion. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 429–441, New York, NY, USA, 1988. ACM Press.

[84] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[85] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[86] D. D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *Design & Test of Computers, IEEE*, 11(4):44–54, Winter 1994.

[87] Daniel D. Gajski and Robert H. Kuhn. New VLSI tools. *Computer*, 16(12):11–14, December 1983.

[88] Jürgen Garloff. The Bernstein expansion and its applications. Tutorial paper. Available from `http://www-home.fh-konstanz.de/~garloff/BeExAppl.doc`.

[89] Jürgen Garloff. Convergent bounds for the range of multivariate polynomials. In *Proceedings of the International Symposium Interval Mathematics*, volume 212 of *Lecture Notes in Computer Science*, pages 37–56, September 1985.

[90] Sylvain Girbal. *Optimisations d'applications - Composition de transformations de programme: modèle et utils*. PhD thesis, Université de Paris-Sud, 2005.

[91] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, 2006.

[92] S. Gopalsamy, Dilip Khandekar, and S. P. Mudur. A new method of evaluating compact geometric bounds for use in subdivision algorithms. *Computer Aided Geometric Design*, 8(5):337–356, 1991.

[93] Anne-Claire Guillou, Patrice Quinton, and Tanguy Risset. Hardware synthesis for systems of recurrence equations with multidimensional schedule. *International Journal of Embedded Systems*, 2005. To appear.

[94] Anne-Claire Guillou, Patrice Quinton, Tanguy Risset, and Daniel Massicotte. Automatic design of VLSI pipelined LMS architectures. In *International Conference on Parallel Computing in Electrical Engineering, PARELEC 2000*, pages 144–149, Aug 2000.

[95] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, pages 249–256. ACM Press, July 2004.

[96] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized generation of data-path from C codes for FPGAs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 162–170, New York, NY, USA, 2004. ACM Press.

[98] Sumit Gupta, Rajesh Gupta, Nikil Dutt, and Alexandru Nicolau. *SPARK, A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[99] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, 23(2):302–312, February 2004.

[100] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, Shih-Wei Liao, and E. Bu. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, December 1996.

[101] Frank Hannig, Hritam Dutta, and Jürgen Teich. Mapping a Class of Dependence Algorithms to Coarse-grained Reconfigurable Arrays: Architectural Parameters and Methodology. *International Journal of Embedded Systems*, 2(1/2):114–127, 2006.

[102] Frank Hannig and Jürgen Teich. Design Space Exploration for Massively Parallel Processor Arrays. In Victor Malyshkin, editor, *Proceedings of the Parallel Computing Technologies, 6th International Conference (PaCT)*, volume 2127 of *Lecture Notes in Computer Science*, pages 51–65, Novosibirsk, Russia, September 2001. Springer.

[103] Reiner W. Hartenstein, Michael Herz, Thomas Hoffmann, and Ulrich Nageldinger. Mapping applications onto reconfigurable Kress Arrays. In *FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications*, pages 385–390, London, UK, 1999. Springer-Verlag.

[104] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *Proceedings of VLSI*, pages 595–619, Lisbon, Portugal, December 1999.

[105] James C. Hoe and Arvind. Operation-centric hardware description and synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(9):1277–1288, September 2004.

[106] Chao-Tsung Huang, Po-Chih Tseng, and Liang-Gee Chen. Analysis and VLSI architecture for 1-D and 2-D discrete wavelet transform. *IEEE Transactions on Signal Processing*, 53(4):1575–1586, April 2005.

[107] Ralf Hungerbühler and Jürgen Garloff. Bounds for the range of a bivariate polynomial over a triangle. *Reliable Computing*, 4(1):3–13, 1998.

[108] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the Ptolemy project. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, July 2003.

[109] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies. In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, volume 1, pages 202–207. IEEE Computer Society, 2004.

[110] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proceedings of International Federation for Information Processing Congress 74, (IFIP)*, pages 471–475, Stockholm, Sweden, Augustus 1974. North Holland, Amsterdam.

[111] M. Kandemir, J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(2):243–260, February 2004.

[112] Ashraf A. Kassim, Pingkun Yan, Wei Siong Lee, and Kuntal Sengupta. Motion compensated lossy-to-lossless compression of 4-D medical images using integer wavelet transforms. *IEEE Transactions on Information Technology in Biomedicine*, 9(1):132–138, March 2005.

[113] I. Kharitonenko, Xing Zhang, and S. Twelves. A wavelet transform with point-symmetric extension at tile boundaries. *IEEE Transactions on Image Processing*, 11(12):1357–1364, December 2002.

