

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK – ESAT
Arenbergpark 10, B-3001 Leuven — België

PLATFORM INDEPENDENT SOURCE CODE TRANSFORMATIONS FOR TASK CONCURRENCY MANAGEMENT

Promoters:
Prof. G. Deconinck
Prof. F. Catthoor

Dissertation submitted in partial fulfillment of the requirements for the degree of "doctor in de ingenieurswetenschappen"

by

Stefaan HIMPE

January 2006



KATHOLIEKE UNIVERSITEIT LEUVEN
Faculteit Ingenieurswetenschappen
Departement Elektrotechniek – ESAT
Arenbergpark 10, B-3001 Leuven — België

PLATFORM INDEPENDENT SOURCE CODE TRANSFORMATIONS FOR TASK CONCURRENCY MANAGEMENT

Jury:
Prof. G. De Roeck, chairman
Prof. G. Deconinck, promotor
Prof. F. Catthoor, promotor
Prof. M. Moonen, assessor
Prof. B. Demoen, assessor
Prof. M. Bruynooghe
Prof. J. Voeten (T.U.Eindhoven)

Dissertation submitted in partial fulfillment of the requirements for the degree of "doctor in de ingenieurswetenschappen"

by

Stefaan HIMPE

U.D.C. 681.3*C3 and 681.3*H51 January 2006

©Katholieke Universiteit Leuven – Faculteit Ingenieurswetenschappen
Arenbergpark 10, B-3001 Leuven – Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2006/7515/4

ISBN 90-5682-664-6

Acknowledgements

What they undertook to do
They brought to pass;
All things hang like a drop of dew
Upon a blade of grass.

William Butler Yeats - Gratitude To The Unknown Instructors, The Winding Stair and Other Poems, 1933

Gedetailleerd napluizen welke personen en gebeurtenissen er voor gezorgd hebben dat dit boekje, dat tot voor kort slechts een vage droom was, tot stand kon komen zou wellicht (nog) meer tijd vragen dan het schrijven ervan. Het zijn soms factoren die op het moment zelf onbeduidend leken die op een of andere manier toch cruciaal blijken voor het welslagen van dit eindwerk.

Desalniettemin kunnen de bijdragen van een aantal mensen hier niet onvermeld blijven. Vooreerst wens ik mijn promotoren, prof. Geert Deconinck en prof. Francky Catthoor te bedanken. Zonder hun blijvende toewijding, aanmoedigingen en deskundige begeleiding was dit boekje een stille dood gestorven nog voor het geschreven was.

Mijn dank ook aan assessoren prof. Marc Moonen en prof. Bart Demoen voor het nalezen en becommentariëren van eerdere versies van deze tekst, en aan de juryleden prof. Maurice Bruynooghe en prof. Jeroen Voeten voor hun bereidwillige medewerking. Aan prof. Guido De Roeck van harte bedankt voor het voorzitten van de jury bij de verdediging.

Vervolgens wil ik mijn collega's op ESAT vermelden. De constructieve en aangename sfeer in onze groep bleef bewaard doorheen alle herorganisaties en vormde — naast het werk zelf — een extra motivatie om met plezier naar het kantoor te komen. Bedankt Tanja en Tom, beiden begeleiders van mijn ingenieurseindwerk bij ACCA. Zij zijn het die prof. Rudy Lauwereins zeiden dat ik misschien wel een doctoraat zou kunnen maken. Daar ben ik hen nog altijd heel

erg dankbaar voor. Bedankt (alfabetisch) Adi, Axel, Dirk, Enzo, Jan, Koen, Marc, Murali, Nam, Pieter, Rodica, Sias en Yulia voor de aangename tijd die we samen doorgebracht hebben, zowel binnen als buiten de kantooruren.

Daarnaast moet ik zeker ook de collega-doctoraatsstudenten van IMEC bedanken waarmee ik samen mocht werken aan TCM. In het bijzonder wil ik mijn naaste collega's Chun Wong, Peng Yang, Pol Marchal en Richard Stahl vermelden. De regelmatige bijeenkomsten en de samenwerking voor het maken van een aantal demonstratorapplicaties bevatten de kiemen die uiteindelijk zouden uitgroeien tot de bijdragen uit deze thesis. Ook Georgios Keramidas, de Griekse student die ik mee hielp begeleiden tijdens zijn verblijf in IMEC verdient een woord van dank.

Van zeer groot belang was ook de logistieke en administratieve hulp van Gerda, Lut, Natacha en Pela.

Dank ook aan mijn ouders en schoonouders die mij regelmatig thuis kwamen "vervangen" tijdens in de laatste faze van dit eindwerk. Mijn vrouwtje Katrien wens ik te bedanken omdat ze steeds heeft volgehouden dat ik dit doctoraat moest afwerken, ondanks het feit dat het ook voor haar een uitputtende eindspurt geworden is. Bedankt ook voor het in orde helpen krijgen van alle administratie.

Schoonderbuken, January 2006

Stefaan HIMPE.

Abstract

Applications that run in embedded systems nowadays become more and more dynamic. In the typical example of a modern GSM, it should be possible to make a phone call while looking up something in the address book leading to applications with a notion of multi-threading. Starting and stopping such applications leads to a varying need for memory space and processing power. Most GSMs nowadays also offer games which require user interaction. Game objects are created and deleted at the user's request, again leading to a greatly varying processing power. As such applications grow more sophisticated more processing power is needed which results in an increased need for energy. To guarantee user satisfaction such applications have soft real-time constraints associated to them. When watching a video stream, at least 25 frames per second should be decoded and displayed to get smooth animation.

Today's platforms have enormous processing power. Due to the dynamic behavior of the applications, most of this processing power is not needed all the time. The task concurrency management (TCM) methodology which forms the backbone of this dissertation proposes a systematic way of exploiting the dynamic behavior and the application's coarse grained parallelism in the presence of soft real-time constraints to reduce the energy needs.

Exploiting the dynamic behavior and the application's coarse grained parallelism requires a representation of the application which highlights exactly these characteristics, and which hides the details that are not relevant. The MTG* model that emerged from applying several of the TCM steps on real-life application is briefly described. Fully formalizing this model was not the aim of this dissertation and is left as future work.

If coarse grained parallelism is the key aspect of the application that can lead to energy savings it is a logical step to see if we can reveal additional parallelism by systematically transforming the application. Two such transformations are described in this text. The first one is called sequentialization of semaphores. It removes synchronization overhead and creates freedom for parallelization. The proposed transformation is demonstrated on a JAVA MPEG2 video decoding

application.

In the chapter on recursion removal we develop a framework for analysis of recursion and then apply it to the task of replacing recursion with iteration. The resulting iteration can easily be parallelized and systematic trade-offs are possible between the amount of memory used and the execution time. As a side result of applying the recursion analysis framework to the Towers of Hanoi algorithm and an MPEG4 Visual Texture Coding innermost loop in order to replace the recursive algorithm with an iterative one, some remarkable results and physical interpretations are formulated.

The appendices contain some detailed technical information which is less important to understand the main ideas described in the thesis.

Glossary

- $\odot$: function composition operator
- $\oplus$: call tree branch operator
- ACP: Algebra of Communicating Processes
- ADDL: Algorithmic Design Description Language
- API: Application Programmer's Interface
- ASIC: Application Specific Integrated Circuit
- ASIP: Application Specific Instruction set Processor
- BDF: Boolean Data Flow
- CCS: Calculus of Communicating Systems
- CDDF: Cyclo-Dynamic Data Flow
- CDFG: Control Data Flow Graph
- CFSM: Concurrent FSM
- CORBA: Common Object Request Broker Architecture
- CPU: Central Processing Unit
- CSDF: Cyclo-Static Data Flow
- CSP: Communicating Sequential Processes
- DDF: Dynamic Data Flow
- DF*: Data Flow graphs with STAte based Reconfiguration
- DSFG: Decorated Signal Flow Graph
- DSP: Digital Signal Processor
- DTPN: Dual Transition Petri Net
- DTSE: Data Transfer and Storage Exploration methodology
- FIFO: First In First Out
- FSM: Finite State Machine
- FSMD: Finite State Machine with Data path
- GOP: Group Of Pictures layer in MPEG movie encoding hierarchy
- GOPs: Giga Operations

- HCFSM: Hierarchical CFSM
- HDF: Homogeneous Data Flow
- HW: Hardware
- I/O: Input/Output
- IMEC: Interuniversitair Micro-Elektronica Centrum
- JPEG: Joint Pictures Expert Group
- LOTOS: Language of Temporal Ordering Specification
- MPEG: Moving Pictures Expert Group
- MTG: Multi Thread Graph
- PC: Personal Computer
- PDA: Personal Digital Assistant
- PDA: PushDown Automata
- PDF: Parametrized Data Flow
- Petri net based Unified REpresentation
- PIC: Picture
- PN: Petri Net
- QoS: Quality of Service
- RAM: Random Access Machine
- RAM: Random Access Memory
- RTTI: Run-Time Type Information
- SDF: Synchronous Data Flow
- SDL: System Description Language
- SM: Shared Memory
- SoC: System-on-Chip
- SPI: System Property Intervals model
- SW: Software
- TCM: Task Concurrency Management
- TDTSE: Task level DTSE
- TF: MTG* Thread Frame
- TI: Texas Instruments
- UML: Unified Modeling Language
- Vdd: symbol denoting voltage at which a device operates
- VHDL: VHSIC Hardware Description Language
- VHSIC: Very High Speed Integrated Circuits
- VLSI: Very Large Scale Integration
- VTC: Visual Texture Coding

Contents

Acknowledgements **i**

Abstract **iii**

Glossary **v**

Contents **vii**

Korte inhoud **xxi**

Samenvatting **xxiii**

0.1 Inleiding xxiii

0.1.1 Doel van de thesis en toepassingsdomein xxiii

0.1.2 Samenvatting van de bijdragen xxiv

0.1.3 Structuur van de thesis xxv

0.2 Kort TCM overzicht xxvi

0.3 Applicatiemodellering xxvii

0.4 Codetransformaties xxx

0.4.1 Semafoortransformatie xxx

0.4.2 Recursieverwijdering xxxiii

0.5 Conclusies en toekomstig werk xli

I Introduction **1**

1 Thesis contributions **3**

1.1 Context and motivation 3

1.1.1 Introduction 3

1.1.2 Platform-based design 4

1.2 An overview of TCM 6

1.2.1 Overview 6

1.3 Contributions 8

1.3.1 Application representation 8

1.3.2 Concurrency improving transformations 11

1.4 Thesis structure 12

1.5 List of publications 12

1.5.1 Related to MTG* and application modeling 12

References **13**

1.5.2 Related to program transformations 14

References **14**

1.5.3 TCM methodology related 15

References **15**

II Application modeling **17**

2 Related work in application modeling **19**

2.1 Introduction 19

2.2 Modeling computation 20

2.3 Modeling data flow and data communication 22

2.4 Concurrency 25
2.5 Liveness 29
2.6 Use of hierarchy and abstraction 29
2.7 Heterogeneity 31
2.8 Reactivity 33
2.9 Dynamism 34
2.10 Conclusions 34

3 MTG* summary **37**

3.1 Introduction 37
3.2 MTG* task level model syntax 39
3.2.1 MTG* nodes 39
3.2.2 MTG* edges 45
3.3 MTG* task level operational semantics 47
3.3.1 Firing rules: enabled nodes 47
3.3.2 Execution rules 49
3.3.3 Stop cluster 53
3.3.4 Inhibitor arc 54
3.4 MTG* intra task model 54
3.5 Gray box model 54
3.5.1 Thread cluster 55
3.5.2 Thread node 55
3.6 Timing constraints 56
3.6.1 Timing constraints 56
3.6.2 Timing guarantees 58
3.6.3 Functional timing: delay generation 59
3.7 Parallelism 59

3.8 Future requirements 59

III Source code transformations 61

4 Task-level transformations 63

4.1 Problem formulation 63

4.1.1 What is a task level transformation? 63

4.1.2 CDFG transformations enable task level transformations 65

4.2 Related work in program transformation 66

4.2.1 Program analysis and transformation 66

4.2.2 Program transformations in hardware/software codesign context 66

4.2.3 Semaphore transformations 70

4.2.4 Recursion removal transformations 72

4.3 Pareto curves and surfaces 75

4.3.1 Design space and design space points 75

4.3.2 Pareto optimality 75

4.3.3 Costs versus constraints 76

4.4 Transformation classification 77

4.5 Conclusions 78

5 Semaphore transformations 81

5.1 Introduction 81

5.2 Unconditional sequentialization of semaphores 82

5.2.1 Introduction 82

5.2.2 Information hiding and analysis 82

5.2.3 Sequentialization transformation 83

5.2.4 Trade-offs 84

5.3 Example: JPEG decoder 86
5.3.1 introduction 86
5.4 Example: JAVA MPEG 1 video decoder 88
5.4.1 MPEG 1 bitstream format 88
5.4.2 Video semaphore 90
5.4.3 Video semaphore transformation 91
5.4.4 Dispatch semaphore 93
5.4.5 Profiling results 93
5.5 Conclusions 95

6 Recursion removal **97**

6.1 Introduction 97
6.2 Basic method and extensions 98
6.2.1 Relation between basic method and its extensions . . . 98
6.2.2 Basic method on motivating example 100
6.2.3 Propagation equation and path evaluation equation . . 102
6.3 Comparison of iterative and recursive control flow 109
6.3.1 Extension 1: leaves at non-constant depth 110
6.3.2 Extension 2: intermediate function calls 111
6.3.3 Extension 3: multiple base cases and multiple recursive cases 119
6.3.4 Extension 4: side effects 122
6.3.5 Extension 5: functions with return values 126
6.3.6 Trade-offs and performance impact of using the different extensions 128
6.4 Practical results and discussion 132
6.4.1 MPEG4 visual texture coding (VTC) 132
6.4.2 Profiling results for vtc 132

6.5 Conclusions 135

IV Conclusions 137

7 Conclusions and Future work 139

7.1 Conclusions 140

7.2 Future work 142

7.2.1 Future work related to modeling 142

7.2.2 Future work related to transformations 143

Curriculum Vitae 145

V Appendices 147

A Gray box model of MPEG4 QoS demonstrator 149

B TCM transformation classification 157

B.1 Introduction 157

B.1.1 TCM instantiation of abstracted DTSE steps 159

B.2 Pareto curves and surfaces 163

B.2.1 Anatomy of a Pareto curve 163

B.2.2 Multiple constraints and multiple costs 164

B.3 Transformation classification 170

B.4 Getting transformation feedback from Pareto curves 173

B.4.1 Cdfg transformations 173

B.4.2 Task-level transformations 175

B.5 Conclusions 177

C Issues and future work 179

C.1 Current issues 179
C.1.1 Application domain type 179
C.1.2 Thread frame boundaries 181
C.1.3 Scenarios and subscenarios 181
C.2 Future work 183
C.2.1 Future TCM work 183
C.2.2 Interaction with other parts of the unified meta flow . . 185

D Recursion removal source code **187**
D.1 BaseBCounter implementation 187
D.2 General recursion removal example 188

References **195**

List of Figures

0.1 Samenvatting TCM xxviii

0.2 Het maken van een synchronizatiekaart. xxxii

0.3 Synchronizatiekaartpatronen. xxxii

0.4 Een voorbeeldalgoritme xxxvi

0.5 Voorbeeld van een functieaanroepboom xxxvi

0.6 Torens van Hanoi xxxviii

0.7 Recursieve MPEG4 textuurdecodeerapplicatie. xl

0.8 Hardwareimplementatie van de argumenttransformaties. xli

1.1 TCM flow 9

4.1 Example point cloud; Pareto points are indicated with squares 76

4.2 Pre-processing and transformation phases in TCM transformation flow. 79

5.1 Graph transformations. Edge guard expressions are preserved if not mentioned explicitly. 83

5.2 Synchronization map patterns. 84

5.3 First sequentialization possibility. 85

5.4 Second sequentialization possibility. 85

5.5 Third sequentialization example. 85

5.6 JPEG decoder task level model. 87

5.7 JPEG decoder synchronization map. 88

5.8 System bitstream format. 89

5.9 Video bitstream format. 90

5.10 Audio bitstream format. 90

5.11 Model for calling get_long from a call site. 92

6.1 An example algorithm 98

6.2 Example call tree 100

6.3 Iterative version of the example algorithm. Only valid within the assumptions stated in the text. 101

6.4 g(2,5) call graph. Leaves are drawn with a bold ellipse. 105

6.5 Comparing the recursive control flow to the iterative control flow 110

6.6 Algorithm with leaves at non-constant depth 110

6.7 Iterative version with non-constant leaf depth 111

6.8 An example algorithm 112

6.9 Call tree annotated with intermediate function activation . . . 112

6.10 Python implementation for intermediate function activation without g_b 114

6.11 Python implementation for intermediate function activation with g_b 115

6.12 Towers of Hanoi problem 116

6.13 Multiple recursive cases: nodes have different number of children depending on recursive case 121

6.14 Multiple recursive cases: effect of adding dummy calls 123

6.15 An example algorithm 124

6.16 An example algorithm 127

6.17 Assignment of recursive algorithm with b=3 to 3 processors. Arrows are labeled with the processor they are assigned to. . . 131

6.18 Recursive MPEG4 texture decoder. 133

6.19 Hardware implementation of the argument transformations. . . 135

A.1 Notation summary 150

A.2 MTG* task level model of MPEG4 QoS demonstrator. 151

A.3 Gray box model of Quality Adjust thread frame. 152

A.4 Gray box model of Scene Update thread frame. 153

A.5 Gray box model of Post Update thread frame. 154

A.6 Gray box model of Initialization thread frame. 155

A.7 Gray box model of System Release thread frame. 156

B.1 Influence of cost factor classification on Pareto surface 164

B.2 3D pareto surface calculated from the given design space points. 172

B.3 Pre-processing and transformation phases in TCM transformation flow. 174

B.4 Intersection points and Pareto curve merging. 176

List of Tables

0.1 Profiling resultaten voor de MPEG2 JAVA semafoor sequentialisatie. xxxiii

0.2 Tijdsmeting VTC decodeerapplicatie xli

3.1 Overview of possible number of entry/exit control ports and data ports in MTG* nodes 44

5.1 Profiling results for MPEG JAVA semaphore sequentialization. 95

6.1 Intermediate function activation. 116

6.2 Time measurement VTC decoder implementations 134

B.1 2D (t, e) example Pareto points 163

B.2 3D (t, e, s) example Pareto points 165

B.3 t,e regions and resulting s for given design space points 170

B.4 t,s regions and resulting p for given design space points 171

B.5 p,s regions and resulting t for given design space points 171

Korte inhoud

De applicaties die tegenwoordig op ingebedde systemen gebruikt worden, vertonen steeds vaker dynamisch gedrag. Neem bijvoorbeeld een GSM. GSM gebruikers verwachten dat ze kunnen telefoneren en ondertussen iets opzoeken in hun ingebouwde agenda. Dit leidt tot applicaties waarin verschillende taken tegelijk actief moeten kunnen zijn. Het starten en stoppen van die taken leidt tot een variërende behoefte aan geheugen en rekenkracht. De meeste GSM's bieden tegenwoordig ook spelletjes aan. Spelletjes vereisen gebruikersinteractie. Op commando van de gebruiker worden nieuwe spelobjecten geïntroduceerd of verwijderd waardoor de vereiste rekenkracht dynamisch varieert. Dergelijke applicaties worden steeds geavanceerder en vereisen steeds meer rekenkracht en geheugen, wat resulteert in een verhoogde energiebehoefte. Om de gebruiker een aangename ervaring te garanderen bevatten dergelijke applicaties tijdsbeperkingen. Bij het bekijken van een video bijvoorbeeld, is het wenselijk om minstens 25 beelden per seconde te kunnen afbeelden.

Hedendaagse platformen beschikken over enorme rekenkracht. Maar door het dynamisme in de applicaties is niet al die rekenkracht voortdurend nodig. De TCM methodologie (task concurrency management, methodologie voor het beheer van gelijktijdig lopende taken) waarop deze thesis steunt doet een aantal voorstellen om op een systematische manier het dynamisch gedrag en het aanwezige taakparallellisme in aanwezigheid van tijdsbeperkingen te benutten om de energiebehoefte van de applicaties te verlagen.

Het exploreren van het dynamisch gedrag en van het taakparallellisme vereist een representatie van de applicatie die net die aspecten van een applicatie in de verf zet, en die toelaat om details die voor ons doel minder relevant zijn te verbergen. Het MTG* model dat kort in deze thesis beschreven wordt is ontstaan door toepassing van de TCM stappen op dynamische multimedia-applicaties. De volledige formalizatie van MTG* was niet de bedoeling van het eindwerk en wordt overgelaten als toekomstig werk.

Als taakparallellisme een sleutelrol vervult bij het optimalizeren van de applicatie met betrekking tot energieverbruik is het een logische stap om te zien of

we er op een of andere manier kunnen in slagen om extra parallellisme bloot te leggen door systematische transformatie van de applicatie. Twee dergelijke transformaties worden beschreven in deze tekst. De eerste wordt sequentialisering van semaforen genoemd. De transformatie verwijdert overhead en creëert vrijheid voor parallellizatie. De transformatie wordt toegepast op een JAVA MPEG2 videodecodeerapplicatie.

In het hoofdstuk over recursieverwijdering wordt eerst een raamwerk voor de analyse van recursie ontwikkeld dat daarna toegepast wordt op het probleem van het vervangen van recursie door iteratie. De resulterende iteratie kan gemakkelijk geparallellizeerd worden en systematische afwegingen met betrekking tot geheugengebruik en hoeveelheid berekeningen kunnen gemaakt worden tijdens de transformatie. Als nevenresultaat van de toepassing van het recursieanalyseraamwerk op het 'Torens van Hanoi' en 'MPEG4 VTC' algoritme worden een aantal merkwaardige resultaten en fysische interpretaties van de corresponderende iteratieve implementaties gevonden.

De appendices bevatten een deel gedetailleerde technische informatie die minder belangrijk is voor het volgen van de gedachtengang doorheen de thesis.

Samenvatting

0.1 Inleiding

0.1.1 Doel van de thesis en toepassingsdomein

De hedendaagse processorplatformen die gebruikt worden in ingebedde systemen worden steeds krachtiger om de steeds dynamischer wordende toepassingen te kunnen verwerken. Dynamisme in dergelijke toepassingen wordt veroorzaakt door het creëren en activeren van van gelijktijdig lopende taken. Een gebruiker van een GSM verwacht dat hij zijn adresboek kan raadplegen zonder de telefoonverbinding te moeten verbreken. Binnen de verschillende toepassingen op de GSM bestaan vereisten met betrekking tot reële-tijdsgedrag. Iemand die een videostroom bekijkt op zijn GSM in een teleconferentietoepassing wil 25 beelden per seconde zien om een vloeiend bewegend beeld te hebben.

Om ervoor te zorgen dat die 25 beelden per seconde gehaald worden, zelfs wanneer ondertussen plots de agenda opspringt om te waarschuwen voor een naderende vergadering, worden de processorplatformen heel erg krachtig gemaakt. Net door dat dynamisch gedrag is die rekenkracht eigenlijk niet voortdurend nodig. Task concurrency management (TCM, vrij vertaald: beheer van gelijktijdig lopende taken) biedt een manier om dat dynamisch gedrag van applicaties te gaan gebruiken om het energieverbruik te verminderen. In de gangbare Quality of Service aanpak wordt de processorkracht altijd zo volledig mogelijk gebruikt om de subjectieve kwaliteit die een gebruiker van een toepassing ervaart te maximaliseren. In een TCM context worden kosten en baten afgewogen. De kost die met TCM geminimaliseerd wordt is energieverbruik, ten voordele van verminderde executietijd en in een geavanceerdere vorm ook de gepercipieerde kwaliteit.

TCM draait rond dit idee: om aan de tijdsvereisten te voldoen moet een bepaalde taak binnen een zekere tijd afgewerkt zijn. Als meerdere stukjes functionaliteit tegelijk kunnen uitgevoerd worden op verschillende processoren, mogen elk van die stukjes afzonderlijk trager werken zonder dat de executietijd langer

wordt. Omdat het energieverbruik kwadratisch stijgt met de kloksnelheid van de processoren kan op die manier energiewinst gehaald worden.

Iets concreter dienen tijdens het ontwerp van de applicatie en het platform zoveel mogelijk optimalizaties te gebeuren. Er wordt ook een zekere vrijheid gelaten voor het nemen van beslissingen terwijl de applicatie uitgevoerd wordt. Zo kan bijvoorbeeld een processor in een tragere, energie-efficiëntere toestand werken zolang er geen nood is aan meer processorkracht. Wanneer plots een nieuwe taak geactiveerd wordt, kunnen de werkingspunten van de verschillende processoren opnieuw geëvalueerd worden en kan de processor tijdelijk naar een snellere, maar minder energie-efficiënte toestand worden omgeschakeld. Taken kunnen eventueel herverdeeld worden over de verschillende processoren in het systeem. Het moge duidelijk zijn dat de applicatie en het operating systeem met deze ideeën in het achterhoofd moeten geschreven zijn (of meer realistisch: aangepast worden).

0.1.2 Samenvatting van de bijdragen

Toepassen van TCM op een applicatie start meestal van een C++ of JAVA beschrijving van die applicatie. Dergelijke beschrijving is eigenlijk niet erg geschikt als basis voor toepassen van TCM. In de plaats daarvan zouden we een model willen dat net de dynamische eigenschappen van de applicaties in de verf zet, en waarin de minder interessante details verborgen worden. Het moet ook toelaten dat extra kennis over de applicatie, bijvoorbeeld in verband met tijdsinformatie, geannoteerd wordt. Het MTG* model dat kort in deze thesis beschreven wordt voldoet aan deze vereisten. Door de applicatie te modelleren in MTG* — een taak die nu nog vooral manueel dient te gebeuren — worden de aspecten die voor TCM van belang zijn in de verf gezet. De beschrijving van MTG* in deze thesis is een eerste poging om los te komen van de informele blokdiagramma's die tot voor kort voor TCM gebruikt werden. MTG* kan worden gezien als een opvolger van MTG, een model dat op IMEC al geruime tijd in gebruik is voor implementatie van DSP toepassingen. Volledige formalisatie en analyse van MTG* is echter niet de bedoeling van de tekst. Eerder is MTG* het pragmatische resultaat van het verzamelen van de gemeenschappelijke vereisten van vele experimenten op C++ en JAVA code.

Aangezien het uitbuiten van parallellisme in de applicatie volgens TCM de belangrijkste manier is om energieverbruik te verminderen is het belangrijk om parallellisme te vinden en additioneel parallellisme bloot te leggen. Hiertoe wordt in TCM voorgesteld om programmatransformaties toe te passen. Zoals tijdens het onderzoek duidelijk geworden is vormen de zuivere parallellisatietransformaties slechts een klein gedeelte van de voor TCM nuttige transformaties. Andere transformaties worden gebruikt om vrijheid te creëren voor die parallelisatietransformaties.

In de tekst worden twee transformaties besproken. De eerste transformatie gaat synchronizatie verwijderen. Twee stukken code die tegelijk kunnen uitgevoerd worden, en die gegevens delen kunnen onverwachte resultaten produceren wanneer het ene stuk code de gegevens verandert terwijl het ander stuk code de gegevens gebruikt. Om dit te voorkomen wordt een een semafoor gebruikt. Een semafoor is een mechanisme waaraan een stuk code toestemming moet vragen om te kunnen starten. Die toestemming wordt dan pas gegeven wanneer geen enkel ander stuk code dat door diezelfde semafoor beschermd wordt aan het uitvoeren is. Dergelijke semafoor verhindert parallelle uitvoering van code. In het stukje over semafoortransformaties worden bepaalde manieren van sempahoorgebruik omgezet naar code die sequentieel uitgevoerd wordt, waardoor de semafoor eigenlijk niet meer nodig is en kan verwijderd worden. De methode wordt geïllustreerd op een JAVA programma. Aangezien synchronisatie in JAVA duur is (in termen van executietijd en dus ook voor energieverbruik) kunnen op die manier mooie winsten gemaakt worden.

De tweede transformatie zet recursie om naar iteratie. Om dit te kunnen doen wordt eerst een raamwerk gemaakt dat kan gebruikt worden om recursie te analyseren. Dit raamwerk wordt vervolgens ingezet om niet-lineaire, niet-wederzijdse recursieve algoritmes op systematische manier om te zetten naar iteratieve algoritmes. Er wordt vertrokken van een eenvoudig recursief algoritme dat systematisch uitgebreid wordt met extra complicaties en oplossingen om die complicaties te verwerken. Het vermelden waard is dat de iteratieve algoritmes die op die manier verkregen worden geen gebruik maken van een "stapel" gegevensstructuur. De iteraties die door de methode verkregen worden introduceren geen nieuwe lusafhankelijkheden indien dat niet willen, en indien de verschillende functieaanroepen geen neveneffecten hebben die onrechtstreeks lusafhankelijkheden veroorzaken, wat toelaat om de lussen te paralleliseren. De methode laat toe om systematische afwegingen te maken tussen hoeveel geheugen er zal gebruikt worden versus hoeveel instructies er moeten uitgevoerd worden. Als nevenresultaat leidt de toepassing van het recursieanalyseraamwerk op een aantal recursieve algoritmes tot een aantal verrassende resultaten en interpretaties van iteratieve versies van bestaande algoritmes.