[114] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, February 2007.

[115] Gauthier Lafruit, Francky Catthoor, Jan P.H. Cornelis, and Hugo J. De Man. An efficient VLSI architecture for 2-D wavelet image coding with novel image scan. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):56–68, March 1999.

[116] Gauthier Lafruit, Lode Nachtergaele, Jan Bormans, Marc Engels, and Ivo Bolsens. Optimal memory organization for scalable texture codecs in MPEG-4. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(2):218–243, March 1999.

[117] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, April 2004.

[118] Z. Li, K. Compton, and S. Hauck. Configuration cache management techniques for FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–36, 2000.

[119] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, December 1997.

[120] F. W. Luttmann and T. J. Rivlin. Some numerical experiments in the theory of polynomial interpolation. *IBM J. Res. Develop.*, 9:187–191, 1965.

[121] Stephane G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[122] R. Martin, H.H. Shou, I. Voiculescu, A. Bowyer, and G.J. Wang. Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design*, 19(7):553–587, July 2002.

[123] M. Martonosi, A. Gupta, and T.E. Anderson. Tuning memory performance of sequential and parallel programs. *IEEE Computer*, 28(4):32–40, April 1995.

[124] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[125] Bingfei Mei, Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkest, and Rudy Lauwereins. Architecture exploration for a reconfigurable architecture template. *IEEE Design & Test of Computers*, 22(2):90–101, March-April 2005.

[126] Bingfeng Mei, Serge Vernalde, Hugo De Man, and Rudy Lauwereins. Design and optimization of dynamically reconfigurable embedded system. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada USA, June 2001.

[127] Benoît Meister. *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization.* PhD thesis, Université Louis Pasteur, Strasbourg, France, December 2004.

[128] Benoît Meister. Approximation of polytope enumerators using linear expansions, 2007. Manuscript in preparation.

[129] Vijay Menon, Keshav Pingali, and Nikolay Mateev. Fractal symbolic analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):776–813, 2003.

[130] Petra Michel, Ulrich Lauther, and Peter Duzy. *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[131] Miguel A. Miranda, Francky V.M. Catthoor, Martin Janssen, and Hugo J. De Man. High-level address optimization and synthesis

techniques for data-transfer-intensive applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):677–686, December 1998.

[132] M. J. Myjak and J. G. Delgado-Frias. A two-level reconfigurable architecture for digital signal processing. *Microelectron. Eng.*, 84(2):244–252, 2007.

[133] Steve Oualline. *Vi IMproved (VIM)*. New Riders, 2001.

[134] Emre Özer, Andy P. Nisbet, and David Gregg. Automatic customization of embedded applications for enhanced performance and reduced power using optimizing compiler techniques. In *Proceedings of the 10th European Conference on Parallel Computing (Europar 04)*, volume 3149 of *LNCS*, pages 318–327, August 2004.

[135] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandecappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, 2001.

[136] Louis-Noël Pouchet, Cédric Bastoul, John Cavazos, and Albert Cohen. A note on the performance distribution of affine schedules. In *2nd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'08)*, 2008.

[137] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International Symposium on Code Generation and Optimization*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society.

[138] Louis-Noël et al. Pouchet. Scalable empirical search of multidimensional schedules. 2008. Accepted for PLDI'08 (Programming Language Design and Implementation), Tucson, Arizona.

[139] H. Ratschek and J. Rokne. *Computer Methods for the Range of Functions*. Halsted Press, Wiley, New York, 1984.

[140] Bob Ramakrishna Rau and Michael S. Schlansker. Embedded computer architecture and automation. *IEEE Computer*, 34(4):75–83, April 2001.

[141] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):942–991, November 1997.

[142] Theodore J. Rivlin. Bounds on a polynomial. *Journal of Research of the National Bureau of Standards-B. Mathematical Sciences*, 74B(1):47–54, 1970.

[143] Theodore J. Rivlin. *An introduction to the approximation of functions*. Dover books on advanced mathematics. Dover Publications, Inc., New York, 1981. Corr. reprint of the 1969 orig. edition.

[144] J. Rokne. Bounds for an interval polynomial. *Computing*, 18(3):225–240, 1977.

[145] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Proc. 38th IEEE/ACM Design Automation Conference (DAC 2001)*, pages 172–177, June 2001.

[146] Peter Schelkens, Adrian Munteanu, Joeri Barbarien, Mihnea Galca, Xavier Giro-Nieto, and Jan Cornelis. Wavelet coding of volumetric medical datasets. *IEEE Transactions on Medical Imaging*, 22(3):441–458, March 2003.

[147] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B. Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology*, 31(2):127–142, 2002.

[148] Alex Semenov, Albert M. Koelmans, Lee Lloyd, and Alexandre Ykovlev. Designing an asynchronous processor using Petri nets. *IEEE Micro*, 17(2):54–64, March/April 1997.

[149] Jeffrey Sheldon, Walter Lee, Ben Greenwald, and Saman Amarasinghe. Strength reduction of integer division and modulo operations. In *Languages and Compilers for Parallel Computing: 14th International Workshop, LCPC*, volume 2624 of *Lecture Notes in Computer Science*, pages 1–14, Cumberland Falls, Kentucky, August 2001.

[150] Simon J. Smith. Lebesgue constants in polynomial interpolation. *Annales Mathematicae et Informaticae*, 33:109–123, 2006.

[151] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 165–176, New York, NY, USA, June 2002. ACM Press.

[152] Dirk Stroobandt, Hendrik Eeckhaut, Harald Devos, Mark Christiaens, Fabio Verdicchio, and Peter Schelkens. Reconfigurable hardware for a scalable wavelet video decoder and its performance requirements. In A.D. Pimentel and S. Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 203–212, Samos, July 2004. Springer.

[153] Chris Sullivan, Alex Wilson, and Stephen Chappell. Using C based logic synthesis to bridge the productivity gap. In *ASP-DAC '04: Proceedings of the Asia and South Pacific Design Automation Conference 2004*, pages 349–354, Piscataway, NJ, USA, 2004. IEEE Press.

[154] Wim Sweldens and Peter Schröder. Building your own wavelets at home. In *Wavelets in Computer Graphics*, pages 15–87. ACM SIGGRAPH Course notes, 1996.

[155] Jürgen Teich and Lothar Thiele. Exact partitioning of affine dependence algorithms. In E. F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, pages 135–153, Springer, Berlin, March 2002.

[156] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, New York, NY, USA, 2004. ACM Press.

[157] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Solving out-of-order communication in Kahn process networks. *Journal of VLSI Signal Processing Systems*, 40(1):7–18, 2005.

[158] Tanja Van Achteren, Francky Catthoor, Rudy Lauwereins, and Geert Deconinck. Search space definition and exploration for nonuniform data reuse opportunities in data-dominant applications. *ACM Transactions on Design Automation of Electronic Systems*, 8(1):125–139, January 2003.

[159] Sven Verdoolaege. *Incremental Loop Transformations and Enumeration of Parametric Sets*. PhD thesis, KULeuven, April 2005. Advisors: M. Bruynooghe, F. Catthoor.

[160] Sven Verdoolaege. *barvinok: User Guide*, 0.24 edition, June 2007. Available from `http://freshmeat.net/projects/barvinok`.

[161] Sven Verdoolaege, Kristof Beyls, Maurice Bruynooghe, and Francky Catthoor. Experiences with enumeration of integer projections of parametric polytopes. In R. Bodik, editor, *Compiler Construction: 14th International Conference*, volume 3443 of *Lecture Notes in Computer Science*, pages 91–105, Edinburgh, March 2005. Springer.

[162] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[163] Jiacun Wang. *Timed Petri Nets : Theory and Application*. Kluwer Academic Publishers, 1998. ISBN: 0-7923-8270-6.

[164] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, June 1991. ACM Press.

[165] Michael Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[166] Yana Yankova, Koen Bertels, Stamatis Vassiliadis, Roel Meeuws, and Acrilio Virginia. Automated HDL generation: Comparative evaluation. In *Proceedings of International Symposium on Circuits and Systems (ISCAS2007)*, May 2007.

[167] Nikos D. Zervas, Giorgos P. Anagnostopoulos, Vassilis Spiliotopoulos, Yiannis Andreopoulos, and Costas E. Goutis.

Evaluation of design alternatives for the 2-D discrete wavelet transform. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(12):1246–1262, December 2001.

[168] Hongwei Zhu. *Computation of Memory Requirements for Multi-Dimensional Signal Processing Applications*. PhD thesis, University of Illinois at Chicago, 2006. Preliminary Doctoral Thesis.

[169] Hongwei Zhu, Ilie I. Luican, and Florin Balasa. Memory size computation for multimedia processing applications. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 802–807, New York, NY, USA, 2006. ACM Press.

[170] Günter M. Ziegler. *Lectures on Polytopes (Graduate Texts in Mathematics)*. Springer, July 2001.

[171] Claudiu Zissulescu, Bart Kienhuis, and Ed Deprettere. Expression synthesis in process networks generated by LAURA. In *16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 15–21, July 2005.