0.1.3 Structuur van de thesis

De thesis is opgedeeld in vier delen en daarnaast nog een aantal appendices.

Het eerste deel bevat vooral introductiemateriaal. De context waarin het onderzoek plaatsvond wordt bondig uiteengezet in hoofdstuk 1. Het hoofdstuk bevat een kort overzicht van de hele TCM methodologie zoals ze vandaag bestaat. Een aantal zaken die recent aan het veranderen zijn worden pas in de appendices vermeld om het geheel niet te complex te maken. Deze veranderingen zijn niet essentieel om de algemene ideeën achter TCM te begrijpen.

Deel twee bekijkt het MTG* model zoals het vandaag gebruikt wordt in TCM (hoofdstuk 3). Voorafgaand aan die beschrijving komt een stukje gerelateerd werk over applicatiemodellering in de wereld van de ingebedde systemen (hoofdstuk 2).

Broncodetransformaties — het hoofdonderwerp van deze thesis — komen aan bod in deel drie. Er wordt gestart met een stukje gerelateerd werk (hoofdstuk 2), in hoofdzaak gefocust op de semafoortransformaties (hoofdstuk 5) en recursietransformaties (hoofdstuk 6) die later in dit deel aan bod komen.

Conclusies en toekomstig werk gerelateerd aan deze thesis worden geformuleerd in deel vier.

De appendices bevatten gedetailleerde technische informatie die niet strikt noodzakelijk in de tekst moet voorkomen om de gedachtengang te kunnen volgen.

0.2 Kort TCM overzicht

Deze sectie introduceert kort de TCM (Task Concurrency Management, ofwel: Methodologie voor het beheer van gelijktijdig lopende taken). Deze methodologie wordt ontwikkeld en verfijnd in IMEC door een team van doctoraatsstudenten. Mensen die nauw bij TCM betrokken waren zijn onder andere Peng Yang, Chun Wong, Pol Marchal, Zhe Ma en ikzelf. Dit onderzoek staat onder technische supervisie van prof. Dr. ir. Francky Catthoor.

TCM vormt één stap in een geheel van stappen die samen een volledige systeemontwerpmethodologie vormen. TCM bestaat zelf uit een aantal substappen die hier kort samengevat worden. Over praktisch elk van deze substappen werd (minstens) één doctoraat geschreven.

1. Eerst wordt een gray box beschrijving van de applicatie gemaakt. Hiervoor wordt gestart van C++ of JAVA broncode. Het gray box model representeert een applicatie als een collectie MTG* thread frames die zelf uit MTG* thread nodes bestaan. Gray box modellering is een manier om enkel die zaken in de verf te zetten die voor TCM van belang zijn.

2. Vervolgens worden transformaties toegepast op de broncode, met behulp van de gray box beschrijving. Deze transformaties moeten er uiteindelijk voor zorgen dat de totale systeemkost omlaag kan gaan. Om dit te bereiken wordt het beschikbare parallellisme expliciet getoond en indien mogelijk wordt extra parallellisme gecreëerd. Twee dergelijke transformaties vormen het hoofdonderwerp van deze thesis. De eerste transformatie gaat iets veranderen aan het synchronizatiegedrag van de applicatie. De

tweede transformatie zet recursie om naar iteratie. Deze transformaties worden respectievelijk beschreven in hoofdstukken 5 en 6.

3. De derde stap bestaat uit een twee-fasen scheduling aanpak.
   (a) In een eerste deelstap gebeurt voor elk thread frame in het systeem een characterisatie met behulp van zogenaamde Pareto curves. Deze curves beschrijven de optimale afwegingen die kunnen gemaakt worden tussen benodigde energie en verwachte uitvoeringstijd. Elk punt op de curve stelt aldus een optimale toekenning van de thread nodes in een thread frame aan de processoren op het platform voor.
   (b) Vervolgens wordt een applicatie-specifieke run-time scheduler geïntegreerd in het real-time operating systeem dat op het platform draait. De run-time scheduler kiest tijdens de uitvoering voor de actieve thread frames een mogelijk werkingspunt van elk van de Pareto curves om zo tot een globaal energie-efficiënt systeem te komen. Als nieuwe taken geactiveerd worden, kan de run-time scheduler nieuwe werkingspunten kiezen voor de bestaande thread frames. Het kiezen van een werkingspunt kan bijvoorbeeld inhouden dat een processor op een hoger voltage begint te werken, zodat hij sneller werkt, maar tegelijk meer energie nodig heeft.

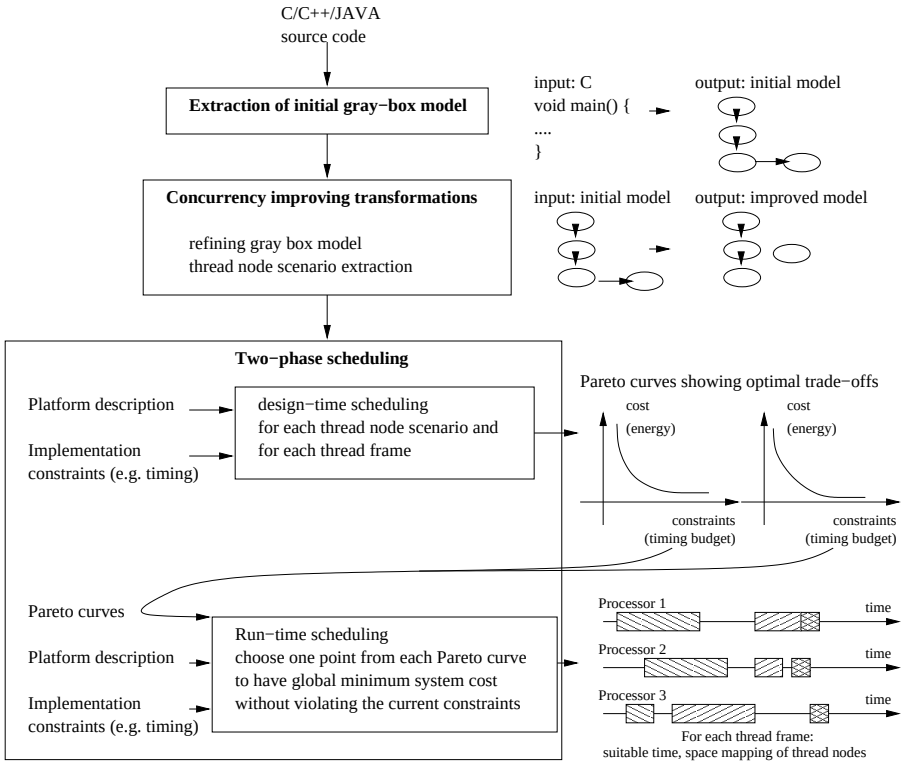
De TCM methodologie wordt samengevat weergegeven in figuur 0.1.

0.3 Applicatiemodellering

TCM haalt energiewinst in vergelijking met state of the art methodes door verregaande exploitatie van het parallellisme en het dynamische gedrag van programma's. Om dit mogelijk te maken op een systematische manier beschikken we liefst over een voorstelling van het programma die net dit parallellisme en dynamische gedrag in de verf zet, en de minder belangrijke details verbergt. Bovendien zouden we graag extra kennis kunnen annoteren op die voorstelling, bijvoorbeeld kennis in verband met tijdsbeperkingen of tijdsgaranties. Zuivere C, C++ of JAVA code, waarvan in de praktijk gestart wordt, is hiervoor niet echt geschikt.

Het MTG* model is hiervoor wel geschikt. Het is oorspronkelijk ontstaan als een informeel blokdiagramma dat gebruikt werd tijdens de design-time scheduling stap van TCM. Ook voor de run-time preparation stap van TCM was duidelijk dat een geschikte voorstelling van het taakparallelisme nodig was. Mettertijd werd duidelijk dat MTG* eigenlijk best een uitbreiding van het bestaande MTG (**M**ulti **T**hread **G**raph) model [196] kon worden. Die uitbreidingen zijn gebaseerd op het DF* (**D**ata **F**low graphs with **STA**te machine based **R**econfiguration) model [52, 154].

Figuur 0.1 Samenvatting TCM



MTG* beschrijft een applicatie in twee niveaus. Het meest abstracte niveau beschrijft het volledige dynamische gedrag van de applicatie. Dit niveau wordt het taakniveau genoemd. Het dynamisch gedrag van de applicatie wordt beschreven als een collectie van MTG* thread frames. Dit zijn zo groot mogelijke stukken functionaliteit die afzonderlijk kunnen geactiveerd worden, en waarin niets gebeurt dat aanleiding zou kunnen geven tot het activeren van de TCM run-time scheduler [221]. In de praktijk komt het erop neer dat binnen in een thread frame geen synchronizatie gebeurt, dat er geen dynamische taken gecreëerd worden en dat er geen events worden afgehandeld. Al deze zaken worden expliciet buiten de thread frames getoond. Al de rest wordt verborgen binnen in de thread frames. Het taakmodel dient dan ook als startpunt voor de run-time scheduling preparation stap van TCM: het taakmodel toont duidelijk wanneer de run-time scheduler moet worden geactiveerd. Taakniveautransformaties resulteren in een veranderd taakmodel met de bedoeling ofwel meer vrijheid te creëren voor de run-time scheduler, ofwel het gedrag minder dynamisch te maken om de run-time overhead te beperken.

Binnen in elke taak worden de details verder beschreven op het tweede niveau: het intrataakniveau. Het model dat hiervoor gebruikt wordt is een dataflow model. In het geval van MTG en MTG* wordt gebruik gemaakt van het DSFG (**D**ecorated **S**ignal **F**low **G**raph) model [130], dat hier niet verder beschreven wordt.

Omdat het DSFG meteen veel extra detail in de beschrijving opneemt wordt het MTG* model ook uitgebreid met het concept van thread nodes. Die vormen een manier om het intrataak model te abstraheren zodat daarin een aantal belangrijke details zichtbaar worden en minder belangrijke details verborgen kunnen worden. Dit abstractere intrataakmodel vormt een zogenaamde gray box beschrijving van de applicatie. Dit is het niveau waarop de TCM designtime scheduler [218] zijn werk doet.

Voor deze thesis is het MTG* model belangrijk omdat het ook de basis vormt voor de codetransformaties die het hoofdonderwerp van deze thesis uitmaken. De semafoortransformatie van hoofdstuk 5 bijvoorbeeld maakt gebruik van de informatie in het taakmodel. De recursieverwijderingstransformatie uit hoofdstuk 6 zet een thread node (de recursieve functie als geheel) om naar een lus van kleinere thread nodes (de 'body' van de iteratieve lus). De codetransformaties worden kort besproken in de volgende sectie.

0.4 Codetransformaties

0.4.1 Semafoortransformatie

0.4.1.1 Inleiding

Semaforen zijn een een techniek waarmee in parallel werkende programma's verhinderd wordt dat stukken code die gemeenschappelijke gegevens veranderen en gebruiken verkeerde resultaten leveren. Programmeurs die in een taal als C++ of JAVA een multithreading applicatie schrijven zullen daarom vaak hele methodes beschermen met zo'n mechanisme.

Vooraleer een stuk code beschermd door een semafoor kan starten moet het toestemming krijgen van de semafoor om te starten. De uitvoering van dat stuk code kan niet verdergaan tot wanneer de semafoor toestemming heeft gegeven. Die toestemming wordt pas gegeven als er geen andere code meer aan het uitvoeren is die de semafoor ook om toestemming gevraagd heeft. Als een bepaald stuk code die door een semafoor beschermd wordt zijn taak uitgevoerd heeft zal hij de semafoor hiervan op de hoogte brengen, waarna een ander stuk code de toestemming kan krijgen om met en op de data te werken.

Het gebruik van semaforen draagt op twee manieren bij tot de totale systeemkost. In de eerste plaats verhinderen ze parallelisme. Hierdoor kan het werk over minder processoren verdeeld worden. Die processoren moet dan sneller werken om resultaat binnen de tijdsbeperkingen klaar te hebben. Hierdoor stijgt dan ook het energieverbruik. Een tweede niet onaanzienlijke kost is het resultaat van de overhead van de implementatie van het synchronizatiemechanisme zelf.

0.4.1.2 Gerelateerd werk

In wezen kunnen drie soorten onderzoek naar semaforen onderscheiden worden:

1. Het eerste type, waartoe ook ons werk behoort, kijkt naar hoe en wanneer synchronizatie dient gebruikt te worden. Typische overwegingen hier zijn het minimaliseren van het aantal benodigde semaforen en het aanpassen van hun invloedssfeer ('scope').

   Bijvoorbeeld [26] verwijdert semaforen die data beschermen die maar voor ten hoogste 1 OS thread bereikbaar zijn. In [3] worden volgende transformaties gedaan:

   - Verwijderen van synchronizatie in code die door ten hoogste 1 thread uitgevoerd wordt.
   - Verwijderen van synchronizatie op data structuren die reeds beschermd worden door een eromheenliggende bescherming door een andere semaphore met grotere invloedssfeer.

- Verwijderen van reëntrante synchronizatie. Dit is een semafoor die verschillende keren na elkaar aangevraagd (en achteraf vrijgegeven) wordt, bijvoorbeeld in een recursieve functie.

Het veranderen van de invloedssfeer van semaforen kan het aantal synchronizatieoperaties verminderen ten koste van een verminderd parallellisme. Bijvoorbeeld in [61] worden twee dergelijke transformaties beschreven: 'data lock coarsening' en 'computation lock coarsening'. Data lock coarsening is een transformatie waarbij verschillende van elkaar losstaande data structuren die gewoonlijk tegelijk aangesproken worden door één enkele semafoor worden beschermd. Computation lock coarsening gaat het vragen van de toestemming om te beginnen uitvoeren uit lussen halen en naar een hoger niveau optillen. Een techniek gelijkaardig aan computation lock coarsening werd vroeger reeds door [170] beschreven.

2. Het tweede type onderzoek kijkt naar efficiëntere implementaties van dergelijke beschermende mechanismen [73, 11, 126, 2, 63]. Dit type van werk is volledig complementair aan het voorgaande type.

3. Een derde type onderzoek onderzoekt welke maatregelen kunnen getroffen worden op hardware niveau om het gebruik van software-gebaseerde synchronizatieoplossingen te vermijden [173, 174]. Dit werk is ook volledig complementair aan het onze.

0.4.1.3 Niet-conditionele sequentializering van semafoorgebaseerde code

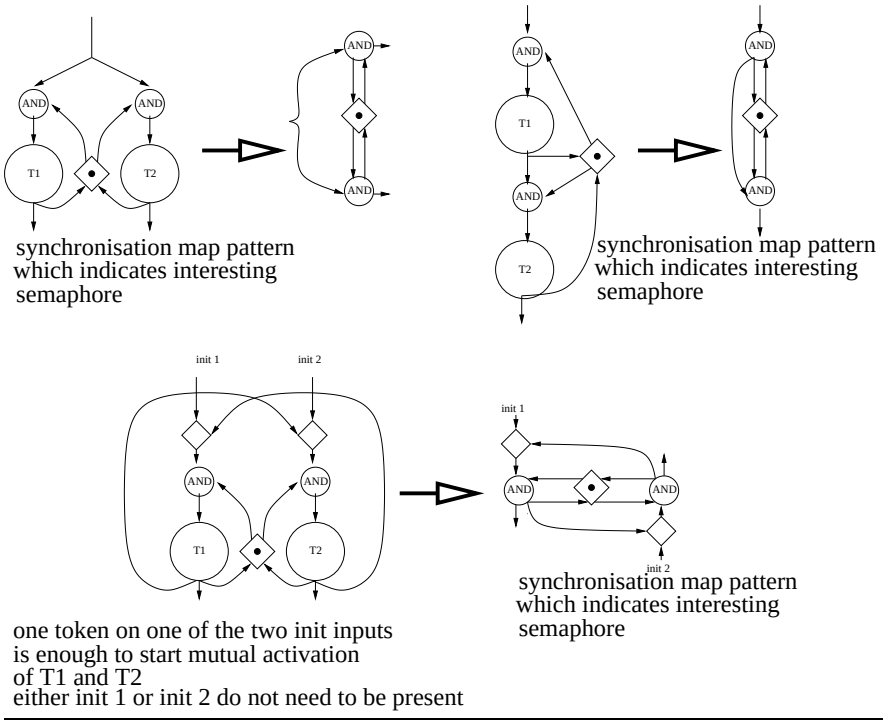
De algemene idee is om bepaalde gebruikswijzen van semaforen te detecteren die kunnen vervangen worden door code die gegarandeerd sequentieel uitgevoerd wordt, waardoor de semafoor niet meer nodig is en kan verwijderd worden.

Om die patronen te detecteren wordt gestart van een MTG* taakniveaumodel. Hierin worden in een eerste stap alle zaken weggelaten die geen synchronizatie impliceren. Het resultaat is een sychronizatiekaart van de applicatie. Een aantal voorbeelden worden getoond in figuur 0.2

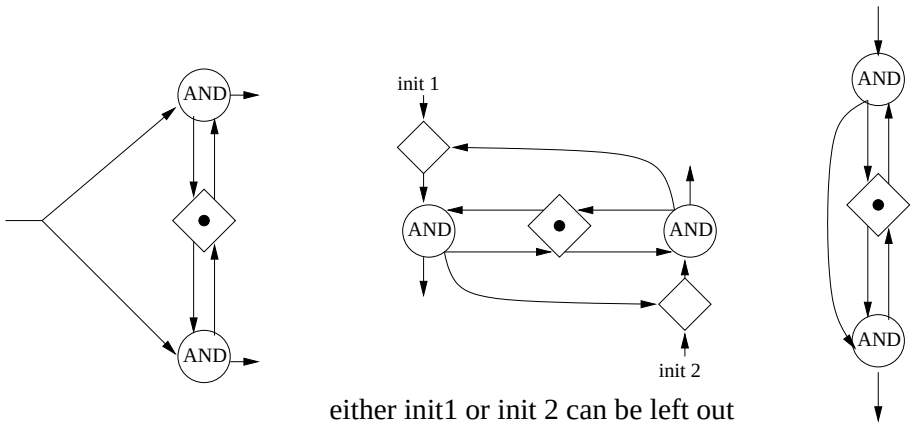
Op die synchronizatiekaart wordt vervolgens naar patronen gezocht die garanderen dat zodra één stuk code die door de semafoor beschermd wordt geactiveerd wordt, alle stukken code die door de semafoor beschermd worden geactiveerd worden. Een aantal patronen worden getoond in figuur 0.3.

De stukken code die tot nu door de semaforen beschermd werden kunnen nu in een sequentiële functie uitgevoerd worden terwijl de semafoor gewoon weggelaten wordt.

Figuur 0.2 Het maken van een synchronizatiekaart.



Figuur 0.3 Synchronizatiekaartpatronen.



aantal byte code instructies	lokaal	globaal
originele versie	160k	25.3M
versie 1	81k	24.7M

Tabel 0.1: Profiling resultaten voor de MPEG2 JAVA semafoor sequentializatie.

0.4.1.4 Voorbeeld: MPEG2 video decoder

Deze techniek werd toegepast op een MPEG2 video decoder en het voornaamste resultaat wordt hieronder samengevat.

- originele versie: dit is de versie zoals ze van het internet gedownload werd
- versie 1: in deze versie werd een semafoor verwijderd met de techniek die in deze sectie samengevat werd

Profiling levert de resultaten getoond in tabel 0.1.

We kunnen zien dat de winst binnen de methode waarin de synchronizatie verwijderd werd aanzienlijk is. Globaal gezien is het aantal uitgevoerde instructies niet zo spectaculair gedaald. De reden hiervoor is dat het programma niet door synchronizatie gedomineerd wordt. De meeste tijd gaat verloren in het echte decoderen van de videobitstroom. Er wordt echter wel verwacht dat het op deze nieuwe versie van het programma toepassen van software pipelining technieken ervoor kan zorgen dat het programma veel efficiënter werkt. Vanwege tijdsgebrek moet dit echter als toekomstige uitbreiding gelaten worden.

0.4.2 Recursieverwijdering

0.4.2.1 Inleiding

Ingewikkelde algoritmes kunnen vaak efficënt gespecifieerd worden met behulp van recursie. Om een recursief algoritme te implementeren wordt meestal gebruik gemaakt van 'stapel' datastructuur. In dit deel van het eindwerk wordt recursie nader bekeken. Er wordt een raamwerk geformuleerd dat kan gebruikt worden om het gedrag van een recursieve functie te analyseren, en dat raamwerk wordt toegepast op het probleem van het omzetten van recursie naar iteratie. Er wordt getoond hoe een brede waaier van niet-lineaire, niet-wederzijdse recursieve algoritmes naar iteratie kunnen worden omgezet zonder gebruik te maken van een stapel. En passant worden de methodes op een aantal algoritmes toegepast, met soms verrassende resultaten tot gevolg.

0.4.2.2 Gerelateerd werk

Recursie is reeds lang het onderwerp van onderzoek en bijgevolg bestaat er al veel literatuur over. In hoofdstuk 4 wordt een uitgebreider overzicht gegeven dan hier.

Zo wordt recursie door compilers geïmplementeerd door het gebruiken van een stapel ("stack"). Dit is een algemene manier om recursieve algoritmes te implementeren. Omzetten van recursie naar iteratie gebeurt soms door het werk van de compiler over te nemen en zelf een stapel in te voegen. Ook is het bekend dat zogenaamde staartrecursie equivalent is met een lus, en in sommige gevallen kan recursie die geen staartrecursie is omgezet worden naar staartrecursie [15].

Onze methode uit hoofdstuk 6 werkt niet zonder wijzigingen voor wederzijdse recursie. Wanneer de recursie wederzijds is (bijvoorbeeld: functie A roept functie B op, en functie B roept functie A op) kan deze soms omgezet worden naar niet-wederzijdse recursie, waarna technieken voor niet-wederzijdse recursie toepasbaar worden [163, 120].

Om recursie om te zetten naar iteratie — meestal met de bedoeling om de uitvoeringstijd te verminderen — wordt vaak gezocht naar bepaalde patronen in de code, of code wordt herschreven tot ze aan een bepaald patroon voldoet [15, 163]. Dit is ook het geval in de methode uit hoofdstuk 6. Voor ons is het verminderen van uitvoeringstijd niet onze eerste en enige bekommernis. In de eerste plaats gaat het om het creëren van vrijheid voor daaropvolgende programmatransformaties.

Veel resultaten in de literatuur in verband met omzetten van recursie naar iteratie werken in een context waar geen neveneffecten kunnen optreden [145, 9, 10]. De methode uit hoofdstuk 6 laat expliciet neveneffecten toe.

Het werk in [140] concentreert zich op het verwijderen van recursie in zeer algemene situaties. Hun basisidee is het identificeren van een veralgemeende incrementele uitvoeringsvolgorde. Hun methode heeft als voordeel een gegarandeerde verbetering in uitvoeringstijd en benodigd geheugen. Nadeel van hun methode is dat de incrementele uitvoeringsvolgorde ervoor zorgt dat parallellisatie van de iteratieve lus niet vanzelfsprekend is: de volgende iteratie kan pas gestart worden als de vorige compleet is.

0.4.2.3 Basismethode en uitbreidingen

Beschouw de code zoals getoond in figuur 0.4. Figuur 0.5 toont wat er gebeurt als de code uit de linkerkolom zou uitgevoerd worden met argumenten x = 2 en y = 0. Bij de uitvoering van het algoritme wordt de daar voorgestelde functieaanroepboom doorlopen in diepte-eerst volgorde. De getallen in de knooppunten stellen de waarden van de argumenten voor zoals ze doorheen de uitvoering van het algoritme getransformeerd worden. Wanneer het recursief algoritme in een blad terecht komt, wordt met de waarden van de argumenten

in dat blad een bewerking uitgevoerd, waarna het algoritme zijn weg vervolgt.

Zoals zichtbaar in figuur 0.5 resulteert elke aanroep van functie bR in ofwel drie ofwel nul nieuwe aanroepen van bR. Drie nieuwe aanroepen zullen gebeuren indien de recursie nog niet stopt, nul nieuwe aanroepen gebeuren wanneer de recursie stopt en er een bewerking met de argumentwaarden plaatsvindt.

De pijlen stellen functieaanroepen voor en zijn als volgt genummerd: elke meest linkse pijl in een groepje van drie pijlen krijgt nummer 0, elke middelste pijl krijgt nummer 1 en elke rechtse pijl krijgt nummer 2. Wanneer een pijl met nummer 0 gevolgd wordt, worden de argument getransformeerd met de argumenttransformatiefuncties zoals die voorkomen in de nulde functieaanroep. De eindpunten van de functieaanroepboom (de 'blaadjes') werden genummerd van 0 tot 8. De ermee corresponderende representatie van het bladnummer in getalbasis 3 (00 tot 22) staat eronder. We kunnen twee zaken vaststellen:

1. De getallen die voorkomen in de representatie van het bladnummer tonen welk pad gevolgd werd doorheen de functieaanroepboom om van aan de top aan het blad te geraken. Met andere woorden: uit niets anders dan het rangnummer van een blad kan de hele weg van de wortel van de functieaanroepboom tot aan het blad gereconstrueerd worden.
2. De reconstructie van een pad van wortel tot blad via de representatie in getalbasis 3 kan voor elk pad tegelijk gebeuren.

Vanuit deze vaststellingen wordt het volgende iteratieve algoritme voorgesteld: maak een teller die in getalbasis 3 telt van $0_{(10)} = 00_{(3)}$ tot $8_{(10)} = 22_{(3)}$. De getalbasis komt overeen met het aantal kinderen van een knooppunt. Het aantal cijfers dat gebruikt wordt om het bladnummer voor te stellen komt overeen met de diepte van de functieaanroepboom. Pas telkens de juiste argumenttransformatiefunctie toe zoals aangegeven door de cijfers in de basis 3 representie van het bladnummer. Zodra de nieuwe argumentwaarden gekend zijn kan de bewerking erop uitgevoerd worden die in het recursieve algoritme uitgevoerd werd op het moment dat de recursie stopt.

Deze methode werkt maar omdat functie bR tamelijk speciaal is:

- De recursie gaat overal even diep, en de diepte is op voorhand bekend. Dit is zichtbaar in de functieaanroepboom waar alle blaadjes op dezelfde hoogte staan;
- Er is maar 1 voorwaarde waaronder de recursie stopt (`x<=0`);
- Er is maar 1 voorwaarde waaronder de recursie verdergaat (`x>0`); item Tussen de verschillende recursieve aanroepen staan geen aanroepen naar andere functies;

Figuur 0.4 Een voorbeeldalgoritme

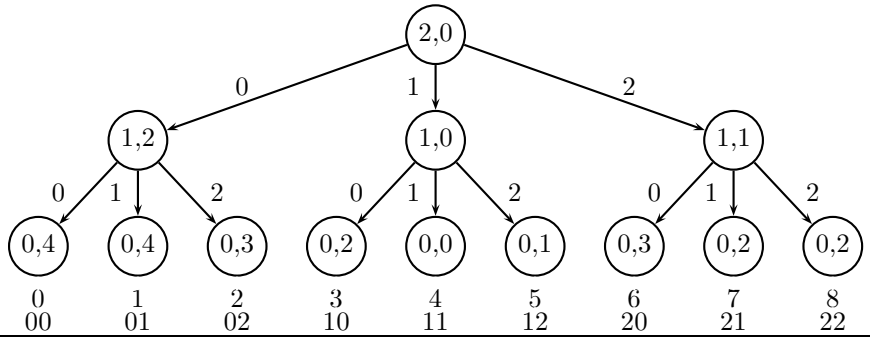
```
1 def bR(x,y):
    if (x<=0):
3       print x, y
    elif (x>0):
5       bR(x-1,y+2);
        bR(x-1,y*2);
7       bR(x-1,y+1);
```

Voorbeeld

```
  def bR(x,y):
2   if bc0(x,y):
        bcc0(x,y)
4   elif rc0(x,y):
        bR(A[0][0](x,y),A[0][1](x,y))
6       bR(A[1][0](x,y),A[1][1](x,y))
        bR(A[2][0](x,y),A[2][1](x,y))
```

Algemenere notatie

Figuur 0.5 Voorbeeld van een functieaanroepboom



- Geen enkele functieaanroep heeft neveneffecten (behalve het print statement wanneer de recursie stopt);

- Geen enkele functieaanroep retourneert een waarde als resultaat.

Deze beperkingen kunnen echter één voor één weggewerkt worden totdat uiteindelijk een brede waaier van recursieve algoritmes kunnen getransformeerd worden van recursie naar iteratie.

De eerste voorwaarde kan worden afgezwakt tot: de maximale diepte van de recursie vlak voor het algoritme uitgevoerd wordt is gekend (het programma mag al aan het lopen zijn). In dat geval kan op mindere diepte dan verwacht al in een blad terechtgekomen worden. Springen naar het eerstvolgende blad komt dan neer op het verhogen van de bladteller met $b^{D_{max}-1-d}$ waarin b het aantal kinderen van een knoop is, d de diepte in de functieaanroepboom en D_{max} de maximale diepte van de recursie.

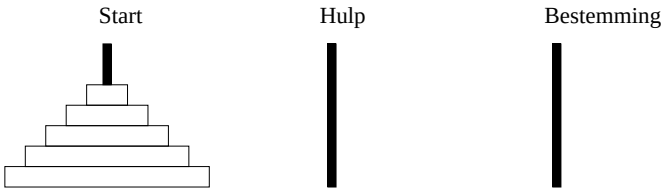
De tweede voorwaarde kan volledig worden weggewerkt net als de derde voorwaarde. Voor het wegwerken van de derde voorwaarde moet rekening gehouden worden met een complicatie. Als de recursie kan onder verschillende voorwaarden verdergezet worden met een verschillend aantal recursieve aanroepen. Hierdoor krijgen verschillende knopen die geen blad zijn in de boom mogelijks een verschillend aantal kinderen. Hierdoor valt het systeem van de voorstelling van het padnummer in een bepaalde getalbasis in het water. Deze complicatie wordt echter weggewerkt door het invoegen van recursieve aanroepen die geen effect hebben, zodat elke knoop weer hetzelfde aantal kinderen krijgt (of nul).

De vierde voorwaarde wordt weggewerkt door ervan uit te gaan dat het bewaren van de juiste volgorde van de verschillende functieaanroepen volstaat om in de aanwezigheid van neveneffect de juiste resultaten te blijven produceren. Een techniek gebaseerd op het onthouden van reeds berekende resultaten wordt voorgesteld om te vermijden dat functies in de iteratieve versie vaker worden uitgevoerd dan in de recursieve versie van het algoritme. Deze resultaten worden in een 'array' gestopt, en niet op een 'stapel' datastructuur gezet.

In de afwezigheid van neveneffecten kan het aantal functies waarvan het resultaat bewaard wordt in die array afgewogen worden ten opzichte van het aantal functieaanroepen dat moet gebeuren. Deze techniek wordt beschreven als partiële memoïzatie. De tijdsduur waarover die resultaten bewaard blijven kan afgewogen worden ten opzichte van de hoeveel parallellisme die gewenst is. Deze techniek wordt terplaatse-optimalizering genoemd, omdat resultaten overschreven worden wanneer ze niet meer nodig zijn.

De voorwaarde die stelt dat de functieaanroepen geen waarden teruggeven kan opgelost worden door die waarden in een 'array' bij te houden. Opnieuw is in die array terplaatse-optimalizering mogelijk.

Figuur 0.6 Torens van Hanoi



Uitdaging: Verhuis alle schijven van de Startpaal naar de Bestemmingspaal. Leg nooit een grotere schijf op een kleinere. Je kan slechts schijven verhuizen die bovenaan liggen. Schijven moeten altijd op een van de drie palen gelegd worden. Doe dit in zo weinig mogelijk stappen.

Algorithm 1. *Torens van Hanoi, recursieve oplossing.*

```
def losop(N, Start, Hulp, Bestemming):
 if N == 0:
  pass
 else:
  losop(N-1, Start, Bestemming, Hulp)
  print "leg van",Start,"op",Bestemming
  losop(N-1, Hulp, Start, Bestemming)
```

De methode waarin de voorstelling van een bladnummer in een bepaalde getalbasis gebruikt wordt als manier om het hele pad van wortel tot blad te reconstrueren wordt wiskundig afgeleid onder de vorm van een propagatievergelijking. Directe evaluatie van die vergelijking blijkt niet zo efficiënt te zijn en daarom wordt een padevaluatievergelijking afgeleid. Deze vergelijkingen vormen de basis voor het symbolisch simplifiëren van de iteratieve versies van de recursieve algoritmes. Waartoe deze simplificaties kunnen leiden wordt samengevat in twee voorbeelden.

0.4.2.4 Voorbeeld: Torens van Hanoi

Torens van Hanoi is de naam van een bekend raadsel dat uitgelegd wordt op figuur 0.6.

Het probleem wordt opgelost door te stellen dat het probleem met N schijven opgelost kan worden door eerst de bovenste N-1 schijven naar de middelste paal te brengen, daarna de grootste schijf op de laatste paal te leggen, en vervolgens de N-1 schijven op de middelste paal naar de bestemmingspaal over te brengen. Dit wordt geïllustreerd in de code in algoritme 1.

Toepassing van de methodes uit de thesis leidt tot de resultaten getoond in algoritme 2.

Algorithm 2. *Berekenen van de j-de zet in een N-schijven Torens van Hanoi probleem*

Om de j-de zet ($j = 1 \ldots 2^N - 1$) van een N-schijven Torens van Hanoi probleem te berekenen: (*elke stap wordt geïllustreerd op de 113de zet van het 9-schijven Hanoi problem*).

1. Schrijf de binaire representatie van j achterstevoren (van LSB tot MSB). Vul achteraan aan met nulletjes tot je N bits gebruikt hebt.
   Toegepast: Binaire representatie van 113 *achterstevoren met* 9 *bits is* 100011100.

2. Neem nu enkel de bits die na de eerste 1 bit komen.
   Toegepast: de bits na de eerste 1 *bit zijn* 00011100.

3. (Een kortere manier om de twee vorige stappen in 1 keer te berekenen:
   laat de eerste nul vallen van de achterstevoren geschreven N-bit representatie van $j\&(j-1)$
   Toegepast: 113&112 = 001110001&001110000 = 001110000*, schrijf achterstevoren en laat meest linkse nul vallen:* 00011100*)*
   Het meest linkse getal zal altijd nul zijn omdat altijd ofwel j ofwel j-1 even is.

4. Pas zo vaak als mogelijk de volgende regeltjes toe:

   (a) bij twee opeenvolgende 0 or 1 bits, verwijder beide (dus 00 →< empty > en 11 →< empty >)

   (b) telkens je twee opeenvolgende 01 or 10 bit-combinaties hebt, vervang je ze door 1 achterstevoren geschreven combinatie (dus 0101 → 10, 1010 → 01)
   Herhaaldelijk toepassen van (a) en (b) op de 113*de zet in een* 9*-schijven probleem resulteert in* 01.

5. Wat overblijft is een van de volgende bitcombinaties:

combinatie	actie
0	Start→Hulp
1	Hulp→Bestemming
01	Hulp→Start
10	Bestemming→Hulp
101 or 010	Bestemming→Start
<empty>	Start→Bestemming

6. Deze reeks van bitsubstituties voldoet aan een paar merkwaardige eigenschappen:

   - Met behulp van modulo rekenen kan eenvoudig bewezen worden dat n mod 3 met $n \in \mathbb{N}$ geschreven in binaire representatie kan berekend worden als het optellen van de waarden van bits 0,2,... en daarvan aftrekken de waarde van bits 1,3,5,... (alle berekeningen modulo 2).
   - Het is bijgevolg ook eenvoudig te bewijzen dat na de bovenstaande bitsubstituties de waarde van j mod 3 niet veranderd is.
   - Door te kijken in de tabel wordt duidelijk dat de startpaal voor een gegeven zet j bijgevolg eenduidig bepaald is door het resultaat van $(j\&(j-1))$ mod 3.
   - Hoewel het hier niet bewezen wordt bestaat er ook een formule voor het bepalen van de bestemmingspaal voor de j-de zet: $((j|(j-1)) + 1)$ mod 3.

Figuur 0.7 Recursieve MPEG4 textuurdecodeerapplicatie.

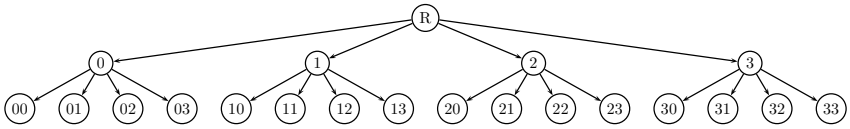
(recursieve code)

```
1
  def RecDecode(x,y,n):
3  if n == 0:
    Decode_pixel_EQ(x,y)
5   InvQuantPixel(x,y)
   else:
7   RecDecode(x,y,n-1)
    if n-1==4: Check_eos()
9   RecDecode(x+pow(2,n-1),y,n-1)
    if n-1==4: Check_eos()
11  RecDecode(x,y+pow(2,n-1),n-1)
    if n-1==4: Check_eos()
13  RecDecode(x+pow(2,n-1),\
              y+pow(2,n-1),n-1)
15  if n-1==4: Check_eos()
```

(iteratieve code)

```
2 def ItDecode(x,y,n):
  for j in range(pow(4,n)):
4  xbc = x
   ybc = y
6  mask = 1
   f = j
8  for i in range(n):
    xbc = xbc+(f & mask)
10  f = f >> 1
    ybc = ybc+(f & mask)
12  mask = mask << 1
   Decode_pixel_EQ(xbc,ybc)
14 InvQuantPixel(xbc,ybc)
   if n > 4 and (j+1) % 256 == 0:
16  Check_eos()
```

b=4, maxdepth=n



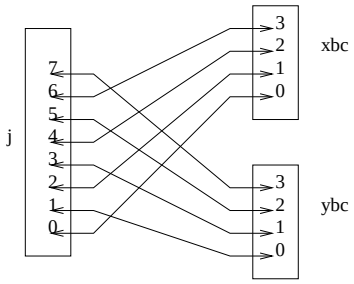
0.4.2.5 Voorbeeld: MPEG4 visual texture coding

De technieken uit deze thesis worden ook toegepast op de binnenste lus van een MPEG4 visuele textuurdecodeerapplicatie. Deze wordt voorgesteld in figuur 0.7.

De binnenste `for i in range(n):` lus van de iteratieve versie van de decoder zoals die wordt afgeleid met de technieken uit de thesis heeft een interessante interpretatie, die zeer efficiënt in hardware zou kunnen geïmplementeerd worden. Deze wordt voorgesteld in figuur 0.8. De opeenvolgende x-waarden en y-waarden die nodig zijn om de bewerking uit de voeren die gebeurt wanneer de recursie stopt zijn eenvoudig uit de binaire representatie van het bladnummer te bepalen.

Wanneer het iteratieve algoritme vergeleken wordt met het recursieve algoritme qua uitvoeringstijd (zonder de hardware-acceleratie die hiervoor beschreven werd), blijkt dat beide algoritmen ongeveer gelijkwaardig zijn. De iteratieve versie is iets beter voor kleine waarden van n. Omdat bij toepassing van dit algoritme op textuurinformatie niet vaak n ¿ 5 voorkomt — toch niet in de voorbeeldtexturen die we konden bekijken — kunnen we dus besluiten dat

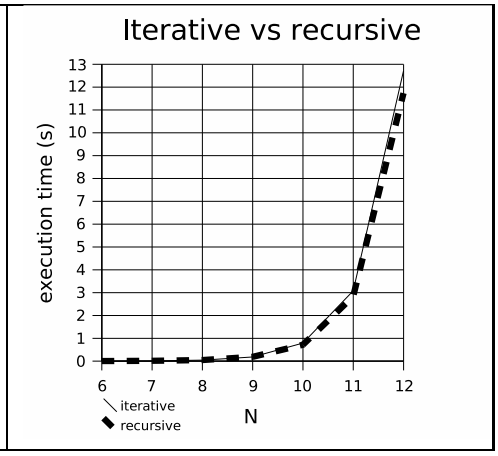
Figuur 0.8 Hardwareimplementatie van de argumenttransformaties.



calculating xbc and ybc from j
example where j is represented using 8 bits

Tabel 0.2: Tijdsmeting VTC decodeerapplicatie

n	iteratief (s)	recursief (s)
6	0.004	0.005
7	0.013	0.014
8	0.046	0.047
9	0.184	0.189
10	0.794	0.718
11	3.084	2.833
12	12.720	11.736



zelfs in software de iteratieve versie beter presteert dan de recursieve versie van de MPEG4 textuurdecodeerapplicatie. Een en ander wordt samengevat in tabel 0.2.

0.5 Conclusies en toekomstig werk

In deze thesis werden verschillende aspecten van TCM nader bekeken. Na een introductie rond TCM in het algemeen, handelde een eerste stukje handelde het MTG* model dat gebruikt wordt in elke stap van de TCM flow. Waar het model dat in TCM tot voor kort uit een een collectie informele blokdiagramma's bestond is het nu gedefinieerd als een extensie van het reeds bestaande MTG

model. Modelleren van een applicatie in MTG* dient hoofdzakelijk manueel te gebeuren — hoewel reeds een initiële tool-ondersteuning bestaat. Volledige formalisatie van MTG* was niet de bedoeling van dit eindwerk en wordt gelaten als toekomstige uitbreiding.

Het hoofdonderwerp van dit eindwerk betreft de broncodetransformaties. In dit eindwerk werden er twee besproken. De eerste transformatie verwijdert synchronizatie die parallellisme verhindert. Die synchronizatie is aanwezig onder de vorm van semaforen. Code die beschermd wordt door semaforen kan in sommige omstandigheden gesequentializeerd worden, waardoor de semafoor eigenlijk niet meer nodig is. Door de verwijdering van de synchronizatie is ook een deel overhead verdwenen. Dit wordt aangetoond met een JAVA implementatie van een MPEG2 video decoder waarin het aantal benodigde byte code instructies in een functie die de decodering van een bepaalde videostroom na de transformatie gedaald is van 160000 to 81000. De transformatie levert globaal gezien een winst op van slechts een kleine 4%. De reden hiervoor is dat de synchronisatie de applicatie niet domineert. De verwijdering van de synchronisatie resulteert wel in nieuwe vrijheid om bijvoorbeeld software pipelining technieken toe te passen. De toepassing van die software pipelining werd uiteindelijk niet voltooid wegens tijdsgebrek en wordt dus als toekomstige uitbreiding beschouwd.

De tweede transformatie zet recursie om naar iteratie. Hierbij werd eerst een raamwerk gemaakt waarin recursie kan geanalyseerd worden, en daarna werd het raamwerk toegepast om zeer algemene recursieve algoritmes om te zetten naar iteratieve vorm. Wederzijdse recursie wordt in dit deel niet behandeld en dit dient dus als toekomstig werk beschouwd te worden. De omzetting van recursie naar iteratie biedt ook een aantal vrijheidsgraden met betrekking tot hoeveel geheugen zal gebruikt worden, versus hoeveel berekeningen zullen gedaan worden, versus hoe parallelliseerbaar het resulterende iteratieve algoritme nog is. Ook het effectief aanwenden van die vrijheidsgraden vormt het onderwerp van toekomstig werk. De verschillende technieken worden omstandig gedemonstreerd met codefragmenten.

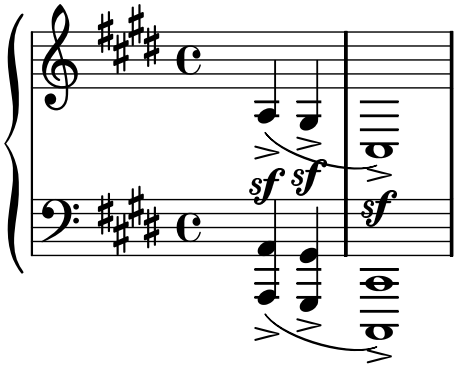
Het raamwerk biedt de mogelijkheid om een symbolische simplificaties van de resulterende iteratieve implementaties door te voeren waardoor de iteratieve versies van de recursieve algoritmes soms sterk vereenvoudigd kunnen worden. Zo wordt bijvoorbeeld een gesloten formule gevonden om de vertrekpaal in het "Torens van Hanoi" probleem voor een willekeurig zetnummer j te bepalen: $(j\&(j-1)) \bmod 3$. Verder wordt een algoritme om zowel de vertrekpaal als de bestemmingspaal te bepalen uit de binaire voorstelling van het zetnummer gevonden. Door toepassing van het recursieraamwerk op een MPEG4 visual texture coding applicatie resultaart na omzetting van recursie naar iteratie een eenvoudige hardware-interpretatie van de bewerkingen in de binnenste lus wat aanleiding kan geven tot een zeer efficiënte implementatie. Kwantificatie van die winst is ook toekomstig werk.

Part I

Introduction

Chapter 1

Thesis contributions



Sergei Rachmaninov, Prelude op.3,no.2

1.1 Context and motivation

1.1.1 Introduction

The merger of computers, consumer and communication disciplines gives rise to very fast growing markets for personal communication, multimedia and broadband networks. Technology advances lead to platforms with enormous processing capacity that are not complemented by the required increase in system design productivity. One of the problem areas is the very dynamic concurrent behavior of many of these new applications. They are fully specified in software oriented languages (like Java, UML, SDL, C++) and need to be executed in

real-time cost/energy-sensitive way on the heterogeneous system-on-chip (SoC) platforms. A systematic way of mapping this SW/HW specification onto an embedded multi-processor platform is required.

Fully design-time based solutions as proposed earlier in the compiler and system synthesis communities cannot handle the problem except by going to costly worst-case assumptions. Also run-time solutions as present in nowadays (real-time) operating systems are too inefficient in terms of cost (especially energy consumption) and are also not fully adapted to meeting hard real-time constraints. The dynamic nature of applications is especially emerging because of the quality of service (QoS) aspects of multimedia and networking applications. Prominent examples of this can be found in the recent MPEG4/JPEG2000 standards and especially the new MPEG21 standard. In order to deal with these dynamic issues where tasks and complex data types are created and deleted at run-time based on non-deterministic events, a novel system design paradigm is required.

In this chapter a task concurrency management (TCM) problem formulation is proposed. TCM has been developed under guidance of professor Francky Catthoor and in cooperation with a team of PhD students including but not limited to Pol Marchal, Peng Yang, Chun Wong, Ma Zhe and myself. It is briefly explained how the use of Pareto curve based exploration enables a solution to the problems mentioned before. The promising results that can be obtained with such a methodology have been illustrated with an MPEG21 based demonstrator mapped on a multi-Vdd multi-processor simulation platform with a hierarchical shared memory organization [226].

1.1.2 Platform-based design

The future of embedded multimedia applications lies in low-power heterogeneous multi-processor platforms [56]. In the near future the silicon market will be driven by low-cost, portable consumer devices which integrate multi-media and wireless technology. Most of these applications will be implemented with compact and portable devices, putting stringent constraints on the degree of integration (i.e. chip area) and on their energy consumption (0.1-2W). The applications will require a computational performance of the order of 1-40GOPs. Current PCs offer this performance. However, their energy consumption is too high (10-100W). We must keep this performance and reduce the energy consumption at least two to three orders of magnitude. Embedded systems are also subject to stringent real-time constraints complicating their implementation considerably. Finally, the challenge to embed these applications on portable devices is increased even further because of user interaction: at any moment the user will be able to trigger new services, change the configuration of the running services, stop existing services. Hence, the execution behavior changes dynamically at run-time.

Therefore, the future embedded system should have sufficient computing and memory resources, consume an extremely low amount of energy, and be flexible enough to cope with the dynamic behavior of future multi-media applications. In spite of modern voltage scaling techniques, existing single processor systems (e.g. found in PDAs) cannot deliver enough computational performance at a sufficiently low energy budget. On the other hand, full custom hardware solutions can achieve a good performance/energy consumption ratio but are not flexible enough to cope with the dynamic behavior and have a relatively long time-to-market delay. Also the technology scaling trend with the explosion of masks costs and of the physical design (especially due to timing closure and test) implies that custom ASICs are only feasible for high volume designs. A combination of the best of both worlds is required for the majority of the new applications [47]. This explains the growing interest in platform-based design [180].

The benefits of heterogeneous platforms can only be fully exploited when applications are efficiently mapped on them. Unfortunately current design technologies fall behind these advances in computer architecture and processing technology. When looking at contemporary design practices for systems implemented with these heterogeneous multi-processor platforms one can only conclude that these systems are nowadays designed in a very ad hoc manner. A systematic methodology and its corresponding tool set are definitely needed.

In section 1.2 IMEC's systematic design methodology to embed multi-media applications on platforms is briefly summarized. With this design methodology especially the management of dynamic and concurrent tasks and their data on heterogeneous multi-processor platforms is targeted. Given an application which has its own specific control and data structure and a parameterizable template for the platform architecture [159] this requires finding a way to decide the platform architecture instance and how to map the application efficiently onto such an architecture.

We only focus on the concurrency management issues in this overview even though the data and communication management part [143, 38] and the instruction management parts [204] are as crucial. Platform-based design also encompasses a physical implementation problem that can be called platform integration but that is not addressed here either.

Platforms typically contain one or more (re-configurable) programmable components, either general-purpose or DSP processor cores or ASIPs, the analog front end, on-chip memory, I/O and other ASICs are all integrated into the same chip (System-on-Chip) or in the same package (System-in-Package). Furthermore, platforms typically contain also some software blocks (API, Operating System and some other middleware solutions), a methodology and a tool set to support rapid architectural exploration. Examples of such platforms are TI's OMAP [195], the Philips Nexperia [169], the ARM PrimeXsys [7] and

the Xilinx Virtex-II Pro [220]. Platforms obtain a high performance at a low energy cost through the exploitation of different types of parallelism [1, 115]. In the context of this overview focus will be on the task-level concurrency but also the instruction and data-level parallelism should be exploited in the mapping environment. In this way the energy consumption can be kept lower by exploiting Vdd and frequency scaling in the processing cores. It is assumed that only a finite discrete set of Vdds can be supported with a not too large range because future process technologies demand that. Large memories even do not allow any Vdd scaling but in that case the energy consumption can be controlled by an efficient mapping on a distributed memory architecture [37].

1.2 An overview of TCM

This section presents an overview of the task concurrency management methodology (TCM). TCM is in the process of being defined and refined by a team of PhD students (including but not limited to Peng Yang, Chun Wong, Pol Marchal, Zhe Ma and myself) under the technical supervision of professor Francky Catthoor at IMEC.

Throughout the text many design decisions are motivated by referring to certain steps of the TCM methodology. To understand such references it is best to keep this chapter in mind while reading the rest of the text. Work not directly concerning the modeling and transformations (e.g. as related to design-time or run-time scheduling, or task level data transfer and storage exploration (T-DTSE)) is not my own contribution, but the contribution of other members of the PhD student team.

1.2.1 Overview

Task concurrency management (TCM) addresses the dynamic and concurrent task scheduling problem on a multi-processor platform for real-time embedded systems where energy consumption is a major concern. It is based on a two-phase scheduling method which is a part of IMEC's overall system design methodology. It provides the required flexibility at a low run-time overhead. The TCM methodology [196, 225, 228] being itself one stage in a complete system design flow comprises the following sub-stages:

1. First a *gray box model* (see chapter 3) of the application is extracted to explicitly show the potential parallelism and dynamic behavior of the application. The gray box model breaks up an application in thread frames and thread nodes.

   Thread frames are maximally sized pieces of code which are suitable to

design-time analysis. Inside a thread frame no dynamic task creation, event handling or synchronization occurs. (Scheduler) thread nodes are groups of instructions that will be considered as a black box during the design-time scheduling. Scheduler thread nodes need to have a fairly constant execution time. If a large range of execution times for a particular thread node is observed the data dependent constructs causing this range must be visualized outside the thread node.

The thread node boundaries can change during the TCM flow, e.g. as the result of clustering and splitting transformations. So called primitive thread nodes are defined in such a way that they never need to be changed during task level exploration.

Syntax and operational semantics of the MTG* model and the gray box model are presented in part II of this thesis, in chapter 3.

2. In the second step of TCM *concurrency improving transformations* are applied to the gray box model. Such transformations include both task level transformations and intra task transformations. Task level transformations can change the number of thread frames in the system, the granularity of thread frames or the synchronization between thread frames. Intra task transformations aim at creating more parallelism between thread nodes and at balancing the workload over different scheduler thread nodes.

   TCM transformations consider thread nodes to be black boxes. This means that the transformation must be defined in such a way that the implementation of a given thread node does not need to be known. Each TCM transformation must therefore specify a suitable definition (granularity) of thread nodes to make this possible and produce correct results.

   After applying transformations the initial gray box model can be refined and scheduler thread node scenarios are identified. To avoid worst-case implementations of thread frames in an application the data dependent constructs in between the thread nodes are characterized using design-time analyzable scenarios. These scenarios allow to prepare at design-time different mappings to the platform of the thread nodes inside the thread frames. At run-time a suitable mapping can be chosen depending on the estimated or known values of the data dependent constructs.

   Two transformations will be developed in part III of this thesis, in chapter 5 about semaphore removal and 6 about recursion removal. A classification of TCM transformations appears in appendix B

3. The third step of TCM consists of a two-phase scheduling approach.

   (a) First the *design-time scheduling step* is applied to each scenario of each identified thread frame in the system. Different from traditional design-time scheduling it does not generate a single solution but a

Pareto curve consisting of different Pareto optimal scheduling cost-performance trade-off points.

Each point on such a Pareto curve represents a particular mapping of thread nodes inside one thread frame to the target platform together with the chosen voltages if the platform can be voltage-scaled. The coordinates of the Pareto point show the minimum obtainable cost in the presence of constraints (also see section 4.3 on page 75).

A TCM design-time scheduler was developed in [218].

(b) Finally an application-specific *run-time scheduler* is integrated in the real-time operating system of the platform. At run-time the TCM run-time scheduler dynamically selects one of the Pareto optimal trade-off points for each running thread frame and constructs a globally energy efficient solution.

More information about a TCM run-time scheduler can be found in [222].

The TCM flow is summarized in figure 1.1 on the facing page.

1.3 Contributions

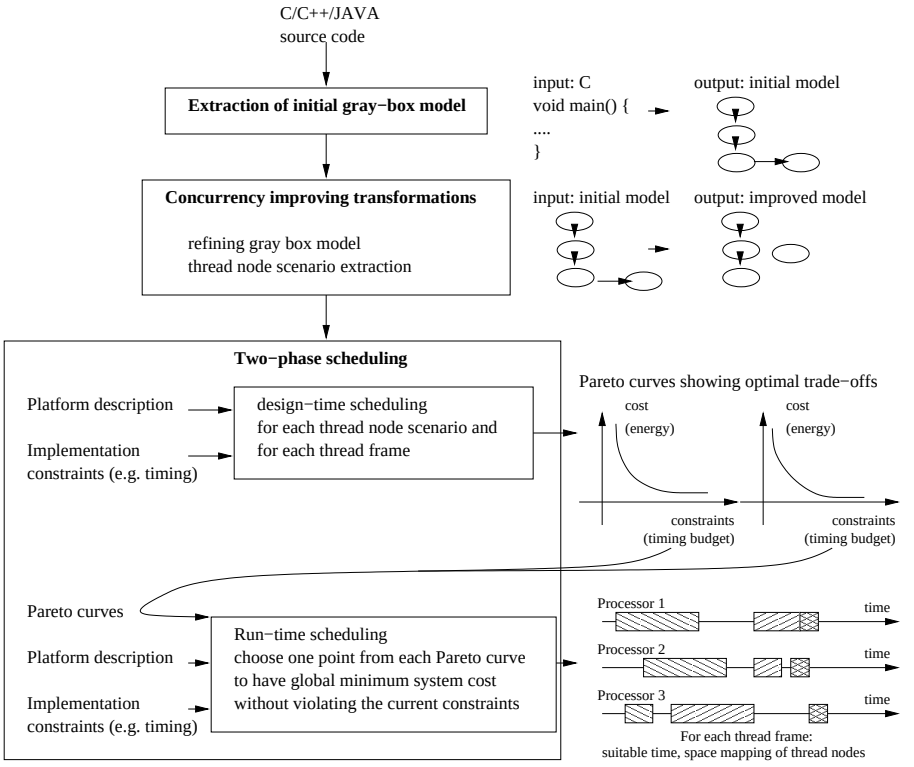
In this section my contributions to TCM are summarized.

1.3.1 Application representation

Part I of this thesis is about application representation. TCM was developed to optimize large applications with soft timing constraints and dynamic behavior for energy consumption. Energy is reduced by postponing certain design decisions until the application is being executed. A number of different options for mapping the software on a given hardware platform is prepared at design-time. Different options correspond to different combinations of execution time and energy. If the workload in the system becomes high due to dynamically created concurrent tasks competing for resources, faster but more energy-hungry mappings of the software to the platform can be selected at run-time. If the workload drops again due to completion or deletion of tasks, slower but less energy-consuming mappings can be selected. Typically, choosing different options will correspond to activating more processors on the platform or voltage scaling of the processors on the platform.

Because making decisions at run-time is a form of overhead the amount and complexity of decisions to be taken at run-time should be minimized. For this reason the existing MTG model developed in IMEC was adapted and developed into MTG* by combining ideas from the DF* model and the MTG model.

Figure 1.1 TCM flow



MTG* can represent any application that can be represented in imperative high-level languages like C, C++ or JAVA. The main difference between MTG* and plain C/C++ is the concept of an MTG* thread frame (TF). Thread frames are maximally sized pieces of code in which no synchronization, task creation or event handling occurs. These pieces of code can still contain data dependent decisions and non-manifest loops. Because no synchronization, task creation or event handling occurs thread frames can be characterized with minimum and maximum (worst-case) execution time, and minimum and maximum energy consumption. Execution time and energy figures vary due to data dependent values influencing the instruction path that is followed through a thread node.

Thread frames in MTG* contain control data flow graphs (CDFGs) describing their functionality. CDFGs are similar to the data structures built inside a compiler.

Because thread frames usually have a coarse granularity and because the CDFGs inside often reveal too many details which are less interesting while doing task-level exploration an extension has been defined on top of MTG*. This extension is called the gray box model. A gray box model clusters part of the CDFGs inside a thread frame into so-called thread nodes. Clustering functionality in a thread node can be thought of as encapsulating functionality in a function. The result is a more abstract version of the code. Thread nodes are a way to selectively show or hide low-level details of the code. Data dependent decisions and non-manifest loops e.g. will not be hidden inside a thread node if they cause a large difference in execution time for different input data. The result is that thread nodes do not exhibit a large variety in execution times or energy consumption. The data dependent values left visible outside the thread nodes can be used as parameters in a parametrized performance and energy model of a thread frame. By means of profiling, data dependent values will be sorted into ranges, and for each range different mappings to the platform can be prepared. A platform mapping corresponding to a range of parameter values is called a scenario.

My contribution to MTG* is to bring together all the views of people using informal block-diagrams at IMEC and attach some meaning to the constructs that were found to be necessary in practice. One of the requirements was to stay as compatible as possible with the original MTG model. A second requirement was to be able to clearly show synchronization, dynamic task creation and deletion and event handling. The transformations described in chapters 5 and 6 show how the MTG* model serves as an aid in defining and performing TCM concurrency improving source-to-source transformations.

Recently it has become clear that some definitions and concepts benefit from relaxing the stringent requirements on them. Current discussions and issues are described in appendix C.

1.3.2 Concurrency improving transformations

Part II of this thesis is about transformations for task concurrency management. Such transformations are often referred to as concurrency improving transformations although this is a misnomer. Instead, many TCM transformations aim at cleaning up the C/C++ implementation or at creating freedom for later concurrency improving transformations. With respect to these concurrency improving transformations I make two contributions in this part of the thesis

1. The semaphore removal transformation discussed in chapter 5 is a task level transformation. This means that the transformation affects the task level model of the application. In other words it will change something to the synchronization, task creation or deletion or the event handling of an application. In the case of chapter 5 semaphores are removed from code by selective sequentialization. The removal of synchronization will result the execution of less byte code instructions, and will therefore reduce the energy consumption. It will be shown by means of a real-life JAVA MPEG video decoder example how the transformation creates freedom for a further parallelization step.
2. Part III, chapter 6 develops a new technique for removing non-linear non-mutual recursion from applications. By means of real-life examples the techniques are demonstrated to make explicit the complete control and data flow of non-linear recursive functions. Moreover it is shown how the technique enables systematic trade off of different costs. The amount of argument transformation calculations that need to be done can be traded off systematically for the amount of memory that will be used. Independent of the first trade off, the amount of memory being used can be traded off for the required amount of parallelism in the algorithm implementation. Application of the method on the towers of Hanoi will additionally result in an algorithm for determining the N-th move from the binary representation of N, and in a closed formula for determining the first pole in the N-th move. Application of the method on a VTC example results in an efficient hardware interpretation of the innermost loop.

The appendices contribute ideas with respect to

1. transformation identification and instantiation (appendix B),
2. multi-dimensional Pareto surfaces and their possible use in transformation impact estimation (appendix B, section B.2 on page 163 and further),
3. as well as an overview of current issues and future work in TCM (appendix C).

1.4 Thesis structure

In this chapter (chapter 1) some context and motivation and a summary of my contributions to TCM were presented. The rest of the thesis is structured as follows.

The thesis is divided in four parts. Part I serves as an introductory part. Part II focuses on aspects of application modeling. Part III looks at source-to-source transformations and their effects in relation to the Task Concurrency Management methodology. Part IV finally presents some conclusions. A number of appendices contain detailed technical information that did not fit in the main text.

Part I introduces the application domain and presents an overview of TCM. In addition the main ideas behind task concurrency management are summarized. Such an overview aids in understanding the relevance of some of the concepts and ideas presented in later chapters. The ideas presented there are somewhat simplified and some of the issues that were deliberately left out in section 1.2 are discussed in appendix C.

Part II starts with a chapter 2 which presents an overview of approaches to modeling applications in the embedded systems community. Chapter 2 thus serves as an introduction to the description of the models used in TCM as given in chapter 3.

Part III, chapter 4 looks at related work in the context of program transformations. The main focus is on work related to semaphore removal and recursion removal. Semaphore removal by sequentialization and recursion removal for non-linear recursion in the presence of side-effects are the subject of chapters 5 and 6 respectively.

Chapter 7 in part IV finally formulates some conclusions.

1.5 List of publications

1.5.1 Related to MTG* and application modeling

In [HDCv02] an older version of the MTG* model is described; it has been used as a reference text for the first two editions of the TCM course organized at IMEC, especially the sessions about application modeling. This same text has been used as starting point for the later editions of the TCM course. Previously, posters [Him01c, Him02b] and project deliverables [Him01a, Him01b, Him02a] already described parts of MTG*. A conference paper about MTG* modeling has been published [HCDv02], as well as a book chapter [HCDv03]. To

help IMEC with an alpha implementation of automated MTG* extraction and scheduling from C++ code, an additional document has been written [Him02c], later subsumed and superseded by [ZYH+03].

Bibliography

[HCDv02] Stefaan Himpe, Francky Catthoor, Geert Deconinck, and Jef van Meerbergen. MTG* and gray box: modeling dynamic multimedia applications with concurrency and non-determinism. In *proceedings of the forum on specification and design languages (FDL02) on CDROM*, Marseille, France, September 2002.

[HCDv03] Stefaan Himpe, Francky Catthoor, Geert Deconinck, and Jef van Meerbergen. *System Specification and Design Languages: Best of FDL02*, chapter MTG* and gray box: modeling dynamic multimedia applications with concurrency and non-determinism. Kluwer Academic Publishers, 2003.

[HDCv02] Stefaan Himpe, Geert Deconinck, Francky Catthoor, and Jef van Meerbergen. MTG* and gray box modeling: a modern bible for TCM methodologists. Technical report, Katholieke Universiteit Leuven, 2002.

[Him01a] Stefaan Himpe. Modeling systems using MTG* and gray box and classification of TCM transformations, July 2001. project deliverable, Natlab-K.U.Leuven overeenkomst RWR-061-JR-01001-jr, project: Dynamic scheduling for embedded multiprocessor architectures.

[Him01b] Stefaan Himpe. Modeling systems using MTG* and gray box model, classification of TCM transformations, July 2001. project deliverable, IWT-ITA-II GBO330 Nr. 980302, SuperVisie: Beeldverwerking voor geïntegreerde en immersieve visualisatie (Final Deliverable).

[Him01c] Stefaan Himpe. MTG* and gray box, 2001. Extended abstract and poster PA3CT symposium 2001.

[Him02a] Stefaan Himpe. Gray box model extraction and TCM transformations on JPEG and MPEG21 demonstrators, July 2002. project deliverable, Natlab-K.U.Leuven overeenkomst RWR-061-JR-01001-jr, project: Dynamic scheduling for embedded multiprocessor architectures.

[Him02b] Stefaan Himpe. MTG* and gray box, modeling of dynamic multimedia applications, 2002. Extended abstract and poster at KUL PhD symposium 2002.

[Him02c] Stefaan Himpe. XML format to store MTG* model. Technical report, Katholieke Universiteit Leuven, November 2002.

[ZYH+03] Ma Zhe, Peng Yang, Stefaan Himpe, Tanja Van Achteren, Patrick David, Chantal Ykman-Couvreur, Francky Catthoor, and Johan Vounckx. Requirements for C annotations, tipsy usage, and XML specification of the gray box model and scenarios (version 0.2). Technical report, IMEC internal report, December 2003.

1.5.2 Related to program transformations

A start was made with TCM transformation identification and classification in report [HDC01]. In the project deliverable [Him02] a number of TCM transformations are identified and applied on a JPEG decoder and MPEG21 QoS demonstrator. The basic idea behind the recursion removal method was described in [Him03] and applied to the VTC decoder in [HDC03]. The recursion removal technique was demonstrated to increase optimization freedom leading to decreased overall system cost in [ZWH+03]. In [HDC04] a preliminary version of chapter 6 about recursion removal is described. It generalizes the approach used in [HDC03]. A journal article [HCD05] reporting on the recursion removal techniques from chapter 6 was prepared and submitted for review.

In [Him04] the ideas of chapter 5 about semaphore removal are summarized.

Bibliography

[HCD05] Stefaan Himpe, Francky Catthoor, and Geert Deconinck. Removing recursion from imperative programs. *Software - Practice and Experience (SPE)*, 2005. Submitted for review.

[HDC01] Stefaan Himpe, Geert Deconinck, and Francky Catthoor. TCM transformation classification. Technical report, Katholieke Universiteit Leuven, 2001.

[HDC03] Stefaan Himpe, Geert Deconinck, and Francky Catthoor. Control flow analysis for recursion removal. In Andreas Krall, editor, *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003)*, volume Lecture Notes in Computer Science 2826, pages 101–116, Vienna, Austria, September 2003. Springer Verlag.

[HDC04] Stefaan Himpe, Geert Deconinck, and Francky Catthoor. Control flow analysis for recursion removal. Technical report, Katholieke Universiteit Leuven, 2004.

[Him02] Stefaan Himpe. Gray box model extraction and TCM transformations on JPEG and MPEG21 demonstrators, July 2002. project deliverable, Natlab-K.U.Leuven overeenkomst RWR-061-JR-01001-jr, project: Dynamic scheduling for embedded multiprocessor architectures.

[Him03] Stefaan Himpe. Control flow analysis for recursion removal, 2003. Extended abstract and poster at PA3CT symposium 2003.

[Him04] Stefaan Himpe. Semaphore transformations for task concurrency management. In *proceedings of PA3CT 2004*, Edegem, September 2004. Extended abstract and poster.

[ZWH+03] Ma Zhe, Chun Wong, Stefaan Himpe, Eric Delfosse, Francky Catthoor, and Geert Deconinck. Task concurrency analysis and exploration of visual texture coder on a heterogeneous platform. In *proceedings of the 2003 IEEE workshop on signal processing systems (SIPS03)*, 2003.

1.5.3 TCM methodology related

In [YMW+02] and [YMW+04] an overview of the ideas behind TCM is given. In report [ZYH+03] an overview is given of existing demonstrator applications for TCM.

Bibliography

[YMW+02] Peng Yang, Pol Marchal, Chun Wong, Stefaan Himpe, Francky Catthoor, Patrick David, Johan Vounckx, and Rudy Lauwereins. Managing dynamic concurrent tasks in embedded real-time multimedia systems. In *proceedings of the international symposium on system synthesis (ISSS02)*, pages 112–119, Kyoto, Japan, October 2002.

[YMW+04] Peng Yang, Pol Marchal, Chun Wong, Stefaan Himpe, Francky Catthoor, Patrick David, Johan Vounckx, and Rudy Lauwereins. *Multiprocessor Systems-on-Chips*, chapter Cost Efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. Morgan Kaufmann (Elsevier), 2004.

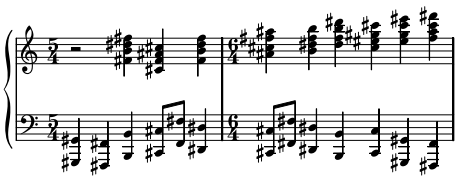
[ZYH+03] Ma Zhe, Peng Yang, Stefaan Himpe, Tanja Van Achteren, Patrick David, Chantal Ykman-Couvreur, Francky Catthoor, and Johan Vounckx. TCM demonstrators. Technical report, IMEC internal report, December 2003.

Part II

Application modeling

Chapter 2

Related work in application modeling



Modest Mussorgsky, Bilder einer Ausstellung (Promenade)

2.1 Introduction

In embedded systems it is important not only to optimize for lowest execution time but also to look at other costs like energy consumption or chip area. Models focusing on functionality only are not a suitable basis for optimization in an embedded system context. We will look at different aspects of embedded systems, present existing approaches and compare them to what our MTG* model has to offer in this area.

The aspects that are looked at are summarized from [136] and go further than the ones in the embedded software overview chapter there.

It is not the intention to claim completeness in describing all the modeling literature because that is simply too extensive today. Describing this requires

a full book on its own. Instead, we focus on the key aspects of the proposed MTG* model and discuss related literature.

2.2 Modeling computation

A large number of models have emerged from different scientific communities. Traditionally the computer science community puts emphasis on functionality, concurrency and interfaces. The hardware/software codesign community builds on the computer science models and adds aspects like timeliness, liveness, heterogeneity and reactivity.

One model of computation of historic importance is the Von Neumann non-biological kinematic self-replicating automaton [209]. He proposed a mathematical abstraction of a machine capable of making copies of itself. Von Neumann's work may have been inspired by one of his visiting PhD students, Alan Turing. Turing came up with a conceptually simple mathematical model for a machine [199, 200] and formulated the Church-Turing thesis, namely that every other practical computing model has either the equivalent or a subset of the capabilities of a Turing machine. While this thesis can not be proved it is generally accepted as being true. Why then is effort being put in developing other models of computation? At least three reasons exist.

- First, models of computation having the capabilities of a Turing machine are said to be Turing-complete. Some properties cannot be decided in general for Turing complete models, e.g. proving that execution will halt or will not deadlock. In mission-critical applications one would like to have some assurance about such properties.
- Second, implementing an algorithm on a Turing machine is not necessarily efficient. It may take thousands of simple instructions to model a real-life application.
- Third, the Turing machine does not clearly represent any of the properties that are important for embedded software as listed in [136].

Other models of computation include

- Combinational logic circuits modeled by Boolean logic
- Finite state machines (FSM) having a concept of memory or state. An FSM can be used to recognizes regular expressions.

  Finite state machines are often used in embedded design. They have a limited expressivity. In particular they cannot represent all partial

recursive functions[1]. The limited expressivity of FSMs allows a greater amount of formal analysis to be done than on Turing complete models of computation. The number of states needed in a pure FSM-based model rises exponentially when concurrency is modeled. FSMs can be relatively efficiently mapped to both software or hardware.

- Pushdown automata (PDA) are more expressive than FSMs in that they can recognize context-free languages. They are typically used in parsers.
- The random access machine (RAM). A RAM has a central processing unit (CPU) and a random-access memory with the property that each memory word can be accessed in one unit of time. Its CPU executes instructions, reading and writing data from and to the memory. Both the RAM and the Turing machine have equivalent expressiveness and can be taken as a fundamental model of computation [182].
- A Petri net [167, 166] is a graphical and mathematical modeling tool. It is a directed bipartite graph which has two types of nodes called places and transitions. Directed arcs connect places to transitions or transitions to places. Input arcs connect places with transitions while output arcs start at a transition and end at a place. Input and output arcs can have a weight associated to them. Other types of arcs exist, e.g. inhibitor arcs. Places can contain tokens. The current state of the modeled system (the marking) is given by the number of tokens in each place. Transitions model activities which can occur. A transition can fire under certain circumstances changing the state of the system (the marking of the Petri net). Transitions are only allowed to fire if they are enabled which means that all the preconditions for the activity must be fulfilled (enough tokens must be available in the input places). When a transition fires it removes tokens from its input places and adds some at all of its output places. The number of tokens removed/added depends on the weight of each arc. When multiple arcs depart from a place only one arc gets a token. The interactive firing of transitions in subsequent markings is called the token game.

  Petri nets are especially useful in a context where concurrency and non-determinism are present. They avoid the state space explosion which is typical when trying to model concurrency with an FSM. Petri Nets play in an important role in the MTG* model. See section 2.4 on concurrency.

In the MTG* model which is briefly described in chapter 3 two layers are distinguished.

- The upper layer is called the task level. Tasks are modeled using Petri nets annotated with dynamic task creation and deletion and task-level

[1] A partial recursive function is a function computed by a Turing machine that need not halt for all possible inputs.

data flow. The task level thus clearly shows the available task-level parallelism and task creation and deletion, event handling and synchronization.

- The lower layer is called the intra task level. It consists of a control data flow graph (CDFG). CDFGs are a type of data structures that are used inside compilers. The particular control data flow graph in use in MTG* is the decorated signal flow graph (DSFG). The latter will not be described in more detail here as a full description is provided in [130].

An MTG* model is typically extracted from C, C++ or JAVA code.

2.3 Modeling data flow and data communication

- In the context of digital signal processing systems a number of data flow models have been defined.
  - The synchronous data flow concurrency model (SDF) [135] consists of blocks representing functions connected by directed arcs. The arcs (usually implemented as FIFOs) carry tokens (data values) which cause the blocks to fire (execute). If a block fires it consumes a fixed number of tokens from the input arc and produces a fixed number of tokens on the output arc. The run-time behavior of an SDF system is completely known at compile-time and hence an optimal schedule can be calculated at compile-time. This means that applications which change at run-time (which are called dynamic applications in this thesis) cannot be handled in such a formalism except by doing run-time recompilation. Unfortunately, according to [136] full run-time recompilation is not efficient and incremental run-time compilation proves to be challenging.
  - Both SDF and homogeneous data flow (HDF) [206] are special cases of Kahn process networks [119] and have been further generalized to cyclo-static data flow(CSDF) [132], boolean data flow (BDF) [30], parametrized data flow (PDF) [21, 153], cyclo-dynamic data flow (CDDF) [215] and dynamic data flow (DDF) [112].

  The different forms of data flow trade off compile-time knowledge of the behavior for expressiveness. Data flow models cannot easily represent complex control flow. Control flow needs to be simulated by routing data values.

- Other more control-oriented models have been extended with constructs to model data communication.

  - Dual transition Petri nets (DTPN) [205] are an extension of Petri nets (see section 2.4 about models for concurrency) with data flow arcs and data transitions. The control arcs have conditional guards associated to them which depend on the values of data tokens. The guards thus form the connection between the control part and the data part of a model.
  - Timed Petri nets [116] have been combined with a data path and instruction dependence graphs to form a Petri net based Unified REpresentation (PURE) [189]. Hierarchy is not supported.
  - Finite State Machines (FSM) combined with data paths [67] are the traditional approach to designing digital hardware systems. The data paths consist of combinational logic and sequential logic, both of which can be described by boolean equations.
  - The FUNSTATE model [190] consists of a network of storage units and functional units and arcs that relate them to model the data intensive part of algorithms and an FSM to model the control intensive parts. Data is represented as valued tokens in the storage units. Activation of functional units in the network is controlled by the FSM.
  - The DF* model [52, 154] combines an FSM with a controller activating cyclo-static data flow blocks.

- Concurrent tasks or threads implemented in high-level programming languages frequently need to communicate data. Two basic mechanisms have been proposed.

  - In the shared memory communication model communication is implicit via loads and stores to and from a global address space. The physical location of data in shared memory is unspecified allowing for exploitation of memory hierarchies by means of caches. Synchronization in shared memory is completely separated from communication. To solve race conditions because of concurrent tasks sharing resources in a shared memory system locking shared memory (SM) areas with semaphores [60] can be used. Because semaphores have a low level of abstraction they have been used as basis for higher level abstractions like monitors [105]. The JAVA programming language [118] has built-in support for monitors by means of the 'synchronized' keyword. In C [123] or C++ [191] this built-in support does not exist. On a semaphore based operating system semaphore abstractions for such languages can be built using operating system calls.
  - Message passing is an explicit communication model. Communication consists of messages from processes or threads having some data to processes or threads needing that data. Arrival of a message constitutes an asynchronous event. Software must be invoked to handle

the event. Synchronization and communication are integrated. In message passing different variants exist. The mailbox model and the rendezvous model are given as examples.

* A mailbox is a place to buffer a certain number of messages. Messages are sent to and received from mailboxes, not processes or threads. A process sending to a full mailbox suspends until room for a new message becomes available.
* A different approach is taken in the rendezvous model: if a send operation is done before a receive operation the send process is blocked until the receive happens at which time the message can be copied directly from the sender to the receiver without any intermediate buffering.
* Rendezvous is easier to implement than a buffered message scheme but is less flexible since the sender and receiver are forced to run in lockstep [194].

A good comparison between shared memory and message passing and reasons to make a hybrid form can be found in [128].

The MTG* model briefly described in chapter 3 uses a shared memory paradigm. If needed, message passing can be implemented efficiently on top of a shared memory system. Shared memory can also be implemented on a message passing system but this is less efficient, especially in dynamically behaving systems or systems which potentially need to share data with fine granularity [127]. Data modeling in MTG* graphically looks similar to data in DTPN, but in contrast to DTPN data communication has not been assigned an operational semantics. Reading or writing data does not imply synchronization. If synchronization is present (e.g. a blocking read) it must be modeled explicitly with the appropriate Petri net constructs[2]. The reasons for not fully formalizing data semantics in MTG* are two-fold:

- In many often used programming languages like C or C++ synchronization is not implicitly present. It should be as easy as possible to extract an MTG* model from a C or C++ program.

- Exact requirements on data modeling were not yet clear while the work on MTG* took place. More formal modeling of data in MTG* had to be left as future work.

[2]Blocking reads hence are a reason to introduce a thread frame boundary). This requirement might be relaxed in the future as discussed in appendix C

2.4 Concurrency

Embedded software is expected to react to multiple user requests and sensor events at once while continuing to control actuators. Concurrency is present in all but the simplest of embedded systems.

- The engineering approach to concurrency is to use processes, threads and semaphores. According to [136] these techniques can be used to represent any algorithm but they have a low level of abstraction, comparable to assembly language. The computer scientific answer to this critique is to build abstractions on top of these primitives, e.g. FIFOs to be used as communication channels with built-in synchronization and monitors to guard shared data and avoid race conditions. In [136] it is argued that compositions of such mechanisms quickly become too hard to understand. In practice this probably need not be the case as long as one sticks to a set of well-understood design patterns [69, 133, 183].
- Mathematically inspired theories of concurrency often reduce concurrency to interleavings, abstracting away the influence of time. The success of λ-calculus to model function definition and function application has lead to development of other formal calculi[3] each tailored to the needs of a particular application domain. We mention the π-calculus (also known as process calculus, a successor to the calculus of communicating systems (CCS)) for modeling concurrent communicating systems [150]. Another formal calculus is the γ-calculus [186] which underlies concurrent constraint satisfaction systems. This calculus is the foundation for the MOZART OZ3 programming language which unifies the common programming paradigms in one programming language. The approach is documented in [203].
- Throughout [203] different types of concurrency are classified in order of increasing expressiveness and their possibilities and limitations are explained. The types of concurrency include the models of declarative concurrency, message-passing concurrency and shared-state concurrency.
  - Declarative concurrency is based on declarative programming where a variable can be bound to a value only once. This value cannot change. If the value of variable is needed but the variable is not yet bound execution blocks until the variable is bound. This way *any part* of a declarative program can be put in a separate thread without altering the end results of the program. Concurrency in this context has the advantage of enabling incremental computation: when new information arrives, execution can continue until it is blocked again

[3] It is rumored that at least one formal calculus for every letter in the Greek alphabet exists.

waiting for new information. According to [203] no observable non-determinism is present in the declarative concurrency model[4].

- Message-passing concurrency is declarative concurrency extended with communication channels ('ports'). The ports represent state (i.e. they have internal memory to hold messages). The message passing happens asynchronously. The receiver decides when to handle an incoming message. Messages are handled in the order in which they arrive. Until they are handled they are kept in the port's internal memory. Many threads can send messages to the same port and it is generally not known in which order this happens. This is a source of observable non-determinism which makes reasoning about the programs harder but increases the expressiveness. This type of concurrency forms the basis of the Erlang language [8] used in specification of telecommunication systems.
- Shared-state concurrency adds the concept of explicit state to concurrency. Explicit state is what a C programmer would call a variable: a memory location where a result can be stored and changed. This allows modeling of concurrent object-oriented systems. It is a model in which threads update passive objects using coarse-grained *atomic* actions.

- Algebraic approaches to modeling concurrent systems are Communicating Sequential Processes (CSP) [106], Calculus of Communicating Systems (CCS) [150] and Algebra of Communicating Processes (ACP) [19].
  - Of these three CSP is the most abstract (it identifies more processes than the other two) and tends towards a specification language. CSP strives to have a simple semantic model in which it is easy to define new operators.
  - The other two, CCS and ACP, are based on the same notion of equivalence (bisimulation[5] equivalence) and are more operationally oriented than CSP. They tend towards a programming language. CCS has links to logic and λ-calculus and ACP may be best characterized as a purely algebraical approach.
    - CCS was designed to have a minimal number of operators with communication as its central concept.

[4] Non-determinism in this context means that at some point in time there is more than one possible future. Non-determinism occurs naturally with multi-threading because often multiple threads can be chosen to execute next. Observable non-determinism is non-determinism which can result in different computation results (e.g. due to a race condition).

[5] In theoretical computer science a bisimulation is an equivalence relation between state transition systems associating systems which behave in the same way in the sense that one system simulates the other and vice-versa. Intuitively two systems are bisimilar if they match each other's moves. In this sense each of the systems cannot be distinguished from the other by an observer.

    * CSP underlies the language Occam [107]. The Language of Temporal Ordering Specification (LOTOS) [160] is based on early versions of CSP and CCS. A comparison of CCS, CSP and LOTOS is presented in [62].

  – The three approaches define complex systems in terms of processes and a number of operators to model process interaction. Properties of such systems can then algebraically be verified e.g. as related to deadlock or starvation.

- Concurrency can also be represented using concurrent finite state machines (CFSM) or Petri nets (PN).

  Hierarchical concurrent FSMs (HCFSM) are FSMs in which a state hierarchically can be further refined to a set of sub states (another FSM) or even be decomposed in multiple concurrently active FSMs which can communicate. A well-known model based on HCFSMs is StateCharts [85] (and its variants [59]). In StateCharts time is specified using linear inequalities in the form of time-outs. Different concurrency models are in use in combination with FSMs.

  – A first concurrency model that is popular in combination with FSMs is Synchronous Data Flow and its variants (see section 2.3 about modeling data).

  – The synchronous/reactive concurrency model consists of a set of blocks communicating instantly through unbuffered directed arcs. Execution occurs at a sequence of discrete instants. Deterministic execution is ensured by imposing certain constraints on the functionality of the blocks and the execution order. The synchronous/reactive concurrency model is suitable for applications with complex concurrent control flow and tight synchronization as is present in e.g. safety-critical real time systems. The tight synchronization on the other hand can cause overspecification for other types of applications. A notion of relative time (an ordering) known throughout the system exists but absolute time between two events has no significance.

  – The discrete-event concurrency model carries a notion of global time that is instantaneously known throughout the system. Events occur, activate blocks and cause new events. Each event carries a time stamp. This time stamp is generated by the block creating the event (resulting from the time stamp of the input events and the block's latencies). The discrete-event concurrency model is typically used in specification of hardware and telecom systems. The notion of global time known throughout the system causes problems of overspecification for large VLSI designs with high clock rates. A second problem with the discrete-event concurrency model is the relative expensive

mapping to software implementation. Time between two events has significance in the discrete-event concurrency model.

- FSMs have been extended to model concurrency using the previously discussed concurrency models. The concurrency semantics normally are tied closely to the FSM semantics, e.g. a synchronous/reactive concurrency model [18] in the ARGOS language [142] and StateCharts [85], a discrete-event concurrency model [36] in codesign finite state machines [46] used in POLIS [13], a Petri net concurrency model in the system description language (SDL) [181]. In [134] an approach which separates sequential execution (modeled by an FSM) and concurrent execution (modeled by one of many concurrency models) is proposed. Such models are used as a basis for implementing languages. Synchronous languages (besides ARGOS) are the ESTEREL language [20], the LUSTRE language [83], the SIGNAL language [4] and the graphical formalisms SpecCharts [68] and SyncCharts [5]. ESTEREL, SyncCharts and StateCharts are mainly useful for control-dominated reactive designs. LUSTRE and SIGNAL are mainly useful for data-dominated reactive designs.

- Petri nets are a formal yet intuitive model for non-determinism, concurrency and synchronization. Petri nets have been constrained and extended in numerous ways. Restrictions can be imposed to avoid making the Petri nets Turing complete in order to be able to reason about properties like deadlock, halting, safety, etc. Extensions that have been proposed to Petri nets include

  - Colored Petri Nets: giving tokens a color (making them distinguishable) [114].
  - Predicate/transition nets are Petri nets in which transitions contain expressions and arcs contain conditions [28].
  - Environment relationship nets [71] are Petri nets where tokens are functions associating values to variables.
  - Product nets [31] introduce new types of arcs (inhibitor arc, erase arc), distinguishable tokens, predicates associated to transitions and a domain associated to places.
  - In higher-order Petri nets [113] tokens can themselves consist of Petri nets and Petri nets can be used as return values or arguments in transitions.
  - Object oriented Petri nets have a concept of objects and classes, polymorphism and dynamic binding [231].
  - Stochastic Petri nets are used for performance analysis (by solving a set of equations) [139].
  - Dual Transitions Petri Nets (DTPN) [205] (see section 2.3) have no notion of time or hierarchy. They provide two types of tokens,

transitions and arcs (control flow and data flow). Control flow arcs are guarded with expressions that depend on the values of the data tokens. The guards thus form the connection between the data flow model and the control flow model.

The MTG* model briefly described in chapter 3 of this thesis explicitly represents task-level concurrency by means of Petri net semantics. Concurrency at the loop level or instruction level can be analyzed from the intra task data flow arcs. Petri nets are used because they form a natural representation of concurrency and non-determinism: two properties that are present in the application domains that are targeted by the TCM methodology. The application domain targeted by TCM comprises dynamically behaving applications with soft real-time constraints. Dynamic behavior is both data dependent behavior (causing varying workload with different input data) and non-deterministic behavior (e.g. events generated due to unpredictable user interaction). Example applications can be found in the multimedia and telecom.

2.5 Liveness

Programs for embedded systems often are expected to keep running forever. This is typical in the streaming programming model [188]. Computations should never be waiting for data that will never arrive (the problem of deadlock), and concurrent computations must get some processor time to run to completion (the problem of starvation). Whereas in software an algorithm that does not terminate is usually considered something faulty, it is often expected from embedded software.

Streaming computations can be expressed in MTG* by explicitly indicating the loops that need to be followed.

2.6 Use of hierarchy and abstraction

Complex software designs are kept manageable by using component technologies.

- The first form of component technology is the use of procedures (or functions) with pre-defined arguments. To use a procedure only the application programmer's interface (API) must be known. The complexity inside the implementation of a procedure can remain hidden for a programmer. The specification of procedure/function headers enables separate compilation which is common practice today for reasons of scalability. Models

that have been proposed to represent algorithms in a procedural language include flow charts and Nassi-Schneiderman diagrams [217].

- The ideas of hiding unnecessary implementation details from a programmer, exposing only interfaces and encapsulating functionality with the data it operates on in one abstract data type (a 'class') have lead to object-oriented design [230] which uses models like UML [157] to represent objects and their interactions. Again the programmer does not need to know the implementation details of an object in order to use it.

- The fact that procedures can call other procedures and that objects can contain other objects enables to build a hierarchy of functionalities. Because of programming techniques like function pointers, polymorphism, inheritance and dynamically loadable libraries (plug-ins) this hierarchy cannot be fully known at compile-time in general, making reasoning about the systems more difficult. At the same time, while making programmer's lives easier because algorithms can be written more generically, run-time overhead is introduced. Some research has been done into automatically removing some of this overhead by replacing dynamically dispatched function calls to static function calls where possible [77].

- Low-level memory management in large designs is a source of difficult-to-find bugs like memory leaks. Automatic garbage collection [146] is a solution for memory management which was first implemented in 1958 by John McCarthy in his LISP language [144]. While it generally works for many software projects it introduces problems with timing. The garbage collection works in the background and steals processor cycles causing unpredictable variations in execution time. This makes garbage collection unsuitable for real time systems. Some research is being done to remedy this [12, 117], but it remains to be seen if such systems can become efficient enough as compared to manual memory management especially with respect to energy consumption.

- Correctness of a software specification is aided by the use of static and dynamic type systems [35].

  - Static typing enables a compiler to detect errors and warn about possible problems at compile-time. It enables a compiler to do better optimization and avoids run-time type checking overhead.
  - Dynamic typing makes for more flexible and generic specification of algorithms but introduces run-time type checking and automatic conversion overhead.
  - Many languages with static typing have back doors (by means of type casts and void pointers) to circumvent the stringent constraints imposed by a static type system. These back doors, when used, make it as difficult as dynamic typing for compilers to analyze and

optimize. C++ [191] is regarded as a statically typed language, but a run-time type information (RTTI) feature which allows dynamic typing exists.

Many problems that matter for embedded software cannot easily be found at compile-time — especially problems related to dynamic behavior and time because such aspects are not represented in the interface definitions. In the MTG* and gray box model described in section 3 a notion of hierarchy exists.

- At the MTG* task level a concept of tasks containing thread frames is defined.
- In the gray box model (an extension of MTG* to make it better suitable for TCM) a concept of hierarchical thread node containing thread nodes and CDFG nodes exists.

Information in the gray box model is represented as a graph of so-called thread nodes (which can be thought of as function calls) with selected details (e.g. C++ conditional constructs) left visible. The thread nodes can be considered black boxes. The selected details can be thought of as small parts of the white box model (where all details are visible). The result is a gray box model. This is done in such a way that task-level exploration of the system originally described in a high level language like C or C++ can still give meaningful results when done on the gray box model without the need to know all possible details of the implementation.

2.7 Heterogeneity

Heterogeneity is an intrinsic part of computation in embedded systems. Often the software parts are written to operate on hardware parts that were specifically designed for the application. Embedded applications may need to react to both periodic events and aperiodic events.

- Periodic events occur e.g. in data acquisition applications.
- Aperiodic events occur e.g. when users press buttons and expect the system to react.

Two approaches can be taken for modeling and simulation of heterogeneous designs.

- Designs incorporating several of these aspects in different parts of the design will benefit from mixing specialized models of computation. This is supported in the Ptolemy-II system [23].
- An alternative approach is to use a general-purpose programming language and implement a number of primitives of models of computation which can be combined to model systems. This approach is taken in the system description language SystemC [192] which is built on top of C++ [191].
- In Ptolemy-II the primitives of the models of computation are language features. In SystemC the primitives can be seen as design patterns [69]. YAPI [55] is an example of Kahn process networks implemented on top of SystemC.

Different parts of an application benefit from different models of computation.

- Data intensive parts of an application benefit from data flow like models whereas the surrounding control layer benefits from control oriented models.
- Hardware parts of a design are traditionally described as finite state machines with data path (FSMD) [67] in hardware description languages like VHDL [110], Verilog [109] or higher-level languages like OCAPI [81] and SystemC [192].
- The software parts of an embedded application are often programmed in a general-purpose language like C [123], C++ [191] or JAVA [118].
- In systems where it is not yet decided what parts will go in hardware or software a hardware/software codesign model [51] is used, e.g. codesign FSMs in POLIS [13] or hierarchical FSMs combined with data flow models in CoCentric System Studio [29]. Co-design FSMs consist of FSMs which internally operate synchronously and which communicate with each other by means of asynchronous signals. Co-design models can be implemented using languages like ESTEREL [20]. Some languages are specialized for one application domain: e.g. SDL [181] and Erlang [8] for telecommunication systems.
- Synchronous languages are not a good match for situations where (network) communication latencies are large. Distributed designs using the APIs provided by systems like RT-CORBA [16] are not a good match for highly specialized DSP systems.

The system property intervals (SPI) model [235] was especially designed for use in heterogeneous designs. It is possible to specify fuzzy knowledge about

a design by using behavior intervals. In addition, different process modes can be distinguished. SPI has the drawback of not distinguishing clearly between data flow and control flow. This distinction was added in its successor FUNSTATE [190].

The MTG* model described in chapter 3 consists of a graph with nodes that can be characterized with property intervals (e.g. minimum and maximum execution time or minimum and maximum energy consumption). Event periods and jitter (if known) can be annotated as well. Currently an implementation of MTG* constructs has been made in the Mozart-Oz3 language [155]. Prototype TCM flow tools that are being developed at IMEC implement MTG* on top of C++ with calls to the Tipsy library [207] for modeling time, concurrency, task creation and events. This way of working is similar to the SystemC approach of defining model primitives on top of a high-level language.

The task level model of MTG* uses Petri nets because they are a suitable representation for the application domain under consideration. Petri nets are a good model for systems where concurrency, synchronization and event handling must be represented. Inside the Petri net transitions a CDFG model is present. CDFG models are traditionally used in compilers and hardware/software synthesis tools.

2.8 Reactivity

In literature a distinction is made between reactive systems, interactive systems and transformational systems [136].

- Reactive systems are systems that react continuously to their environment with the speed of the environment.

  Reactive systems have real-time constraints and are frequently safety-critical. They must be able to adapt to changing conditions: service demands, computing resources and sensors may appear and disappear.

- Interactive systems react with the environment at their own speed.

- Transformational systems take input data and transform it to output data, often with no stringent timing constraints.

Purely transformational systems can be represented using the MTG* intra task model only. In fact using MTG* is not recommended because what sets MTG* and gray box apart from other models is mainly their concept of thread frame and thread node respectively. In purely transformational systems the whole systems can often be represented as a single thread node.

MTG* and gray box as described in chapter 3 are aimed at modeling reactive and interactive systems with soft real-time constraints. Hard real-time timing constraints can be annotated in MTG* but the TCM methodology cannot optimize applications with hard real-time annotations very well due to the need for making worst-case assumptions. With soft real-time constraints optimizations can be more aggressive because an occasional deadline can be missed without further consequences.

2.9 Dynamism

In mobile embedded systems a growing number of applications is supported ranging from calendar to video decoding and playing 3D games. Such systems need to handle a highly dynamic workload and a variable number of active tasks in the system. At any point in time new tasks can be requested by the user by means of non-deterministically occurring user events. Other tasks can be deleted from the system. A video decoder task could demultiplex and decode any number of incoming video and audio data streams.

To accommodate the dynamically changing available resources during the execution of an embedded application the quality of service (QoS) can be varied. Algorithms can be tuned at run-time to adapt to limited available computing, memory and network bandwidth resources [82].

In TCM [225] orthogonally to implementing QoS at the software level the hardware platform's capabilities can be scaled down at run-time to save energy.

We distinguish between two forms of dynamic behavior:

- Task level dynamism: MTG* provides annotations for dynamic task creation and dynamic task deletion, dynamic data port creation and events that enter the system from the environment.
- Intra task level dynamism: Varying workload in the intra task MTG* model is caused by data dependent conditions and non-manifest loop bounds. Such conditions and loop bounds will be represented in between the thread nodes that form the gray box model.

2.10 Conclusions

Many models already exist. MTG* reuses from existing models those features that are useful in a TCM context. Several aspects of MTG* are not new. The

concepts of MTG* thread node and MTG* thread frame, however, are unique to MTG* and crucial for enabling TCM.

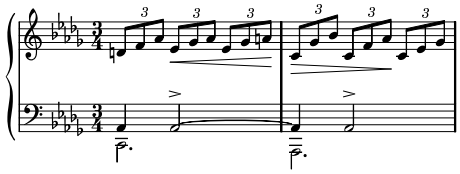
MTG* thread frames partition the system in two layers, leading to a distinction between task-level and intra task-level functionality. On the intra task-level, MTG* thread nodes separate important detailed information from less important details. The meaning of importance here depends on the task at hand. During transformation other details will be considered important than during scheduling. This leads to a hierarchy of MTG* tasks and thread frames at the task-level, and MTG* thread nodes and CDFG on the intra task-level.

With respect to execution semantics, the MTG* task-level bears most resemblance to Petri Nets. In fact this was already the case in its predecessor MTG. MTG* therefore is good for modeling concurrent systems, such as the ones considered for optimization by TCM. MTG* uses a shared memory communication model. Writing to memory or reading from it does not imply synchronization. This is needed because MTG* is typically used to represent C++ applications which also allow reading/writing from/to memory without synchronization. The MTG* task-level model shows concurrency at the task-level. Concurrency at the intra-task level is represented using the intra-task control and data flow graphs.

The way time information is represented in MTG* does not differ significantly from other models in the literature. The presence of timing constraints in the application is crucial for TCM to allow for making trade-offs between execution time and energy consumption.

Chapter 3

MTG* summary



Franz Schubert, Impromptu op.142,no.2

3.1 Introduction

TCM is able to save energy by exploiting the dynamic behavior and the task level parallelism inside an application. To make this possible in a systematic way a clear representation of the application's dynamic behavior and task level parallelism is desired. It should also be possible to annotate extra knowledge, e.g. as related to timing constraints or guarantees. Pure C, C++ or JAVA code from which one normally starts to apply TCM to is not suited for that purpose.

The MTG* on the other hand is a model that can be used for these purposes. Before MTG* a set of informal block diagrams were used. The block diagrams described a kind of task graph that was used in the TCM design-time scheduler. Also for the run-time scheduler the need to have a model that represents the task level parallelism was identified. While applying TCM to real life examples it became clear that MTG* could reuse much of the existing MTG (**M**ulti **T**hread **G**raph) model [196]. Some extensions were needed for features

missing in MTG. These extensions are based on similar extensions present in the DF* (**D**ata **F**low graphs with **STA**te machine based **R**econfiguration) model [52, 154].

MTG* describes an application in two levels. The most abstract level giving an overview of the application's dynamic behavior and coarse grained parallelism is called the task level model. The task level model consists of a collection of MTG* thread frames. Thread frames are maximally sized pieces of functionality that can be activated separately, and in which nothing happens which could result in an activation of the TCM run-time scheduler [221]. In practice this means that inside a thread frame no synchronization is allowed, no tasks can be dynamically created and no events are handled. Such things are shown explicitly in the MTG* task level model. The rest is hidden inside the thread frames. The task level model serves as a starting point for the run-time scheduling preparation step in TCM. The model clearly shows when to activate the run-time scheduler. Task level source code transformations are transformation that change this task level model to either create more optimization freedom at run-time, or to reduce the amount of overhead at run-time.

Within each task the detailed functionality is described using the second level of MTG*: the intra task level. The model currently in use in MTG* is the DSFG (**D**ecorated **S**ignal **F**low **G**raph) model, which will not be described further in this thesis as it is fully described in [130].

The DSFG describes the internals of a thread frame in all its details. Usually this is too detailed for a tool like the design-time scheduler. For this reason the MTG* model was further extended with the concept of a thread node. Thread nodes are a way to abstract the intra task model. Important details for a given analysis or task at hand can be highlighted whereas less important details can be hidden inside the thread nodes. This abstracted version of the intra task model forms a so-called gray box description of the application. This is the level on which the TCM design-time scheduler [218] works.

In this thesis the MTG* model is important because it also forms the basis for the code transformations which form the main part of the thesis. The semaphore transformation of chapter 5 directly uses the information in the task level of MTG*. The recursion removal technique in chapter 6 transforms a bigger thread node — consisting of the complete recursive function — to a loop which repeatedly activates a smaller thread node — the loop body. This increases the freedom for the design-time scheduler.

3.2 MTG* task level model syntax

In what follows $a.foo$ indicates the attribute 'foo' of a. $a(foo = bar)$ denotes an a for which $a.foo = bar$. Similarly $A(foo = bar)$ denotes the set of elements $e \in A$ for which $e.foo = bar$. Sets are usually notated with a name starting with a capital. Elements have a name starting with a lowercase letter. All notations for elements can be extended and used with sets. $a(foo = bar, egg = spam)$ is used to denote an a for which $a.foo = bar \wedge a.egg = spam$. We note $a(foo \in S)$ to indicate that $a.foo \in S$. The notation $\#A$ gives the number of elements (cardinality) of set A.

An MTG* model M consists of a set $M.O$ of nodes o_i and a set $M.E$ of edges e_j:

$$M = (O, E)$$

3.2.1 MTG* nodes

Nodes o have a *type*, *Controlportlist*, *Dataportlist*, *extras* and *annotation* attribute:

$$o = (type, Controlportlist, Dataportlist, extras, annotation)$$

for a node o,

$$o.type \in \{dynamictask, task, threadframe, stopcluster, diamond, and, \\ deterministicOR, sync, event, synchro, sharedmemoryvariable, \\ systemI, systemO\}$$

An $o(type = foo)$ node in an MTG* model M is often concisely referred to as an *MTG* foo node* or even a *foo node*. The *extras* attribute has a different definition depending on the type of node.

In what follows the terminology Nodelist is used to denote a set of nodes o: Nodelist $= \{\ o \in M.O\ \}$. In what follows we only consider Nodelists which contain at least one dynamic task, task or thread frame: $N\ is\ a\ Nodelist \Rightarrow \exists o(type \in \{dynamictask, task, threadframe\}) \in N$

A Tflist is a Nodelist containing only dynamic task, task and thread frame nodes. $\forall o \in Tflist : o(type \in \{dynamictask, task, threadframe\})$

Execution rates are stored in the rtcs attribute:

$$rtcs = (max\ inst\ rate, min\ inst\ rate, \\ max\ average\ rate, min\ average\ rate)$$

Execution time related information is stored in a tuple Λ.

$$\Lambda = \quad (\delta, \Delta, eft),$$

δ	is minimum needed execution time of node to which it belongs
Δ	is maximum needed execution time of node to which it belongs
eft	is the elapsed firing time of the node, initialized while firing and updated during execution

λ = the actual execution time, depends from activation to activation : $\delta(o) \leq \lambda(o) \leq \Delta(o)$

$$Place = a\ set\ of\ tokens$$

Then:

$$\begin{aligned}
o(type = dynamictask).extras &= (amount, low, up, Nodelist, annotation,\\
&\quad rtcs, \Lambda, status),\\
&\quad low \leq amount \leq up \in \mathbb{N}\\
&\quad status \in \{creatable, created\}\\
o(type = task).extras &= (Nodelist)\\
o(type = threadframe).extras &= (annotation, rtcs, cdfgmodel, \Lambda, status)\\
&\quad cdfgmodel = (dsfg, annotation)\\
&\quad status \in \{runnable, running, suspended,\\
&\quad terminated\}\\
&\quad status\ default\ value\ is\ runnable\\
o(type = stopcluster).extras &= (Tflist)\\
o(type = diamond).extras &= (Place, rtcs)\\
o(type = and).extras &= (rtcs)\\
o(type = deterministicOR).extras &= (rtcs)\\
o(type = sync).extras &= (rtcs, timeout)\\
&\quad timeout \in \mathbb{R}^+ \cup \{+\infty\}\\
&\quad timeout\ default\ value\ is\ +\infty\\
o(type = event).extras &= (period, jitter, t_{min}^{arr}, rtcs)\\
o(type = synchro).extras &= (period, rtcs)\\
o(type = sharedmemoryvariable) &= (annotation)\\
o(type = systemI) &= (annotation)\\
o(type = systemO) &= (annotation)
\end{aligned}$$

The gray box model can use the $threadframe.extras.annotation$ field to store the Pareto curves for each scenario extracted by the TCM design-time scheduler.

3.2.1.1 Control ports

A Controlportlist is a set of controlports cp_i.

$$cp_i \in Controlportlist \Rightarrow cp_i \ is \ controlport$$

A controlport cp_i has an attribute 'type'.

$$cp_i.type \in \{entry, exit\}$$

A controlport cp_i has an attribute 'weight'.

$$cp_i.weight \in \mathbb{N}_0$$

If no weight is specified the default value is 1. A controlport cp_i has an attribute 'label'. If no label is specified the default value is the empty string.

3.2.1.2 Data ports

A Dataportlist is a set of dataports dp_i.

$$dp_i \in Dataportlist \Rightarrow dp_i \ is \ dataport$$

A dataport dp_i has an attribute 'direction'.

$$dp_i.direction \in \{in, out\}$$

A dataport dp_i has an attribute 'type'.

$$dp_i.type \in \{primitive, hierarchical, multi\}$$

A data port dp_i has an attribute 'range'.

$$dp_i.range \in \mathbb{N}_0$$

3.2.1.3 Port cardinality restrictions

A number of restrictions on the cardinality of Controlportlist and Dataportlist exist, depending on the type of the node to which they belong.

$$\begin{aligned}
\#o(type = task).Controlportlist(type = entry) &\quad \geq 1 \\
\#o(type = task).Controlportlist(type = exit) &\quad \geq 1 \\
\#o(type = task).Dataportlist(direction = in) &\quad \geq 0 \\
\#o(type = task).Dataportlist(direction = out) &\quad \geq 0 \\
\#o(type = dynamictask).Controlportlist(type = entry) &\quad = 1 \ or \ 3 \\
\#o(type = dynamictask).Controlportlist(type = exit) &\quad = 1
\end{aligned}$$

$$
\begin{aligned}
\#o(type = dynamictask).Dataportlist(direction = in) &\geq 0\\
\#o(type = dynamictask).Dataportlist(direction = out) &\geq 0\\
\#o(type = threadframe).Controlportlist(type = entry) &= 1\\
\#o(type = threadframe).Controlportlist(type = exit) &= 1\\
\#o(type = threadframe).Dataportlist(direction = in) &\geq 0\\
\#o(type = threadframe).Dataportlist(direction = out) &\geq 0\\
\#o(type = diamond).Controlportlist(type = entry) &\geq 0\\
\#o(type = diamond).Controlportlist(type = exit) &\geq 0\\
\#o(type = diamond).Dataportlist &= 0\\
\#o(type = and).Controlportlist(type = entry) &\geq 1\\
\#o(type = and).Controlportlist(type = exit) &= 1\\
\#o(type = and).Dataportlist &= 0\\
\#o(type = deterministicOR).Controlportlist(type = entry) &\geq 1\\
\#o(type = deterministicOR).Controlportlist(type = exit) &= 1\\
\#o(type = deterministicOR).Dataportlist(direction = in) &= 1\\
\#o(type = deterministicOR).Dataportlist(direction = out) &= 0\\
\#o(type = sync).Controlportlist(type = entry) &= 2\\
\#o(type = sync).Controlportlist(type = exit) &= 1\\
\#o(type = sync).Dataportlist &= 0\\
\#o(type = event).Controlportlist(type = entry) &= 0\\
\#o(type = event).Controlportlist(type = exit) &= 1\\
\#o(type = event).Dataportlist &= 0\\
\#o(type = synchro).Controlportlist(type = entry) &= 1\\
\#o(type = synchro).Controlportlist(type = exit) &= 0\\
\#o(type = synchro).Dataportlist &= 0\\
\#o(type = stopcluster).Controlportlist(type = entry) &= 2\\
\#o(type = stopcluster).Controlportlist(type = exit) &= 0\\
\#o(type = stopcluster).Dataportlist(direction = in) &= 1\\
\#o(type = stopcluster).Dataportlist(direction = out) &= 1\\
\#o(type = sharedmemoryvariable).Controlportlist &= 0\\
\#o(type = sharedmemoryvariable).Dataportlist(direction = in) &\geq 0\\
\#o(type = sharedmemoryvariable).Dataportlist(direction = out) &\geq 0\\
\#o(type \in \{systemI, systemO\}).Controlportlist &= 0\\
\#o(type = systemI).Dataportlist(direction = in) &= 0\\
\#o(type = systemI).Dataportlist(direction = out) &= 1\\
\#o(type = systemO).Dataportlist(direction = in) &= 1
\end{aligned}
$$

$$\#o(type = systemO).Dataportlist(direction = out) \quad =0$$

The entry control ports of an $o(type = task)$ node are the entry control ports of the thread frames in the $o(type = task).extras.Nodelist(type = threadframe)$, in which an edge arrives that departs from a node outside the task.

The exit control ports of an $o(type = task)$ node are the exit control ports of the thread frames in the $o(type = task).extras.Nodelist(type = threadframe)$, from which an edge departs that arrives in a node outside the task.

The entry control ports of an $o(type = dynamictask)$ node have been assigned labels. If only one entry control port is present, it is labeled 'activate'. If three entry control ports are present they are labeled 'activate', 'create' and 'delete'. The control port weights of the 'create' and 'delete' control port must be equal to 1.

The entry control ports of an $o(type = sync)$ node have been assigned labels: 'stop' and 'resume'. The control port weights of the 'stop' and 'resume' control port must be equal to 1.

The entry control ports of an $o(type = stopcluster)$ node have been assigned labels: 'stop' and 'resume'. The control port weights of the 'stop' and 'resume' control port must be equal to 1.

Partially overlapping tasks are not allowed: if two tasks have thread frames in common, one task must be completely contained within the other one.

$$\begin{aligned}
&\forall o_i(type = task) \in M.O, o_j(type = task) \in M.O : \\
&\quad o_i.extras.Nodelist \cap o_j.extras.Nodelist \neq \{\} \Leftrightarrow \\
&\quad o_i.extras.Nodelist \subset o_j.extras.Nodelist \\
&\quad \vee o_j.extras.Nodelist \subset o_i.extras.Nodelist
\end{aligned}$$

3.2.1.4 Named nodes and node subsets

Some useful node subsets are defined.

- The subset of *operational nodes* $M.O_{op}$ of an MTG* model M are nodes o for which $o.type \in \{threadframe, diamond, and, deterministicOR, sync, event, synchro\}$.
- The subset of *control flow nodes* $M.O_{ctrl}$ of an MTG* model M are nodes o for which $o.type \in \{diamond, and, deterministicOR, sync, event, synchro\}$.
- The subset of *behavioral nodes* $M.O_{beh}$ of an MTG* model M is the set $M.O_{op} \setminus M.O_{ctrl}$.

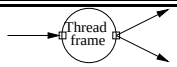
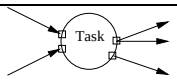
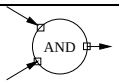
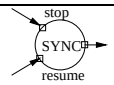
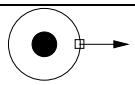
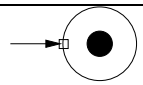
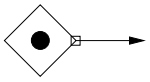
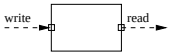
type of node	graphical representation	#entry control ports	#exit control ports	#data ports
MTG* thread frame		1	1	>= 0, both in/out
MTG* task		>= 1	>= 1	>= 0, both in/out
MTG* diamond node		>= 0	>= 0	0
MTG* AND node		> 1	1	0
MTG* sync node		2	1	0
MTG* deterministic OR node		> 1	1	1, of type in
MTG* event node		0	1	0
MTG* synchro node		1	0	0
MTG* source node (form of diamond node)		0	>= 1, although usually == 1	0
MTG* sink node (form of diamond node)		>= 1	0	0
MTG* nondeterministic OR node (form of diamond node)		> 1	1	0
MTG* semaphore (form of diamond node)		> 1	> 1	0
MTG* shared memory variable		0	0	>= 1 of type in, >= 1 of type out

Table 3.1: Overview of possible number of entry/exit control ports and data ports in MTG* nodes

- The subset of *data nodes* $M.O_{dat}$ of an MTG* model M are nodes o for which
  $o.type \in \{sharedmemoryvariable, systemI, systemO\}$.
- The subset of *passive nodes* $M.O_{pas}$ of an MTG* model M are nodes o for which
  $o.type \in \{diamond, sharedmemoryvariable, systemI, systemO\}$
- The subset of *active nodes* $M.O_{act}$ of an MTG* model M are the nodes $M.O \setminus M.O_{pas}$

In the current state of MTG* operational nodes have operational semantics (firing and/or execution rules), data nodes not. Synchronized communication can be modeled explicitly by adding control flow edges and control flow nodes.

In specific configurations, the diamond node has been given a more convenient name.

$$\begin{cases} o(type = diamond).Controlportlist(type = entry) & =0 & \\ o(type = diamond).Controlportlist(type = exit) & \geq 1 & \text{: source node} \\ \#o(type = diamond).extras.Place & =1 & \\ o(type = diamond).Controlportlist(type = entry) & \geq 1 & \\ o(type = diamond).Controlportlist(type = exit) & =0 & \text{: sink node} \\ \#o(type = diamond).extras.Place & =0 & \\ o(type = diamond).Controlportlist(type = entry) & \geq 2 & \\ o(type = diamond).Controlportlist(type = exit) & \geq 2 & \text{: semaphore} \\ \#o(type = diamond).extras.Place & \geq 1 & \\ o(type = diamond).Controlportlist(type = entry) & \geq 1 & \\ o(type = diamond).Controlportlist(type = exit) & =1 & \text{: non-det OR} \\ \#o(type = diamond).extras.Place & =0 & \\ o(type = diamond).Controlportlist(type = entry) & =1 & \\ o(type = diamond).Controlportlist(type = exit) & \geq 1 & \text{: non-det choice} \\ \#o(type = diamond).extras.Place & =0 & \end{cases}$$

There must be exactly one source node and exactly one sink node in an MTG* model.

3.2.2 MTG* edges

Edges have an *type*, *tail* and *head* and an *extras* attribute:

$$e = (type, tail, head, extras)$$

An edge e is directed from its $e.tail$ to its $e.head$. The extras attribute has different contents depending on the edge type.

$$e.type \in \{controlflow, dataflow, timingconstraint, inhibitor\}$$

$$\begin{aligned} e(type = controlflow).extras = (&Place, guardexpression, weight),\\ &weight \in \mathbb{N} \cup \{undefined\},\\ &guardexpression\ evaluates\ to\\ &r \in \{true, false\}\\ &guardexpression\ can\ depend\ on\\ &shared\ memory\ variables \end{aligned}$$

$$e(type = dataflow).extras = ()$$

$$\begin{aligned} e(type = timingconstraint).extras = (&value)\\ &value \in \mathbb{R} \end{aligned}$$

$$e(type = inhibitor).extras = ()$$

If no e.(type=control flow).extras.weight is specified, it defaults to undefined. If no e.(type=control flow).extras.guardexpression is specified, it defaults to true.

3.2.2.1 Edge restrictions

Restrictions on where edges can start and end exist. For an MTG* model M: $\forall edge\ e \in M.E : \exists node\ o \in M.O, \exists edge\ e' :$

$$e(type = controlflow).tail \in o.Controlportlist(type = exit)$$

$$e(type = controlflow).head \in o.Controlportlist(type = entry)$$

$$e(type = dataflow).tail \in o.Dataportlist(direction = out)$$

$$e(type = dataflow).head \in o.Dataportlist(direction = in)$$

$$\begin{aligned} e(type = dataflow).head \in o(type = deterministicOR)\\ \Rightarrow e.tail \in o(type = sharedmemoryvariable).\\ Dataportlist(direction = out) \end{aligned}$$

$$e(type = timingconstraint).tail \in o.Controlportlist$$

$$e(type = timingconstraint).head \in o.Controlportlist$$

$$e(type = timingconstraint).tail \neq e(type = timingconstraint).head$$

$$e(type = inhibitor).tail = e'(type = controlflow).extras.Place$$

$$e(type = inhibitor).head \in o.Controlportlist$$

At most one control flow edge can arrive in a given controlport:

cp is controlport:

$$\#\{M.E(type = controlflow).head | M.E(type = controlflow).head = cp\} \leq 1$$

Unless another timing constraint explicitly is specified an implicit edge $e(type = timingconstraint, value = 0)$ is present in parallel with each $e(type = controlflow)$ that departs from the exit control port of an $o(type = AND)$ node.

3.3 MTG* task level operational semantics

In addition to the structural information expressed by the graph nodes and edges, the model is equipped with a token flow, expressing the system dynamics.

A token marking μ of an MTG* model is a mapping $\mu : e(type = controlflow) \rightarrow \mathbb{N}^{\infty}$. μ, when applied to an edge e, returns the number of tokens currently present in $e.Place$.

$$\mu(e) = \#e.Place$$

An edge $e(type = controlflow)$ is marked if $\mu(e) \geq 1$. Otherwise it is unmarked.

For an MTG* model M, define the preset of a node $o \in M.O$, $Pre(o)$, as the set of incoming edges of node o:

$$Pre(o) = \{e \in M.E | e.head \in o.Controlportlist(type = entry)\}$$

Similarly the postset of a node $o \in M.O$, $Post(o)$ is the set of outgoing edges of node o:

$$Post(o) = \{e \in M.E | e.tail = o.Controlportlist(type = exit)\}$$

3.3.1 Firing rules: enabled nodes

In what follows M is an MTG* model.

3.3.1.1 Behavioral nodes

MTG* tasks have no semantics of their own. They inherit the semantics of their contents. For all nodes $o \in M.O(type = task)$, first substitute the node with its contents $o.extras.Nodelist$, together with all edges e_j that are in the pre- or postsets of elements of $o.extras.Nodelist$.

A thread frame $o(type = threadframe) \in M.O$ is enabled if the number of tokens $\mu(e)$ on its incoming edge $e = Pre(o)$ is larger than or equal to the thread frame's entry control port weight $e.head.weight$, and its $o.extras.status$ is set to $runnable$.

A dynamic task $o(type = dynamictask)$ with one entry control port is enabled if the number of tokens $\mu(e)$ on the edge $e \in Pre(o)$ arriving in the 'activate' controlport $o.Controlportlist(type = entry, label =' activate')$ is larger

than or equal to the node's entry control port weight $e.head.weight$, and its $o.extras.status$ is set to $runnable$.

A dynamic task $o(type = dynamictask)$ with three entry control ports is enabled if the number of tokens $\mu(e_c)$ on the edge $e_c(head.label =' create')$ is bigger than or equal to the corresponding control port weight $e_c.head.weight$ and the number of tokens $\mu(e_a)$ on the edge $e_a(head.label =' activate') \in Pre(o)$ is larger than or equal to the node's entry control port weight $o.Controlportlist(type = entry, label =' activate').weight$, and its $o.extras.status$ is set to $runnable$.

3.3.1.2 Control flow nodes

An and node $o(type = and) \in M.O$ is enabled if the number of tokens $\mu(e)$ on each edge $e \in Pre(o)$ is bigger than or equal to the corresponding entry control port weight $e.head.weight$.

An deterministic OR node $o(type = deterministicOR) \in M.O$ is enabled if the number of tokens $\mu(e)$ on the edge $e(type = controlflow)$, as indicated by the value of the shared memory variable v that is connected to the single $o.Dataportlist(direction = in)$ element, is bigger than or equal to the corresponding entry control port weight $e.head.weight$. As long as the firing has not started, the deterministic OR node can change from being enabled to being not enabled due to changes in the value of v.

A sync node $o(type = sync) \in M.O$ with $o(type = sync).extras.timeout = +\infty$ is enabled if the number of tokens $\mu(e)$ on the edge $e(type = controlflow, label =' resume') \in Pre(o)$ is bigger than or equal to the corresponding control port weight $e.head.weight$. A sync node $o(type = sync) \in M.O$ with $o(type = sync).extras.timeout \in \mathbb{R}^+$ is enabled if the number of tokens $\mu(e)$ on the edge $e(type = controlflow, label =' stop') \in Pre(o)$ is bigger than or equal to the corresponding control port weight $e.head.weight$.

A synchro node $o(type = synchro) \in M.O$ is enabled if the number of tokens $\mu(e)$ on the edge $e(type = controlflow) \in Pre(o)$ is bigger than or equal to the corresponding control port weight $e.head.weight$.

An event node $o(type = event) \in M.O$ is enabled by the outside world which injects a token into it.

A stop cluster $o(type = stopcluster)\ inM.O$ is enabled by a token arriving on the edge connected to either one of its entry control ports.

A diamond node $o(type = diamond) \in M.O$ cannot be enabled or fire. It is merely a collection and distribution point of tokens. Tokens are requested from a diamond node by an active node (through a control flow edge) when it needs a token to become enabled.

3.3.1.3 Data nodes

For now data nodes in MTG* have no operational semantics.

3.3.2 Execution rules

A node $o \in M.O$ which is enabled according to its firing rule can fire resulting in the three phases with associated markings.

3.3.2.1 Thread frame

firing: at time $t = t_{start}$ the tokens needed for enabling the thread frame $o(type = threadframe)$ are tied to the node. They are not yet consumed. The tied tokens can no longer be consumed by a different node than o. The $o.extras.\Lambda.eft$ is initialized to 0. The marking of the edges in $Pre(o)$ could still change due to other nodes completing.

execution: this takes $\lambda(o)$ time. $o.extras.\Lambda.\delta \leq \lambda(o) \leq o.extras.\Lambda.\Delta$. The elapsed firing time $o.extras.\Lambda.eft$ in this phase is defined as $t - t_{start}$.

completion: at time $t_{end} = t_{start} + \lambda(o)$, the guard expressions associated to the outgoing control flow edges $e(type = controlflow).extras.guardexpression$, for $e(type = controlflow) \in Post(o)$ are evaluated.

The number of tokens as indicated in the entry control port weights $cp.weight$, for $cp \in e_i(type = controlflow).Controlportlist(type = entry)$, $e_i \in Pre(o)$ are consumed (removed) from its input edge $e_i(type = controlflow) \in Pre(o)$.

The number of tokens as indicated in the exit control port weights $cp.weight$, for $cp \in e_o.Controlportlist(type = exit), e_o \in Post(o)$ are produced (added) on the output edges $e(type = controlflow) \in Post(o)$ for which the associated guardexpressions $e.extras.guardexpression$ evaluated to true.

If abstraction is made of the elapsed fire time and control port weights, this is equivalent to a Petri net transition.

3.3.2.2 Dynamic task

Dynamic tasks with 1 entry control port follow the same rule as thread frames.

The rule for dynamic tasks with 3 entry control ports is given now:

- If a token is available on the edge arriving in the entry control port labeled 'create', the same rule is used as for thread frames. The entry control

port used in the thread frame's execution rule description corresponds to the entry control port labeled 'activate'.

- If no tokens are available on the edge arriving in the entry control port labeled 'create', the dynamic task o cannot be enabled. Tokens arriving on the edge connected to the entry control port labeled 'activate' are consumed but no time passes and no tokens are produced on the outgoing edges $Post(o)$.
- If a token arrives on the edge connected to the control port labeled 'delete', the token from the edge connected to the control port labeled 'create' is consumed. A task that was executing completes execution first and produces tokens on the outgoing edges for which the associated guardexpressions are true.

A dynamic task can be seen as a tuple containing a data dependent number of MTG* thread frames. The syntax of a dynamic task $o(type = dynamictask)$ has attributes $o.extras.upper$ and $o.extras.lower$, which indicate an upper and lower boundary on the number of tasks that are to be created. The outgoing edge of a dynamic task can be connected to any node. Specific nodes can be used to model specific synchronization mechanisms. A list is given. Let $E_o = Post(o)$ be all the outgoing edges of the dynamic task $o(type = dynamictask)$. Let $e_i \in Pre(o)$ be the incoming edge of the same node o.

- if $e \in E_o$ is connected to an $o(type = and)$ node, all created tasks need to be finished before a token can be produced on the elements of E_o.
- if $e \in E_o$ is connected to an $o(type = sync)$ node and from $e_i.tail$ another edge $e_r(type = controlflow)$ with $e_r.extras.guardexpression = 1$ departs which arrives in the entry control port labeled $resume$ of the node o, then a token can be produced on the outgoing edges E_o of the dynamic task when the first task in the set of created tasks has finished.
- if $e \in E_o$ is connected to an $o(type = deterministicOR)$ it is possible to wait for a specific task in the set of created tasks to finish.
- if $e \in E_o$ is connected to an $o(type = non - deterministicOR)$ a token can be produced on the outgoing edges E_o for each task that finishes in the set of created tasks. This is the default if the outgoing edge of the dynamic task is connected to a different kind of node.

3.3.2.3 And node

firing: At time $t = t_{start}$ the tokens needed for enabling the node o are tied to the node. They are not yet consumed. The tied tokens can no longer go to a different node than o.

execution: This takes no execution time.

completion: The guard expressions associated to the control flow edges $e.extras.guardexpression$,for $e(type = controlflow) \in Post(o)$ are evaluated.

The number of tokens as indicated in the entry control port weights $cp.weight$, for $cp \in e_i(type = controlflow).Controlportlist(type = entry)$, $e_i \in Pre(o)$ are consumed from the input edges $Pre(o)$.

The number of tokens as indicated in the exit control port weight $cp.weight$, for $cp \in e_o.Controlportlist(type = exit), e_o \in Post(o)$ are produced on the output edges $e(type = controlflow) \in Post(o)$ for which the associated guardexpressions $e.extras.guardexpression$ are true.

3.3.2.4 Deterministic OR node

firing: At time $t = t_{start}$ the tokens needed for enabling the node o are tied to the node. They are not yet consumed. The tied tokens can no longer go to a different node than o. The current value of the shared memory variable v associated to an $o(type = deterministicOR)$ node is taken into account for the rest of this execution. Its value indicates to which entry control port the node is listening. Changes in its value during firing, execution or completion have no effect on the firing, execution and completion phases.

execution: This takes no execution time.

completion: The guard expressions associated to the control flow edges $e.extras.guardexpression$, for $e(type = controlflow) \in Post(o)$ are evaluated.

The number of tokens as indicated in the entry control port weights $cp.weight$, for $cp \in e_i(type = controlflow).Controlportlist(type = entry)$, $e_i \in Pre(o)$ are consumed from the input edge $e_i \in Pre(o)$ as indicated by the value of v.

The number of tokens as indicated in the exit control port weight $cp.weight$, for $cp \in e_o.Controlportlist(type = exit), e_o \in Post(o)$ are produced on the output edges $e(type = controlflow) \in Post(o)$ for which the associated guardexpressions $e.extras.guardexpression$ are true.

3.3.2.5 Sync node

The behavior is different depending on whether a token is present on the edge e_s connected to the entry control port labeled *stop* of the sync node or not. Call e_r the edge connected to the entry control port labeled *resume* of the sync node.

Two $o(type = sync)$ configurations are allowed:

1. $o.extras.timeout = +\infty$
2. $o.extras.timeout \in \mathbb{R}^+$

For the first configuration:

firing: At time $t = t_{start}$ the tokens needed for enabling the node o are tied to the node. They are not yet consumed. The tied tokens can no longer go to a different node than o.

execution: This takes no execution time.

completion: If no token is present on e_s, a token on e_r is consumed, and no token is produced. If at least one token is present on e_s, a token arriving on e_r is consumed together with a token on e_s and a token is produced on the node's outgoing edges $e_o \in Post(o)$ for which the associated guardexpressions are true.

For the second configuration:

firing: At time $t = t_{start}$ the tokens needed for enabling the node o are tied to the node. They are not yet consumed. The tokens can no longer go to a different node than o.

execution: If no token appears on the edge connected to the control port labeled 'resume' within the amount of time specified by the $o.extras.timeout$ value the node times out.

completion: If the node did not time out, the completion is the same as in the first configuration with at least one token present on the edge e_s. If the node did time out, the token on the control port labeled 'stop' is consumed and a token is produced on the node's outgoing edges $e_o \in Post(o)$ for which the associated guardexpressions are true.

3.3.2.6 Synchro node

firing: At time $t = t_{start}$ the tokens needed for enabling the node o are tied to the node. They are not yet consumed. The tied tokens can no longer go to a different node than o.

execution: This takes no execution time.

completion: The number of tokens as indicated in the entry control port weights $cp.weight$, for $cp \in e_i(type = controlflow).Controlportlist(type = entry), e_i \in Pre(o)$ are consumed from the input edge $e_i \in Pre(o)$.

3.3.2.7 Event node

firing: The node fires when the outside world injects a token into the model.

execution: This takes no execution time.

completion: The number of tokens as indicated in the exit control port weight $cp.weight$, for $cp \in e_o.Controlportlist(type = exit), e_o \in Post(o)$ are produced on the output edges $e(type = controlflow) \in Post(o)$ for which the associated guardexpressions $e.extras.guardexpression$ are true.

3.3.2.8 Diamond node

Despite the fact that a diamond node can have multiple control ports, it cannot be enabled or fire. It is merely a collection of tokens. Active nodes can ask a token from the diamond node through their common control flow edge if they need it to become enabled.

For each control flow edge $e(type = controlflow)$ that arrives on a diamond node $o(type = diamond)$, tokens that arrive on the edge e are put in the diamond node's associated place $o(type = diamond).extras.place$. A token in the diamond node's place can be put on exactly one edge $e \in Post(o)$ for which the associated guardexpression is true. If multiple edges have guardexpressions which are true the token goes to one of them. This choice is made non-deterministically.

The guardexpressions of all the edges $e \in Post(o)$ are evaluated upon receiving a token request.

3.3.3 Stop cluster

A stop cluster $o(type = stopcluster)$ contains a list of thread frames $o.extras.Tflist$. Call e_s the edge connected to the stop cluster's entry control port labeled *stop*. Call e_r the edge connected to the stop cluster's entry control port labeled *resume*.

firing: At time $t = t_{start}$ the tokens needed for enabling the node o are tied to the node. They are not yet consumed. The tokens can no longer go to a different node than o.

execution: This takes no execution time.

completion:

- If a token is present on e_s but not on e_r, the $extras.status$ attribute of each thread frame t in the $o.extras.Tflist$ is set to *suspended*. The token is consumed from the e_s edge. The $t.extras.\Lambda.eft$ keeps increasing for t nodes that are executing.

- If a token is present on e_r but not on e_s the $extras.status$ attribute of each thread frame in the $o.extras.Tflist$ is set to $runnable$. The token is consumed from the e_r edge. The $t.extras.\Lambda.eft$ keeps increasing for t nodes that are executing.
- If a token is present on both edges e_s and e_r, both tokens are consumed and the $extras.status$ attribute of each thread frame in the $o.extras.Tflist$ remains unchanged. The $t.extras.\Lambda.eft$ keeps increasing for t nodes that are executing.

It is left to a user to decide how to implement suspending. It can also remain unimplemented in which case suspending is not supported and has no visible effect.

3.3.4 Inhibitor arc

An inhibitor arc $e(type = inhibitor)$ is directed from a place $p = e.tail$ to a control port $c = e.head(type = entry)$. If place p contains one or more tokens, the inhibitor arc inhibits the enabling of the node with control port c.

If place p contains no tokens, the inhibitor arc e which connects the place p to entry control port c no longer inhibits the enabling of the node. The node then sees the inhibitor arc as control flow edge with as many tokens on it as indicated in the entry control port's weight $c.weight$.

The inhibitor arc thus senses the absence of a token.

3.4 MTG* intra task model

Inside the thread frames the functionality is detailed using a hierarchical control data flow graph. In MTG* this model is the decorated signal flow graph, described in detail in [130].

It is stored $o(type = threadframe).cdfgmodel.dsfg$ attribute.

3.5 Gray box model

The gray box model consists of the MTG* model and the two additional concepts. At the task level the concept of thread cluster is defined. At the intra task level the concept of thread node is defined.

In TCM, thread frames are annotated with Pareto curves. The gray box model uses the $o(type = threadframe).extras.annotation$ field to store such curves which are extracted by the TCM design-time scheduler.

3.5.1 Thread cluster

A new type of node is added to the task level MTG* nodes: $o(type = threadcluster)$. A thread cluster node has the following attributes:

$$o(type = threadcluster).extras = (Nodelist, annotation)$$

The TCM run-time scheduler considers the thread cluster as one unit for scheduling and expects to see a single Pareto curve. Such a Pareto curve which results from combining the separate Pareto curves in the thread frames inside the thread cluster can be stored in the $o(type = threadcluster).extras.annotation$ field.

3.5.2 Thread node

The control and data flow nodes in the gray box model can be clustered in thread nodes.

This leads to 2 new nodes

$$o(type = threadnode)$$

and

$$o(type = hierarchicalthreadnode)$$

. Thread nodes have the following attributes:

$o(type \in threadnode, hierarchicalthreadnode).extras = (Nodelist, annotation)$

The $o(type \in threadnode, hierarchicalthreadnode).extras.annotation$ field can be used for storing energy and execution time estimates or measurements for use in the TCM design-time scheduler. The Nodelist attribute of a thread node can contain cdfg nodes and edges.

Hierarchical thread nodes can contain everything thread nodes can contain as well as other hierarchical thread nodes. The granularity of thread nodes (in terms of cdfg nodes and edges inside them) is determined by what step of the design flow is being done. Thread node boundaries are defined in such a way that they can remain unaltered while transforming (transformation thread nodes) or design-time scheduling (scheduler thread nodes). Information that is not relevant at a certain moment is hidden inside the thread nodes.

Whenever a TCM transformation or other TCM step is defined on the gray box model a suitable thread node boundary must be defined for use in the step.

The semaphore removal transformation described in chapter 5 does not consider thread nodes because it operates at the thread frame level. Thread nodes only exist inside thread frames. In the semaphore removal transformation information is hidden at the task level while extracting a synchronization chart.

For the recursion removal transformation described in chapter 6, the thread node boundaries are defined by giving an algorithm schema in which to rewrite the algorithm. Such thread nodes are called transformation thread nodes.

The TCM design-time scheduler operates on scheduler thread nodes. One scheduler thread node should not have a large variation in execution times over different invocations. Different scheduler thread nodes also should not show a large variation in execution times - their load must be as balanced as possible. Initial boundaries of scheduler thread nodes are defined by visualizing (= not putting them inside a thread node) cdfg-constructs which cause large differences in execution time over different invocations. The most important constructs which are candidates for defining a scheduler thread node boundary are data dependent conditions and non-manifest loop bounds. The scheduler thread node graph subsequently is improved by TCM transformations that increase the available parallelism and the thread node balancing.

3.6 Timing constraints

Timing constraints specify restrictions that must be obeyed by the final implementation. The timing constraints are summarized from [196] which also explains how to adapt the firing rules to keep the token flow consistent with timing constraints.

3.6.1 Timing constraints

3.6.1.1 Timing constraint edges

A timing constraint edge e_t starts in a control port $c_s(type \in \{entry, exit\}) \in M.O.Controlportlist$ and ends in a control port $c_e(type \in \{entry, exit\}) \in M.O.Controlportlist$. Note that both $e_t.head$ and $e_t.tail$ can be a controlport of type entry or exit.

The two control ports $e_t.head$ and $e_t.tail$ belong to respective nodes o_i and o_j. If $e_t.head(type = entry)$ then $t^{s/e}(o_i)$ is the starting time of o_i. If $e_t.head(type = exit)$, then $t^{s/e}(o_i)$ is the ending time of o_i. $t^{s/e}(o_j)$ can be defined similarly. A timing constraint edge e_t from node o_i to node o_j also has a weight $w_{ij} =$

$e_t.extras.value$.

A timing constraint edge then specifies a constraint of the following form:

$$t^{s/e}(o_j) \geq t^{s/e}(o_i) + w_{ij}$$

For $w_{ij} \in \mathbb{R}^+$, the timing constraint edge e_t specifies a minimum latency constraint:

$$t^{s/e}(o_j) - t^{s/e}(o_i) \geq |w_{ij}|$$

For $w_{ij} \in \mathbb{R}^-$, the timing constraint edge e_t specifies a maximum latency constraint:

$$t^{s/e}(o_i) - t^{s/e}(o_j) \leq |w_{ij}|$$

Loops on the same entry or exit control port of a single node o_i are not allowed: $e_t(type = timingconstraint).head \neq e_t(type = timingconstraint).tail$

In a gray box model, timing constraint edges can also be specified between thread nodes. For this reason thread nodes can be equipped with entry and exit control ports and connected. In a simulator, a given thread node configuration can be simulated with similar operational semantics as thread frames. Decisions about exploitation of available parallelism made by the TCM design-time scheduler can then can be represented by adding suitable control flow edges. Timing constraints can be specified with timing constraint edges.

Latency constraints starting or arriving in the exit control port of an event node thus express minimum and maximum response time constraints.

3.6.1.2 Execution rate constraints

These constraints constrain the execution of the same operation in consecutive time instantiations of an MTG (sub)graph. They are specified as intervals. Suppose o is a node with an $o.extras.rtcs$ attribute. Call $R(o,t)$ the instantaneous execution rate of node o.

$$R(o,t) = lim_{\Delta t \to 0} \frac{\Delta n}{\Delta t}$$

, the marginal number of executions n per time of node o. Call $R^{av}(o)$ the average execution rate of node o.

$$R^{av}(o) = \frac{\int_{t_s}^{t_e} R(o,t)\,\mathrm{d}t}{t_e - t_s}$$

$$o.extras.rtcs.min\ inst\ rate \leq R(o,t) \qquad \leq o.extras.max\ inst\ rate$$

$$o.extras.rtcs.min\ avg\ rate \leq R^{av}(o) \qquad \leq o.extras.max\ avg\ rate$$

If not specified,

$$o.extras.rtcs.min\ inst\ rate = 0$$
$$o.extras.rtcs.min\ avg\ rate = 0$$
$$o.extras.rtcs.max\ inst\ rate = +\infty$$
$$o.extras.rtcs.max\ avg\ rate = +\infty$$

3.6.2 Timing guarantees

Timing guarantees represent knowledge about timing of interactions with outside the model.

3.6.2.1 Events

Events can be periodic or aperiodic. Events themselves take no time, their start time is equal to their end time. But the start time itself is non-deterministic, determined by the environment. Call $t^k(o)$ the start time of the k-th firing of event $o(type = event)$. Then a periodic event is an event for which the difference in successive starting times is a constant:

$$t^{k+1}(o) - t^k(o) = o.extras.period,\ with\ o.extras.period \in \mathbb{R}$$

An aperiodic event is an event for which a constant T^{arr}_{min} can be found such that:

$$t^{k+1}(o) - t^k(o) \geq T^{arr}_{min}$$

The largest such T^{arr}_{min} is called the minimum inter-arrival time of the event o. The minimum inter-arrival time of event o can be specified in the $o.extras.t^{arr}_{min}$ attribute.

In practice there can be some deviation (jitter ϕ) on the exact starting times, even for periodic events. For this reason the definition of periodic event with jitter ϕ is given. A periodic event with period 'period' and jitter 'ϕ' is an event $o(type = event)$ for which for which an as large as possible constant $period \in \mathbb{R}^+$ and an as small as possible constant $\phi \in \mathbb{R}^+$ can be found such that $\phi < period$ and:

$$period - \phi \leq t^{k+1}(o) - t^k(o) \leq period + \phi$$

3.6.2.2 Latency constraints

Timing constraint edges specified with respect to event nodes and synchro nodes can specify timing guarantees as well.

3.6.3 Functional timing: delay generation

A control flow edge $e(type = controlflow)$ has an attribute $e.extras.weight$. This weight attribute can be used to specify a delay which must be generated. Specifying a control flow edge $e(type = controlflow)$ with weight $w > 0$ which connects exit control port cp_i from node o_i to entry control port cp_j from node o_j is equivalent to specifying two timing constraint edges:

$$\begin{aligned} e_1(type = timingconstraint).tail &= cp_i \\ e_1(type = timingconstraint).head &= cp_j \\ e_1(type = timingconstraint).extras.weight &= w \end{aligned}$$

$$\begin{aligned} e_2(type = timingconstraint).tail &= cp_j \\ e_2(type = timingconstraint).head &= cp_i \\ e_2(type = timingconstraint).extras.weight &= -w \end{aligned}$$

3.7 Parallelism

The gray box model allows to distinguish several types of parallelism in an application:

task level parallelism: Task level parallelism is visible on the task level MTG* model. During the completion phase of a node's execution, each outgoing edge for which the guardexpressions were evaluated to be true get a token. Each of these tokens can activate other thread frames showing task level parallelism.

thread node parallelism: Thread node parallelism (especially useful for the TCM design-time scheduler) is visible on the gray box model. A particular choice of parallelism can be indicated by adding control flow edges between thread nodes, similar to how it is done for thread frames. Examples include: data parallelism (often loop-level parallelism) and functional parallelism which results from applying slicing criteria.

instruction and data level parallelism: Instruction level and data level parallelism are visible in the MTG* intra task level.

3.8 Future requirements

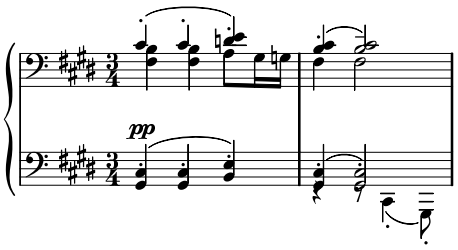
The model provides a hook in the form of the *annotation* attribute of nodes to extend the model for future specification requirements.

Part III

Source code transformations

Chapter 4

Task-level transformations: problem formulation and related work



Claude Debussy, Pour le piano (part 2, Sarabande)

4.1 Problem formulation

4.1.1 What is a task level transformation?

Thread frame boundaries indicate the places where the run-time scheduler might need to be activated. To minimize overhead at run-time the smallest possible number of thread frame boundaries is defined. This means that the boundaries are defined only at those locations where they are really necessary.

Thread frame boundaries occur at source code constructs that introduce run-time behavior which cannot be suitably characterized at design-time. Such places include event handling routines (randomly activated by tokens from the environment), dynamic task creation and deletion, non-determinism and synchronization (e.g. also locking/releasing semaphores). Non-manifest conditions and loops do not constitute reasons to introduce a thread frame boundary: it would introduce too fine-grained a task level model and TCM provides a number of techniques to explore the effects of such constructs at design-time (see the discussion about scenarios and sub-scenarios in appendix C).

The part of the model showing exactly these thread frame boundaries and modeling the activation and completion of the thread frames is called the MTG* task level. Task level TCM transformations therefore are transformations that change something to the MTG* task level model, either by inserting or removing synchronization, or by inserting or removing dynamic task creation combined with splitting and regrouping of existing thread frames to get the new thread frame boundaries.

After extraction of an initial gray box model or after improving of the gray box model by transforming it to reveal additional parallelism, thread frames contain a thread node graph. The task level transformations operate at the level of granularity of such thread nodes. If the scope of a semaphore is changed e.g. this is done by including or excluding a complete thread node. Task level transformations do not look inside thread nodes.

This way the task level transformations are actually performed on the gray box model as opposed to doing them on the fully detailed intra task MTG* level. The advantage to doing transformations this way is the reduced complexity of the information to be taken into account. On the other hand the freedom to do transformations is limited to the granularity of the thread nodes. Part of future research should focus on characterizing the trade-off between transformation freedom, complexity of the exploration and obtainable cost savings.

Transformations that change more than the task level model are called CDFG transformations. They are mainly needed while extracting thread nodes, e.g. to increase parallelism inside the thread frames. They create freedom for task level transformations and for the TCM design-time scheduler.

All these transformations are platform independent because they are needed for any platform instantiation. The actual platform dependent steps are performed in the 2-phase scheduling sub stage. Indeed, the thread nodes for each thread frame are mapped on the platform in different ways by the design-time scheduler[1] to get (energy consumption, execution time) trade-offs. Pareto-optimal trade-offs are visualized using Pareto curves.

[1] In the future also partly at run-time, see appendix C

4.1.2 CDFG transformations enable task level transformations

CDFG transformations are done to expose parallelism in the specification. If the parallelism inside the CDFG is very regular (e.g. synchronized parallel iterations of loops) the exposed parallelism increases the freedom during the design-time scheduling step: more parallelism can be exploited without the need to introduce synchronization that results in long idle times. As an example the separate instances of a loop iteration can be mapped to different combinations of number of processors and number of (discrete) voltages for each processor to create different trade-offs in execution time and energy consumption.

The QoS policies and timing constraints in the system will eventually make sure that one specific option gets chosen at run-time. Without a global system timing constraint an unlimited amount of time would be available to calculate the results so they would not have to be calculated at all thereby consuming 0.0 joule. An example of a global system timing constraint would be a desired frame rate in a 3D rendering application.

If the parallelism is less regular — this means a large difference in execution time between the parallel paths exists — several options exist. A first possibility is to split the parallel paths in separate thread frames. This postpones the decision of how to exploit the parallelism on the platform to run-time. A second possibility is to insert synchronization points and make the parallelism more regular. Inserting synchronization, however, will put some of the processors in a waiting state. It depends on the sophistication of the run-time scheduler and the irregularity of the execution times which solution is best; an energy consumption and execution time model of the run-time scheduler component is required to evaluate the trade-off.

For a data-dependent amount of parallelism (e.g. exploiting inter scene object parallelism while rendering a scene with a data dependent number of scene objects) again a possibility to postpone the mapping to run-time or alternatively to handle it at design-time exists.

For handling such a situation at design-time a parametrized design-time schedule Pareto curve can be generated. It can be parametrized according to the number of processors available, to the number of objects in the scene and possibly also the weights (execution times) of the objects and distribution of the weights over the objects.

Postponing the decisions to run-time results in having to split the handling of each (cluster of) scene object(s) in a separate MTG* task and having to leave the decision of working points to the run-time scheduler[2]. In the case of a

[2]In the future splitting of thread frames for such reasons will no longer be done. Instead

parametrized Pareto curve the run-time scheduler needs a way to decide which Pareto curve to use. This requires some statistics to be tracked at run-time (namely statistics related to the parameters used in parametrize the Pareto curves) and hence might be less power efficient than deciding everything at run-time. Some of the issues that need to be considered: is the extra information that needs to be tracked available for free or does it require extra resources? How is the behavior with respect to power consumption/execution time of the run-time scheduling component?

4.2 Related work in program transformation

It is again not the intention to give a comprehensive overview of the vast amount of related work that exists currently on program transformations. Instead, the focus is on the transformations most directly related to the two classes that are studied in this thesis. In addition, the embedded (hardware/software codesign) related transformations are looked at because they are in many cases also driven by similar costs as our more system-level transformations.

4.2.1 Program analysis and transformation

The topic of program analysis and transformation is a broad and well-studied subject. Approaches based on formal methods [15, 163] as well as more ad hoc approaches (see e.g. [185, 64, 122]) exist and are in use. Program transformations as are used inside optimizing compilers [156] are correct by construction. Most compiler transformations occur at too low a level of abstraction for direct use in TCM.

4.2.2 Program transformations in hardware/software codesign context

Traditionally imperative programming languages are used to implement specifications of hardware/software codesign systems. Imperative programming languages often seem to be easier for specifying large and complex systems but on the other hand have a concept of explicit state which can result in side-effects that make them more difficult to automatically analyze and understand. For this reason other approaches have been considered including functional and logic programming languages or languages directly implementing some formal calculus.

---

thread frames will be able to shift operating points while running. See the discussion in appendix C.

4.2.2.1 HW/SW codesign based on imperative programming

- In the context of digital signal processing (DSP) systems many algebraic transformations [57, 162] have been described to derive efficient hardware implementations from given recurrence equations. Such transformations include pipelining, retiming, modifying computation order, folding, unfolding, strength-reduction and look-ahead transformations.

- Early work [210] defines transformations at a higher level of abstraction than was the case until then. The work does not not consider functional block, gate or circuit level but rather defines algorithmic level transformations. The notion of algorithmic level used in [210] is a combination of algorithmic level, concurrent task level and concurrent data level in TCM. One of the transformations described is pipestaging a design.

- The CAMAD system [165] starts from a PASCAL-like imperative language called Algorithmic Design Description Language (ADDL) and maps it into an extended Petri net formalism (ETPN). The Petri net notation captures control flow information. Data intensive parts of the design are modeled with a data path. On this description a number of transformations are performed. Operation scheduling transformations analyze available instruction level parallelism, decide on how to divide or group operations in time steps and change the order of operations. Furthermore a number of data path oriented transformations and control flow transformations are defined as well.

- The COSMOS system [201] takes an SDL model as input and maps it to C and VHDL. C is used to describe software parts and VHDL describes hardware parts of the final implementation. Transformations in the COSMOS system are user guided and influence functional decomposition (partitioning), allocation of functionality to processors, mapping of high-level communication primitives into implementations and architecture generation. Transformations are implemented on top of a kernel of primitive transformations like *split*, *merge*, *move*, *flatten* and *map*.

- At Stanford the Olympus system [58] takes HardwareC as input language. HardwareC has a C-like syntax, but provides cycle-based semantics and supports concurrency and timing constraints. A number of user-driven and automatic transformations are defined. User driven transformations include selective inlining and selective mapping of operators to library elements. Automatic transformations are standard compiler transformations and typical hardware/software transformations. The standard compiler transformations include for-loop unrolling, constant and value propagation, dead code elimination and common subexpression elimination. Typical hardware/software codesign transformations include collapsing of conditionals for branches containing only combinational logic to in-

crease the application scope of logic synthesis tools. Data flow analysis is performed to identify data and control dependencies.

- In [79] a series of code motion techniques is presented for synthesis of high-performance microprocessor functional blocks. Example transformations include reverse speculation and early condition execution. The results are improved by combining the code motion techniques with loop transformations. The transformations are implemented in the SPARK transformation system [78].

Whereas the approaches mentioned up to now combine optimization with lowering the level of abstraction (similar to compilers) also source-to-source optimizations are being done. Such optimizations take a description in one language and output an optimized description in the same language. In hardware/software codesign source-to-source transformations are known as pre-synthesis transformations [137, 138, 125, 164]. Loop transformations tailored to the needs of hardware/software codesign [72, 74, 41, 22, 108, 172, 229, 38] can be seen as a kind of pre-synthesis transformations.

Source-to-source transformations in hardware/software codesign are also used to optimize complex addressing [152, 80].

4.2.2.2 HW/SW codesign based on functional programming

Hardware/software codesign transformation systems based on functional programming include the digital design derivation system (DDD) [27], the FORM language [43, 44, 45], the STRIDE tool [33, 124, 32], ForSyDe [179, 219] and the SILAGE language [70].

- In DDD specifications are written in a purely functional subset of the programming language Scheme. DDD distinguishes between behavioral specifications and structural specifications and defines a way of going from behavioral to structural specification. Transformations can be done on both types of specification.
- The FORM language is a functional programming language with some constructs that are needed to capture codesign specifications. Transformations include recursion removal, conditional resource sharing, multiplexer reduction, parallelization, serialization and loop unfolding.
- In STRIDE a functional programming language called STRICT (similar to VHDL but using a functional paradigm) is taken as input language. Only transformations that are correct by construction can be applied to the design. Such transformations are presented as a series of rewriting rules. In ForSyDe a systematic refinement methodology is developed

based on transformations which gradually transform a high-level, function oriented system description into a synthesizable model.

- The SILAGE language serves as input to HYPER [40, 39] which implements transformations like retiming, loop unrolling and software pipelining.

4.2.2.3 HW/SW codesign based on logic programming

Approaches to hardware/software codesign transformations based on logic programming languages have been proposed as well [131, 184]. In [211] the semantics for a subset of VHDL was formalized in higher-order logic to enable correctness proofs of transformations on this VHDL subset.

4.2.2.4 HW/SW codesign based on underlying formal models

Transformations in hardware/codesign context are also often done on a language implementing an underlying formal model. The underlying model makes it easier to be certain about the correctness of a given transformation and provides a means for formal calculation of the effects of a transformation.

- In [14] the problem of partitioning a design described in a subset of OCCAM (derived from CSP) in hardware and software parts is tackled.
- In [147, 148] the TRADES system is described. It allows for designer-driver small and local behavior-preserving transformations like refinement transformations and space-time transformations (to make an area/delay trade-off). It takes the SPRITE Input Language (SIL) as input and generates VHDL. SIL is a CDFG description language.
- An approach based on action systems is presented in [171].
- An approach based on Timed LOTOS (implementing a process calculus) is followed in the FORMAT project [76]. FORMAT takes T-LOTOS as input and generates VHDL. Formal transformations can be selected by the designer in order to restructure the design.
- The SHE method [208, 202] operates on the POOSL language which itself is using concepts from UML.
- The SILAGE language serves as input to SynGuide [178] which implements loop transformations using a geometric polyhedral model for reducing memory requirements and execution cycles.
- The semantics of VHDL were formalized by [161] to enable reasoning about correctness of transformations on VHDL.

The program transformations that are considered in detail in this part of the thesis are semaphore removal and recursion removal. Related work for these transformations will be looked at in more detail in the next two sections.

4.2.3 Semaphore transformations

With respect to semaphore transformations three different approaches can be distinguished.

- A first approach looks at when and how to use synchronization. This first approach is concerned with minimizing the number of locks and changing their scope. Related work to this first approach is considered in more detail in the rest of this section. Our work from chapter 5 also belongs here.
- A second approach is concerned with finding efficient ways of implementing synchronization primitives [73, 11, 126, 2, 158, 63]. Results of research into this approach are fully complementary to results into the first approach.
- A third approach looks at how the execution environment can help lock-based programs to execute in a lock-free manner [173, 174]. This approach is also complementary to the first approach.

Synchronization operations are a source of overhead. In [3] it is reported that even in single-threaded JAVA applications synchronization operations can consume 10% to 50% of the execution time. The reason for this is JAVA's usage of monitors [105] which are simple to understand and use but expensive in terms of execution time. MTG* semaphores can be used as monitors. In [185] it is noted that adding synchronization to a JAVA method degrades that method's performance with a factor 9 to 50 even when running in a single thread. It is more expensive to check the availability of a lock in JAVA than to acquire it. Synchronization overhead looks worse when traditional compiler optimizations improve. Because of the large influence of the quality of the compiler optimizations it is not trivial to quantify synchronization overhead.

In [3] three transformations are proposed that remove unnecessary synchronization.

- The first transformation removes synchronization in code that is executed by at most one thread. A similar transformation is described in [26] which removes locks protecting data structures that are reachable by at most one thread. A drawback of this transformation is that the code in general can no longer be reused in other software projects. In the context of TCM that is not an issue.

- The second transformation that is proposed removes locks on data structures that are protected by enclosing locks. If different locks in enclosing scopes protect the same data all but one can be removed.
- A third transformation is to remove reentrant synchronization. Reentrant synchronization occurs when the same lock is being acquired several times in nested fashion (e.g. inside a recursive function call).

These three transformations remove unneeded synchronization. The transformations are claimed to be correct by construction and automatable.

In [185] synchronization is removed by manually restructuring code. Typical transformations include removing locks if they are not needed in the current program or making specialized versions of synchronized methods to provide fast entry points for threads already holding other locks that protect the same data. When a program is changed later on such transformations can be the cause of subtle bugs.

Enlarging the scope of a lock can decrease the synchronization overhead at the expense of parallelizability. In [61] two forms of this transformation are proposed: data lock coarsening and computation lock coarsening.

- Data lock coarsening is a transformation to protect with a single lock multiple data structures that are usually accessed together.
- Computation lock coarsening is a transformation that lifts acquiring and releasing of a lock outside a loop. A similar technique is described in [170]. They use the concept of access regions to optimize synchronization overhead in sequential executions of the same program. An access region is a piece of code that has exclusive or non-exclusive access to an object. When an access region is entered all its locks are locked and they are only released upon leaving the access region. Access regions are made as large as possible to minimize the time spent in locking and releasing. No trade-off between locking overhead and waiting for a lock to become available is considered.

Other types of synchronization (notably barrier synchronization) are topics of research in e.g. the high-performance computing community [42].

- In [149] techniques are described to remove synchronization in loops with loop-carried dependencies. Such techniques include loop transformations. The loop transformations are done without consideration for data management.
- Barrier synchronization is eliminated where possible in [198, 84]. In some cases barrier synchronization can be removed entirely. An example of this

occurs when data dependence analysis can prove that the parallel loop instances access disjoint data. In other cases barriers are replaced with implementations based on counters.

- Based on data dependence analysis [89] finds the minimum number of barriers needed.
- In addition to minimizing the amount of barrier synchronization [168] also minimizes the amount of intermediate storage needed.

The semaphore transformation defined in chapter 5 belongs to the class of semaphore transformations that look at when to use semaphores. In contrast to the existing approaches, the transformation is defined on top of the gray box model. Other research about synchronization and semaphores in particular is fully complementary to our technique. Once a gray box model is available automated application of the technique in chapter 5 becomes possible.

4.2.4 Recursion removal transformations

Recursion removal is a well studied subject. Many results are available both in theory and practice. One remarkable result is that any recursive function that ends in a finite amount of time can be implemented in an iterative way by insertion of a stack data structure.

This is not the only way to remove recursion and replace it with iteration though. Several iterative versions of recursive functions can be found which behave differently with respect to memory and time requirements as compared to the recursive function. Insertion of a stack also does not provide insights into how the recursive algorithm is evaluated: the run-time stack mechanism which is automatically generated by the compiler is just replaced by a hand-optimized stack implementation resulting in a simulation of the recursive control flow.

Other well-known results include that tail recursion is equivalent to a loopand non-tail recursion can be rearranged to become tail recursion [214, 15]. Similarly non-linear recursion can sometimes be linearized [87].

Application of the technique from chapter 6 on tail recursion will essentially result in the same loop as does application of tail recursion removal. Extensions of existing recursion removal methods also exist for removal of mutual recursion. An approach for removal of mutual recursion consists of inlining some of the functions into the other functions resulting in direct recursion which can then be further optimized using recursion removal techniques for simple recursion [163, 120] including the one recursion removal technique described in chapter 6 of this thesis.

Many results about recursion removal in literature are based on recognizing

patterns in the source code which can be proved, using a myriad of techniques, to be equivalent to certain iterative patterns. These patterns are known as recursion schemes [15, 163]. The technique developed in this thesis will also start by fitting the code in a certain pattern. Compared to other schemas our schema is more general and flexible. The schema defines certain model elements which are used in the formal derivation of the method.

Many results in recursion removal were derived in the context of optimizing compilers for functional programming languages.

- In 1962 John McCarthy implemented tail recursion removal in his LISP compiler [145].
- Later John Backus worked on formal transformations on his FP programming language [9, 10].
- In [86] linear recursive forms are translated to loops.
- Transforming a program to continuation passing style [66] has applications in optimizing compilers for functional languages [121]. It has the effect of transforming any function call to a tail call. This means that functions never have to return and can theoretically be implemented as a simple jump instruction. For this reason the programming style is sometimes called "goto with arguments". A function never returns but instead calls a new function (the continuation [65, 175]). As a result any recursion becomes tail recursion and can be replaced by iteration. The need to build functions at run-time that are passed around as parameters of other functions in continuation passing style is a source of overhead. Means to address such overhead have been described in [151] and further examined in [187].
- In [88] recursion is transformed to iteration using stacks that are compiled as vectors. His techniques work in the presence of side-effects but are only suitable for linear recursion.
- In [48] algebraic properties are identified to help predicting redundancy patterns and corresponding tabulation schemes. Using these properties it becomes possible to eliminate recursive calls.
- In the context of action systems, [212, 213] proves a general recursion removal theorem. He uses it in a reverse engineering context to transform iterative functions back to recursive ones because they are supposedly easier to understand.

In functional languages, however, some assumptions can be made which are not true for imperative languages [49]. One typical assumption is the lack of

side-effects as a result of modifying state since pure functional programming languages have no concept of state [177].

In [34] transformation and simplification of recursive programs using techniques like "unfolding" and "folding" is discussed. Unfolding in their terminology means symbolically evaluating the program. Folding means recognizing patterns and abstracting functionality in recursive function calls. Their techniques can lead to iterative versions of recursive algorithms [53] but no systematic way of working is formulated. More systematic approaches based on the same ideas are summarized in [163].

More recent work [140] has concentrated on removing recursion in general situations and providing measures for the effects in terms of time and space complexity. Their basic idea is to identify a so-called generalized function increment and then use a loop to calculate the result using the increment. The method is very general and can guarantee improvements in both time and memory requirements. In the context of TCM which is concerned with program transformations to enable parallelization for multi-processor embedded platforms their method has a potential draw-back: it introduces explicit loop iteration dependencies. Indeed, before the next iteration can be evaluated the previous one needs to be completed.

For special forms of recursion specialized methods have been devised. Recurrence equations e.g. can be solved mathematically using techniques like repeated substitution while looking for a generalized pattern or guessing the form of a solution and proving it correct by induction [176]. For special forms of (linear) recurrences it is possible to use the method of generating functions [75]. For some divide-and-conquer recurrence equations it is possible to get an idea of the complexity of the solution using the so-called master theorem [17, 50]. Systematic methods exist to solve linear homogeneous recurrence relations with constant coefficients and to solve certain non-homogeneous recurrence relations with constant coefficients [176]. Recurrence equations often occur in digital signal processing (DSP) applications (FIR and IIR filters). Typically they cannot be solved mathematically. In such cases, different implementations of a given recurrence equation can be made by applying algebraic unfolding and look-ahead transformations [162]. Look-ahead transformations reshape the equation to increase the available parallelism in an implementation.

Our work addresses non-linear recursion in strict imperative languages that can be rewritten in a certain schema. This schema defines the thread node boundaries on which the transformation operates. The recursive algorithm can contain code with side-effects and systematically finds an iterative version that has no explicit iteration dependencies caused by the recursion removal method.

While transforming from recursion to iteration our method allows for several systematic trade offs to be made.

- A first systematic trade off can be made between the number of calculations that must be done versus the amount of memory that will be used.
- A second systematic trade off can be made between the amount of memory needed and the available parallelism.

These iterative versions of recursive algorithms are expected to be beneficial in the context of task level parallelization of algorithms.

4.3 Pareto curves and surfaces

4.3.1 Design space and design space points

Each thread node in a gray box model can be characterized by a number of cost factors obtained from profiling the thread node on a platform or by using high-level cost estimators. In the simplest case these cost factors are total execution time and total energy consumption. In a more elaborate case more cost factors can be taken into account.

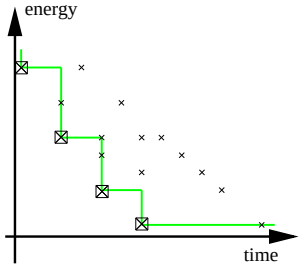
Mapping the thread nodes on the target platform in a variety of ways results in different implementations of a thread frame based on the cost factors. On a platform with two processors each of which can work in a slow but energy efficient mode or a fast but energy consuming mode one can choose to map all thread nodes sequentially on one processor or map some thread nodes in parallel on the two processors. Each mapping results in a certain quality level, user benefit, code size, execution time, data memory power and instruction memory power. Each cost factor can be considered a dimension in a *design space*. One particular mapping can be visualized as one point in this design space. Such a point is called a *design space point*.

4.3.2 Pareto optimality

A design space point $p_i(time_i, energy_i)$ is *not* a Pareto point if another design space point $p_j(time_j, energy_j)$ exists for which both $time_j \leq time_i$ and $energy_j \leq energy_i$. The design space point p_j is said to *dominate* design space point p_i. A special case exists when both $time_j = time_i$ and $energy_j = energy_i$; in that case the design space points are said to be *equal*.

This criterion extends into N dimensions: a point $p_i(c_{i_1}, c_{i_2}, ..., c_{i_N})$ is dominated by another point $p_j(c_{j_1}, c_{j_2}, ..., c_{j_N})$ iff all $c_{i_q} \geq c_{j_q}$, *where* $q = 1..N$. A special case exists when all $c_{i_q} = c_{j_q}$, $q = 1..N$. In that case the points are said to be *equal*.

Figure 4.1 Example point cloud; Pareto points are indicated with squares



Given a number of design space points one can filter out all non-Pareto design space points. The design space points that are left are called the *Pareto points* (see figure 4.1 for an example in two dimensions).

Note that equal design space points not necessarily imply equal implementations or mappings — especially if not all existing costs are accounted for. As a result in general it is not allowed to remove all but one of the equal design space points. For reasons of transformation impact estimation, however, all but one of the equal design space points can be removed because the estimation is high-level. It does not take into account information that is not visible in the Pareto surfaces. Impact estimation takes place on the Pareto surfaces generated by the Pareto optimal design space points. These Pareto surfaces do not depend on the number of equal Pareto-optimal design space points.

4.3.3 Costs versus constraints

A distinction is maintained between the notion of *cost* and *constraint*.

- a *constraint* for our purposes is a Boolean relation, often an equality or inequality relation, between the values of one or more cost factors (often two). Cost factors involved in constraints are called constraint variables. E.g. $t < 3$ ms is a constraint on the constraint variable execution time t. Constraints are typically chosen by designers or imposed by standards. In a video application there may be a need to decode at least 25 frames per second. This will result in constraints on the execution time: decoding a frame must be finished within 40 ms. If two videos are to be decoded at the same time on a single processor decoding a frame must be finished within 20 ms. Constraints on execution time e.g. can change at run-time depending on the workload in the system or as the result of integrated end-to-end quality of service (QoS).
- a *cost* is a cost factor that is not a constraint variable. Its value follows from choosing values for the constraint variables that make the constraints true.

When considering only Pareto points, costs are automatically minimized by just meeting the constraints. Making the constraints more stringent results in larger costs. As an example: changing the constraint on maximum execution time from 3 ms to 2 ms results in a higher energy needed to complete in time.

The designer decides what are considered constraints and costs. In a high-performance computing context power (related to energy) can be a constraint. Power must be kept low enough to not overheat the system. By limiting the power also the maximum speed of sequential computations is limited. Execution time here is a cost resulting from constraining energy consumption.

In a real-time low-power embedded context (for which TCM was developed) execution time is a constraint: each thread frame has to finish in a given time to guarantee timely results. When more tasks run simultaneously on the same platform the constraints on each task separately become more stringent. As a result the energy consumption needs to go up. Energy consumption here is a cost resulting from constraining execution time.

In a context of real-time low-power embedded devices with QoS management the concept of quality can be represented using two different cost factors. The first one is a hard constraint ensuring a minimal quality of service is present in the system. The second cost factor is a quality cost (quality cost $\sim 1/\text{quality}$) that should be minimized together with the other costs as described in section B.2.2.

Because multiple cost factors are taken into account at the same time it does not make sense to try to define a single optimal implementation. Instead different implementations will result in different Pareto optimal design space points. In general the constraints should be chosen as relaxed as possible to get a minimization of the costs.

Examples of constraints in a typical TCM context include execution time budget and minimal video quality level. Examples of costs in a TCM context include energy consumption, code size and quality cost.

4.4 Transformation classification

As proposed in appendix B TCM transformations are thought of as being organized in two steps: a pre-processing step and an exploration step. Both these steps can be further broken down in an abstraction substep, a freedom creation substep, a concurrency analysis and improvement step and a complexity reduction step. During pre-processing transformations are done that do not introduce a system-wide trade-off of two or more costs. During exploration transformations system-wide trade-offs are evaluated.

During pre-processing the source code is transformed in such a way that none of the considered costs become higher. Functional behavior can potentially be changed as long as the expected end results are still the same. Applying this in a TCM context implies that the Pareto curves as derived by the TCM design-time scheduler shift towards the origin. TCM pre-processing transformations were originally defined as transformations that do not globally trade off two or more costs. Another way of putting it: pre-processing transformations perform advanced cleaning up of an initial implementation. Looking at the Pareto curves before and after the pre-processing transformation reveals no intersection between the two. The shape of the Pareto curves before and after pre-processing transformations is not necessarily identical because smaller local trade-offs are allowed as long as they do not result in intersecting Pareto curves. Moreover the transformations often create new possibilities to map the thread nodes on the platform and hence new Pareto points can be created. So a new definition for TCM pre-processing transformations can be given.

A *TCM pre-processing transformation* is a transformation that does not result in a new Pareto curve intersecting with the old Pareto curve.

A second stage in the transformation script is the application of transformations that introduce system-wide trade-offs. A system-wide trade-off involves making the value of one cost factor lower at the expense of the value of another cost factor. In a TCM context this causes the Pareto curve after transformation to be rotated as compared to the Pareto curve before transformation. Comparing the Pareto curves before transformation and after transformation reveals one (or more) intersection points.

A *TCM exploration transformation* is a transformation whose Pareto curves before and after transformation show an intersection. See figure 4.2 on the next page for an illustration.

Note that practical Pareto curves necessarily are discrete step functions because the design-time scheduler can only generate a finite number of different mappings of thread nodes on a real platform.

More details about Pareto curves can be found in appendix B from section B.2 onwards.

4.5 Conclusions

This chapter gave an overview of related work in the area of program transformations in general and semaphore and recursion removal transformations in particular.

In addition some terminology related to TCM transformations and used in the

Figure 4.2 Pre-processing and transformation phases in TCM transformation flow.

Pre-processing transformation:no intersection

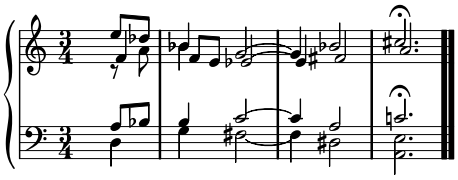
Exploration transformation: intersection

Combined Pareto curves

rest of the thesis was introduced. More discussion about Pareto curves and their relation to transformations is given in appendices B and C.

Chapter 5

Semaphore transformations



Alexander Scriabin, Prelude op.74,no.4

5.1 Introduction

Semaphores are constructs which are used in parallel programs to prevent race conditions from happening. A race condition is an unpredictable (and often wrong) end result of a program due to two separate pieces of code that read and modify the same shared data simultaneously.

Before a piece of code protected by a semaphore is executed it needs to get permission from the semaphore. It tries to acquire (lock) the semaphore. Execution blocks as long as no permission is received. When permission is received, execution takes place. After executing the semaphore is signaled that execution is finished (the semaphore is released). The semaphore now can give permission to another piece of code waiting for a permission to execute. This mechanism ensures that only one piece of code simultaneously can access the shared data.

Because semaphores introduce a thread frame boundary, transformations on semaphores are task level transformations.

Usage of semaphores contributes in two ways to the overall system cost. First, semaphores prevent parallelism. This makes that the code has to execute on less processors which then have to work faster and hence energy consumption increases. The second cost comes from the semaphore implementation and the resulting overhead. Our semaphore transformation does not look at semaphore implementations because they are operating system and platform dependent [1]. Instead certain patterns of how semaphores are used are detected and replaced with other patterns. This is explained in section 5.2.

Semaphores in MTG* are represented using a diamond node with more than one incoming arc and more than one outgoing arc, and with (usually) one initial token.

5.2 Unconditional sequentialization of semaphores

5.2.1 Introduction

We propose to sequentialize the execution of two thread frames guarded by the same semaphore under certain circumstances. If the sequentialized code appears in a loop a possibility for pipelining is created. If the sequentialized code does not appear in a loop at least the synchronization overhead is removed.

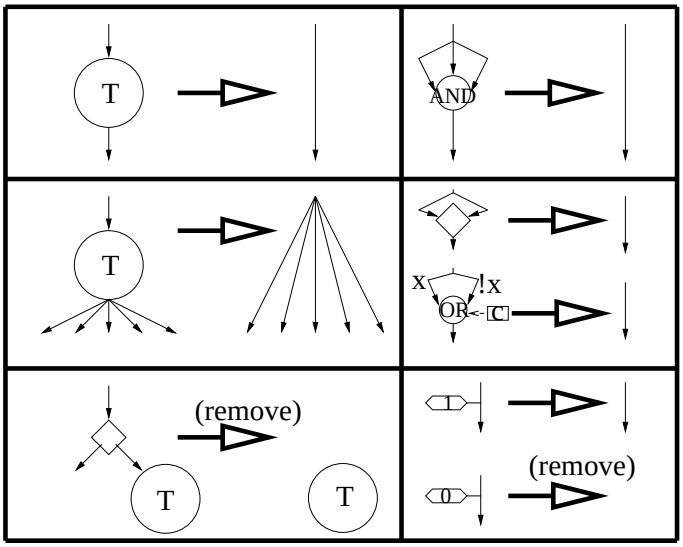
5.2.2 Information hiding and analysis

Before the execution of two thread frames guarded by a semaphore can be sequentialized using the technique described here, it needs to be certain that both thread frames are always both activated. Then the semaphore can be interpreted as implementing a dependency (with additional protection of common resources).

To find out if both thread frames are always activated the task-level model needs to be analyzed. The proposed analysis looks at the tokens which activate the protected thread frames and tries to find out if a common token source exists that will always cause the two thread frames to be activated. As a way to do this it is proposed to reduce the task-level model to a synchronization map by means of elementary graph transformations. The graph transformations reduce the information which is visible at the task-level (without changing anything to the essential token flow). A list of such graph transformations is given in figure 5.1 (this list is not meant to be exhaustive). The graph transformations remove model elements in such a way that the token flow in

[1] For reasons of transformation impact estimation a reasonable semaphore implementation can be assumed.

Figure 5.1 Graph transformations. Edge guard expressions are preserved if not mentioned explicitly.



between the synchronization elements is not changed. When no more graph transformations can be applied one is left with a synchronization map.

After the synchronization map has been constructed it is used to identify patterns of semaphores used as dependencies. The interesting patterns are those that guarantee that whenever a token activates one piece of code guarded by a semaphore S, eventually all pieces of code guarded by S are activated.

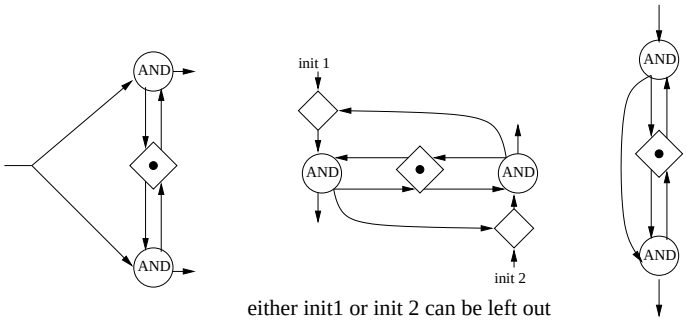
Examples of interesting patterns are listed in figure 5.2. For the situation with two pieces of code being guarded by one semaphore this pattern catalog is probably complete. The patterns can easily be generalized and combined for situations with more than two pieces of code guarded by the same semaphore.

5.2.3 Sequentialization transformation

Given that an interesting pattern on the synchronization map has been identified different sequentialization possibilities are proposed in figures 5.3, 5.4 and 5.5.

Removal of a semaphore by sequentialization in itself does not cause problems with resource conflicts. If the code is later to be pipelined care will have to be taken not to re-introduce resource conflicts then. If the resource being protected is situated in shared memory it is sometimes possible to enlarge the buffer size and usage to remove problems. In some cases it is possible to replace a simple

Figure 5.2 Synchronization map patterns.



buffer with a double buffering mechanism and a barrier synchronization. How to do this systematically and automatically is left for future research. It will most likely require using the PDG model to correctly model and transform data transfer and storage. The results of such a transformation must conform to the constraints generated by the task-level DTSE stage of the application optimization trajectory, which is done before TCM.

5.2.4 Trade-offs

It is clear that preparation for pipelining will introduce a trade-off: memory size versus synchronization overhead and reduced parallelism. After sequentialization the two thread frames are reduced to two thread nodes. Now several possibilities to reduce latency exist (as far as they do not result in violations of global timing constraints).

1. The two thread nodes can first be merged, then sliced or data parallelized and be rewritten to form two (or more) independent (parallel) thread nodes. These thread nodes should be more or less balanced with respect to execution time, and ideally communicate data in one direction only.

2. If (1) is not possible and the code is inside a loop it may be possible to first merge the thread nodes (possibly after partially unrolling (replicating) the loop) and then split them in such a way as to enable pipelining of the loop. In order to hold intermediate results needed for pipelining an increased buffer size may be needed. This is a trade-off.

3. If neither (1) nor (2) are possible it is still possible to merge the thread nodes and try to exploit the increased possibilities for instruction level parallelism.

Figure 5.3 First sequentialization possibility.

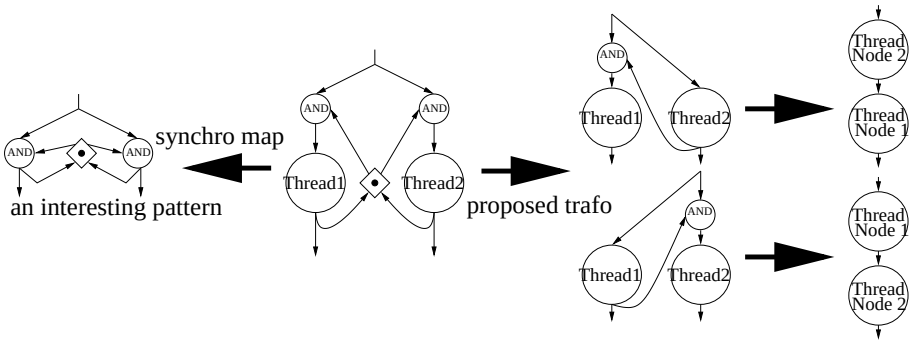


Figure 5.4 Second sequentialization possibility.

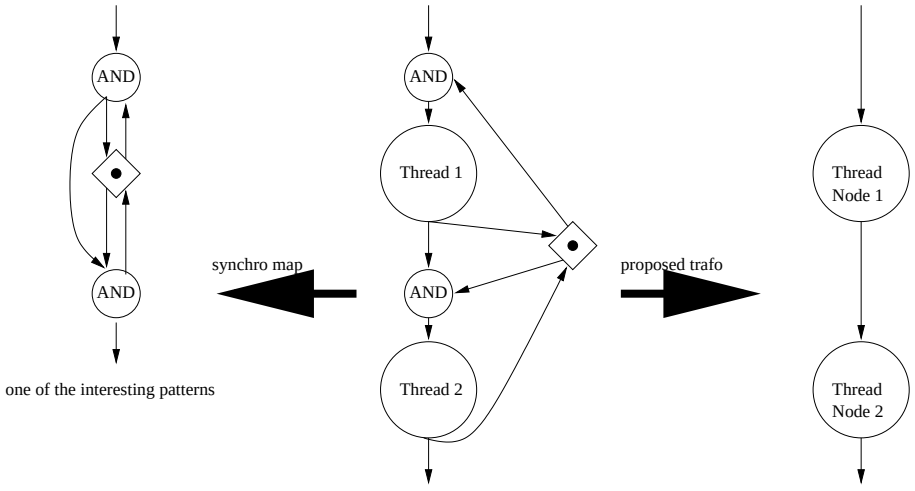
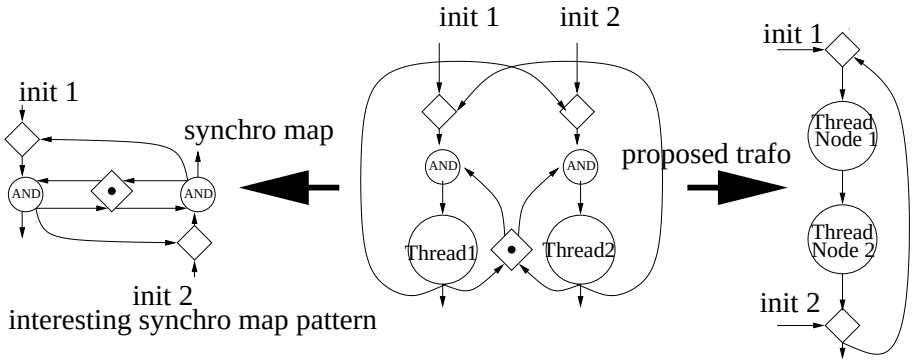


Figure 5.5 Third sequentialization example.



5.3 Example: JPEG decoder

5.3.1 introduction

Figure 5.6 shows the MTG* task-level model of an application that decodes a series of images (slideshow). Each time a new image is available for decoding, an event NewImage injects a token into the model, and a the MTG* "ParserThread" thread frame is activated. The ParserThread thread frame decodes the image header data and creates a DecoderThread for each color channel. When all the color information has been decoded, it is used by a RenderThread. Before it is displayed, a filter operation can be applied to the image. The filter coefficients can be replaced at run-time by the user. This is modeled using an event NewFilter. A third event StopApplication stops the application. The model shown in figure 5.6 was implemented in an MTG* simulator which allows for simulating its execution.

The JPEG decoder in its original implementation did not contain any semaphores. In order to add some dynamism some task creation was added to the application which required semaphores to protect shared memory from being accessed from different threads simultaneously. It is expected that no opportunities to remove semaphores by unconditional sequentialization are found.

The corresponding synchronization map after applying the graph transformations listed in figure 5.1 is shown in figure 5.7.

Figure 5.7 shows three semaphores (= diamond node with more than one incoming and outgoing edges). For semaphore sema_B and semaphore sema_C the situation is identical: one of the token suppliers to the AND1 (AND6) node is a non-deterministically occurring event Event NewImage (Event NewFilter) which is totally unrelated to the tokens coming from AND2 (AND5). Therefore these semaphores cannot be sequentialized with the technique in this chapter: no guarantee exists that both thread frames protected by the same semaphore are always both activated.

For semaphore sema_A, AND3 and AND4 are activated at the same time if AND1 and AND5 are activated at the same time (tracing back the predecessor token suppliers). However, AND1 is activated by a non-deterministically occurring event which has no influence on AND5. Hence also semaphore sema_C cannot be sequentialized.

This corresponds to our expectations.

Figure 5.6 JPEG decoder task level model.

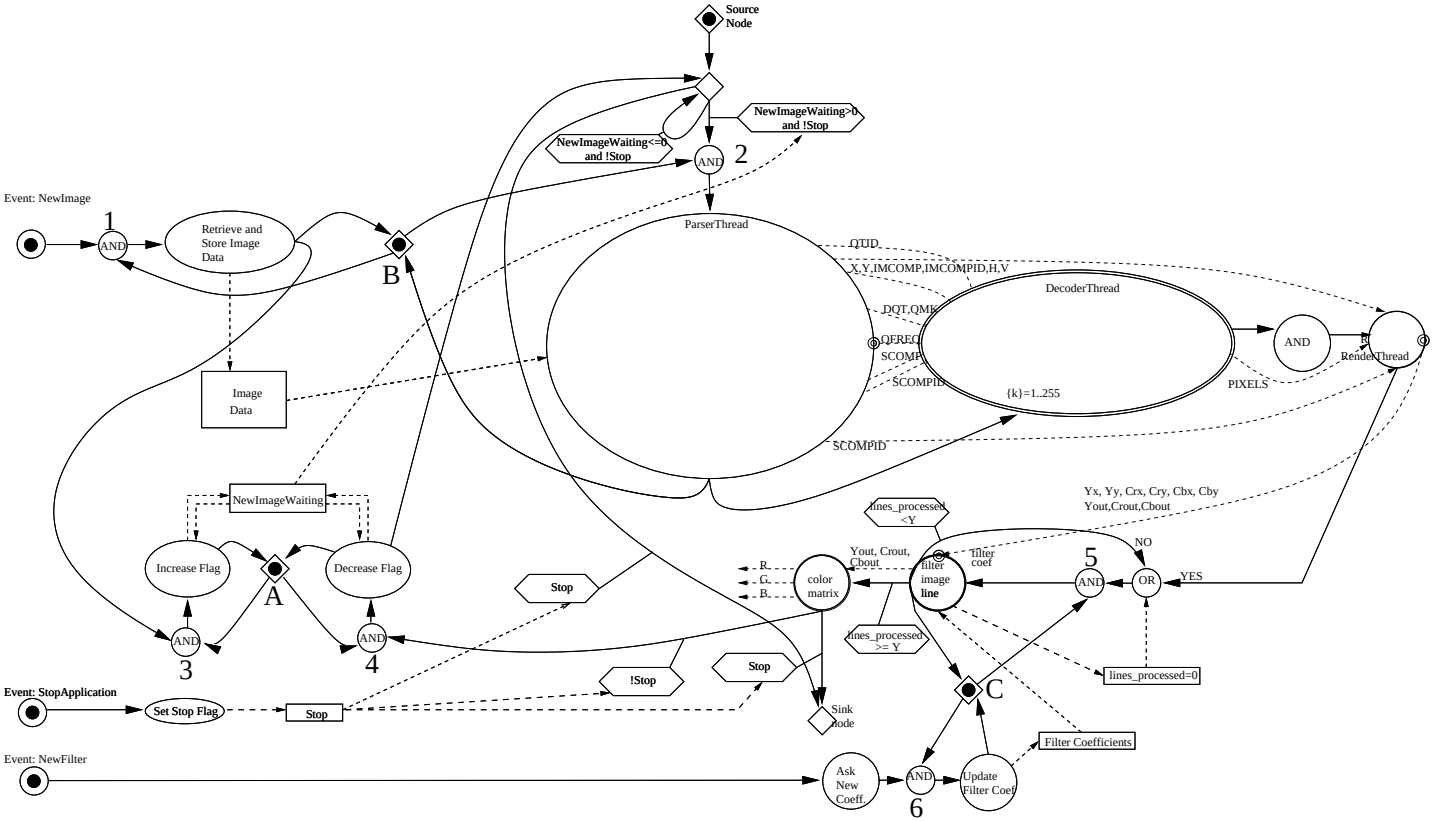
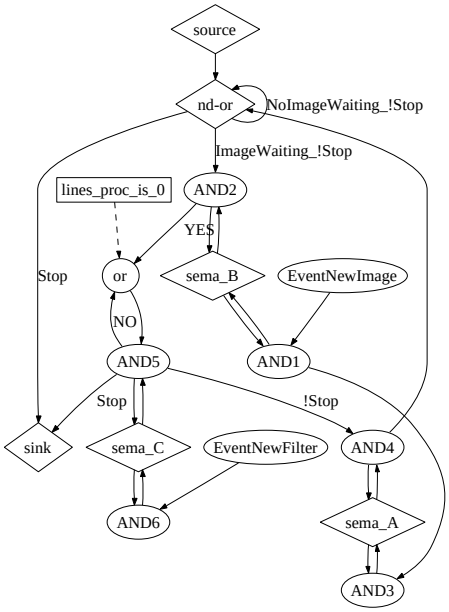


Figure 5.7 JPEG decoder synchronization map.



5.4 Example: JAVA MPEG 1 video decoder

This example is taken from a real-life application and shows that it is possible to remove semaphores from such applications using the technique explained in this chapter.

5.4.1 MPEG 1 bitstream format

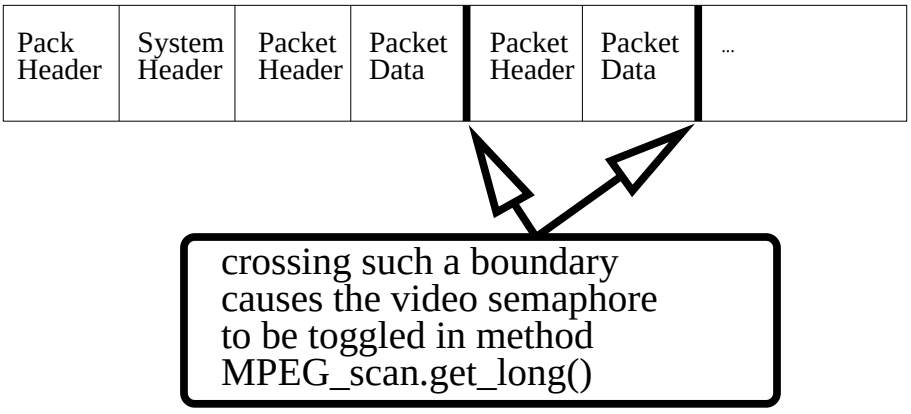
The MPEG 1 bitstream format is described because it dictates the implementation of one particularly difficult to understand part of the application. MPEG 1 bitstreams consist of audio and video bitstreams which are multiplexed to a system bitstream. The system bitstream format is shown in figure 5.8.

The system Data Packets which have the same streamId property[2] must be concatenated together to get a valid Video or Audio stream. The Video stream format is shown in figure 5.9. Each picture header has a property 'set-type' which contains the picture type. The type can be one of three types.

1. I-type: intra-frame. This is a still image; no prediction data needed.

[2]explanation about the meaning of the properties is found in the MPEG 1 standard, part 1. Alternatively one can look at http://www.cs.cornell.edu/dali/overview/mpeg.html, especially the comments in the detailed C specifications.

Figure 5.8 System bitstream format.



2. P-type: predicted frame. This is an image which is predicted from the last P or I frame that occurred with the aid of additional prediction error correction information.

3. B-type: bidirectional frame. This is an image which is predicted from the closest two I or P frames: one in the past and one in the future.

Typically the frames are not multiplexed in correct display time ordering. They are transmitted in a way that allows the decoder to work in a streaming fashion. As such the future I or P-type image which is needed to decode a B-type image is transmitted before the B-type image information itself. Suppose the following pattern denotes the types of images that need to be displayed over time:

display time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
picture type:	I	B	B	P	B	B	P	B	B	P	B	B	I	B	B	...

Then the transmission sequence could be:

display time:	0	-2	-1	3	1	2	6	4	5	9	7	8	12	10	11
transmission time:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
picture type:	I	x	x	P	B	B	P	B	B	P	B	B	I	B	B

Usually pictures are transmitted in a fixed pattern. In the example given the pattern is IBBPBBPBBPBB with a period of 12 frames or about 0.4 seconds; after this time a new I-type image is transmitted and prediction data can be fully reset. This allows to start decoding video in the middle of a stream. Pictures further consist of slices which consist of macroblocks. Macroblocks consist of blocks which consist of pixels.

The audio bitstream format is shown in figure 5.10.

Figure 5.9 Video bitstream format.

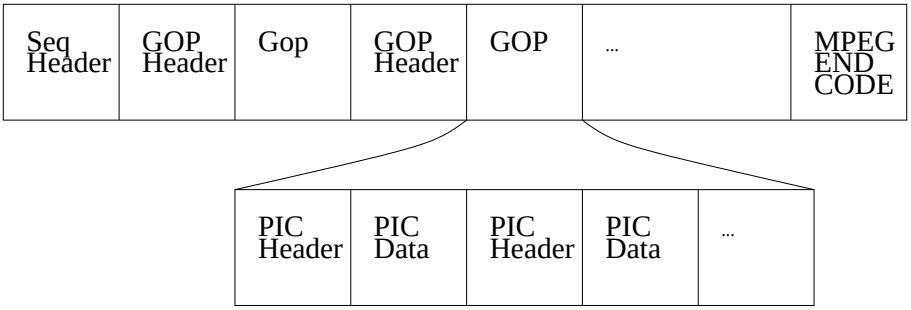
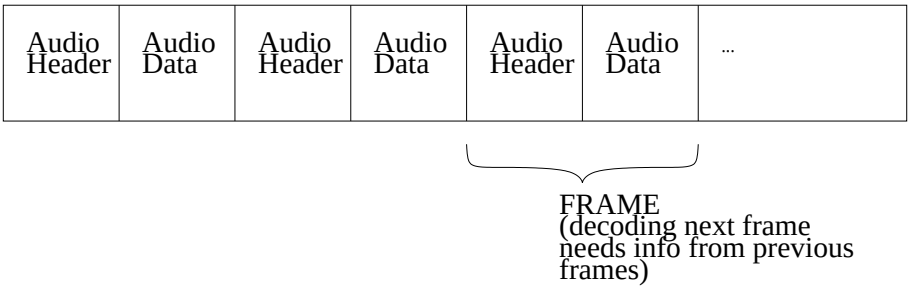


Figure 5.10 Audio bitstream format.



It is important to realize that the boundaries of the packet data are not guaranteed to correspond to boundaries of Video or Audio data packets. This complicates the implementation of the decoder. The implementation in JAVA results in a program giving some difficulties in modeling and transformation. These problems are discussed in section 5.4.3.

5.4.2 Video semaphore

Two versions of the application exist. In the first version (the non-jit version) the whole bitstream is decoded and kept in a buffer of images. Displaying starts as soon as a first image is available. In the second version (the jit version) the bitstream is decoded and displayed one image at a time. The images are not kept in a buffer. The rest of our work is based on the jit version because the non-jit version necessarily runs out of memory for some bitstreams.

The program takes an input stream as parameter. If the input stream consists only of video packets, the video decoder and the renderer are started. If the input stream is a system bitstream consisting of multiplexed audio and video more work is needed. The application then scans the system bitstream, skips audio packets and picks out the video packets.

The video packets need further decoding into images. Scanning the system bitstream is done in a java thread MPEG_scan which uses an object io_tool. Concurrently with MPEG_scan a video decoding java thread which decodes the video packets coming from the bitstream into images is present. The bitstream decoding thread and the video packet decoding thread are implemented as JAVA threads which are mutually exclusively activated. When the video decoder part needs more video packets it activates the MPEG_scan thread and suspends. After MPEG_scan thread has found and retrieved the next video packet it reactivates the video decoding part which continues and eventually suspends again.

Concurrently to both decoding threads a java thread AnimatorThread runs which displays decoded images when they are available. In the non-jit version all decoded images are stored in a circular buffer and when the bitstream is fully decoded replaying starts using the images stored in the buffer. In the jit-version no decoded images are kept in a buffer and replaying requires redecoding the system bitstream.

The bitstream scanner uses an io_tool to do the actual bitstream scanning. The io_tool object contains a method get_long which contains a conditional branch with a call to toggle a semaphore (the video semaphore). This semaphore is used to implement the mutual activation and suspension of the bitstream scanner and the video information decoder. The get_long method is activated in literally hundreds of different places in the video decoder code. Therefore the semaphore is potentially toggled in hundreds of places in the code (by means of the next_bits or get_bits methods). This results in a very fine-grained complicated task-level MTG* model.

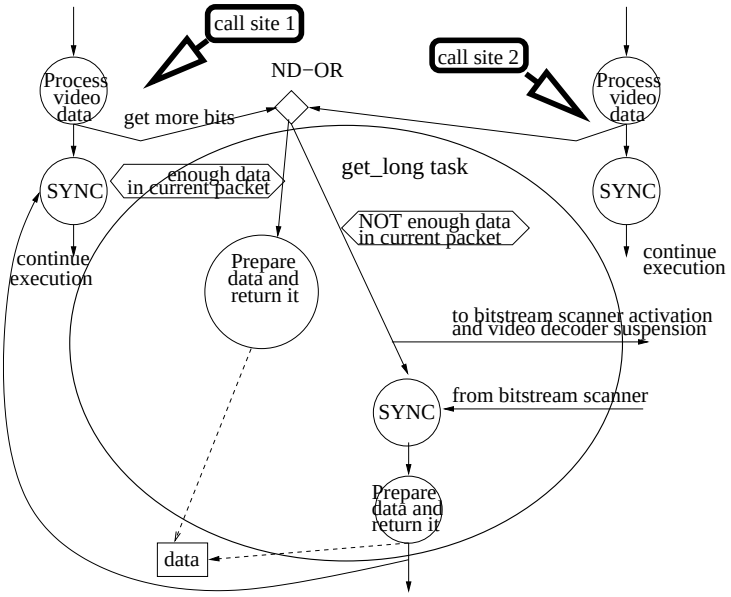
Part of an MTG* model for this situation is shown in figure 5.11. Each additional call site causes an additional control flow edge to the non-deterministic OR node and a thread frame boundary at the call site.

5.4.3 Video semaphore transformation

The fact that the system bitstream packet boundaries do not necessarily coincide with the video packet boundaries makes that any call to the io_tool.get_long method can result in toggling the video semaphore (with the corresponding thread suspensions and activations). Therefore the task-level model cannot be simplified by inlining and specializing the get_long calls in certain call sites.

By increasing the 'intelligence' and packet buffer size in the bitstream scanner, however, it is possible to make the model behave less non-deterministic. It is possible not only to scan but also to interpret the contents of the system bitstream data packets (duplication of some video decoder functionality within the bitstream scanner). Then it is possible to decode enough video data packets

Figure 5.11 Model for calling get_long from a call site.



at the same time to result in a limited number of get_long call sites where the video semaphore can be toggled (e.g. only at PIC or GOP boundaries in the video decoder). This would allow specialization of the get_long method at the call sites where the semaphore is certain not to be toggled. The thread frame boundaries at the call sites which are certain not to toggle the semaphore are then reduced to thread node boundaries. The task-level model is thus simplified and opportunities for thread node parallelization are created.

The trade-off which has to be made consists of non-determinism (= overhead at run-time due to synchronization, unpredictability, suspension, activation of tasks) versus buffer size (needed to hold video packets) and video packet interpretation cycles. The bigger the buffer size to hold video packets, the less fine-grained the interaction of the MPEG_scan and video decoding part of the task-level model is because the number of call sites where the semaphore can be toggled can be reduced. At the same time this should reduce context switching overhead.

In an extreme case both the bitstream scanner and video decoder part are fully integrated as opposed to work as two separate threads. This should certainly reduce overhead at run-time but it has two disadvantages. The first disadvantage is that the integrated system bitstream scanning and video stream decoding cannot be reused in case that the given stream is not a system stream but a video stream. This leads to a code size increase due to duplication of video decoder functionality. The second disadvantage is that the implementa-

Algorithm 3. *Sequentialized algorithm*

```
while (moreFrames) {
  DecodeNextFrame();
  DisplayNextFrame();
}
```

tion becomes harder to understand. The first disadvantage can probably be solved by clever programming (inserting many conditions). Unfortunately this will make the implementation even more difficult to understand[3].

5.4.4 Dispatch semaphore

The dispatch semaphore is the second semaphore which exists in the application. It is used as follows: when an image is ready it is put in a buffer and a boolean variable is toggled to indicate the presence of a new frame. The AnimatorThread notes the changed variable and uses this frame to display the image. It toggles the boolean variable again to signal the fact that a new image may be put in the buffer.

The dispatch semaphore is thus used to prevent resource collision. Application of the test to see if the semaphore is used as a concealed dependency returns true for the jit version of the application. The AnimatorThread does not need to be a separate thread and this part of the application could be rewritten as a simple loop shown in algorithm 3.

5.4.5 Profiling results

To get an idea of the impact of the sequentialization shown in algorithm 3 the JAVA multi-threading high-level estimator (byte code count profiler) developed by Richard Stahl at IMEC was used. Local impact is the impact of the transformation on the number of executed byte-code instructions in the function in which the synchronization occurs. Global impact is the impact of the transformation on the number of executed byte-code instructions in the whole program. Measurement results can be found in table 5.1. The following versions were built and tested:

- version 1: remove the dispatch semaphore with the technique from this chapter
- version 2: remove task creation from AnimatorThread and ScanThread (make them functions instead of JAVA threads)

[3]This is only an issue if the rest of the TCM flow is not fully automated. It is safe to assume this is the case.

- version 3: perform transformation on the image handling reducing required memory space from 10 complete images to only 2 complete images

Version 1 removes the dispatch semaphore with the technique from this chapter. The profiling results indicate that about half of the time spent in the function was due to the synchronization operation. An improvement of about 50% results from sequentializing the semaphore. The semaphore prevented two concurrent tasks from running concurrently, thus making the use of concurrency nothing but overhead. Removing the semaphore in this case was a pre-processing transformation as it made no cost factors worse.

Version 2 was enabled by removal of the dispatch semaphore. It shows no real improvement in either local or global measurements. This is mainly because the task creation and activation is a one-time event in the course the program execution. The effect of removing the task creation drowns in the instruction count of the bitstream and video decoding. Again this transformation is a pre-processing transformation.

Version 3 mainly improves on memory usage but the JAVA profiler at that time was not able to characterize this effect. Normally version 3 would be produced by the DTSE steps coming before TCM. It is not really a TCM transformation. It is a DTSE pre-processing transformation. In this particular case the transformation was independent of transformations 1 and 2 and hence it could have been done before transformations 1 and 2[4].

Looking at the global results only a small improvement (not more than 3.6%) can be observed. The small improvement is due to the synchronization operations not being dominant in this application. The real bottlenecks lie in the data processing. The sequentialized algorithm has been chosen by Richard Stahl at IMEC to serve as a starting point for automatic software parallelizing. This will typically require a bigger buffer for holding intermediate image results. The increased parallelism after software pipelining is expected to further decrease the execution time. Results of this transformation are still future research. This transformation will be an exploration transformation because it will improve execution time but requires larger memory space[5].

[4]The reason for making a remark about the transformation order is that at IMEC it is proposed to do T-DTSE transformations before TCM transformations. This is part of the so-called unified meta flow, a design flow for design flows aiming at dynamic applications with timing constraints and emphasis on low energy consumption.

[5]Note that the order of transformations does not conflict with the TCM script derived in appendix B.

byte code instruction count	local	global
original version	160k	25.3M
version 1	81k	24.7M
version 2	81k	24.7M
version 3	81k	24.4M

Table 5.1: Profiling results for MPEG JAVA semaphore sequentialization.

5.5 Conclusions

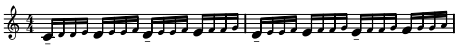
This chapter introduced the technique of unconditional sequentialization of semaphores to create freedom for parallelization. The gray box model of the application is used as an input for this transformation. First the model is simplified using graph transformations that reduce the model to a synchronization map. Secondly, interesting patterns are identified on the synchronization map. Interesting patterns are those which guarantee that whenever a token activates one piece of code guarded by the semaphore, all pieces of code guarded by the semaphore are eventually activated. If interesting patterns are identified, the semaphore in the gray box model can be replaced by one or more control flow edges. These control flow edges activate the different pieces of code guarded by one semaphore sequentially. This technique is correct by construction.

Because semaphores prevent parallelism the sequentialization does not remove parallelism and because it also removes synchronization overhead it is classified as a pre-processing transformation.

After sequentialization it sometimes becomes possible to increase parallelism. This was indicated in the MPEG JAVA example where software pipelining was proposed to increase the parallelism of the implementation. Because the effect of the semaphore transformation can enable such transformations as software pipelining it is classified as a freedom creation pre-processing transformation.

Chapter 6

Recursion removal



Manfred R. Schroeder (Dress Sequence),
in book: Number Theory in Science and
Communication, Springer

6.1 Introduction

Recursion is a powerful tool for specifying complex algorithms. Compilers for imperative languages typically implement recursive algorithms by maintaining a run-time stack. Every time a function calls itself the state of that function is pushed on the run-time stack, and execution restarts. Whenever the function returns, the state of the previous invocation is popped from the stack, and execution continues. This goes on until the stack is empty meaning that the recursion ended.

Special forms of recursion can be recognized by compilers and implemented more efficiently as a loop, avoiding the maintenance of the run-time stack. It is common belief that this is only possible for tail recursion or special forms of recursion that can be rewritten as tail recursion.

In this chapter a way of looking at recursion is presented that results in new ways to write recursion as an iteration. This is done in such a way that no new dependencies are introduced between the iterations. The method thus allows for easy parallelization and in addition it allows for systematic trade-off between execution time, memory and available parallelism. Applying the

Figure 6.1 An example algorithm

```
def bR(x,y):
  if (x<=0):
    print x, y
  elif (x>0):
    bR(x-1,y+2);
    bR(x-1,y*2);
    bR(x-1,y+1);
```

Example

```
def bR(x,y):
  if bc0(x,y):
    bcc0(x,y)
  elif rc0(x,y):
    bR(A[0][0](x,y),A[0][1](x,y))
    bR(A[1][0](x,y),A[1][1](x,y))
    bR(A[2][0](x,y),A[2][1](x,y))
```

General notation

method on a tail recursive algorithm results in the same loop as would result from traditional tail recursion optimization. The method presented in this chapter, however, goes much further in that it can be used to transform very general forms of recursion to iteration. More specifically it is applicable to non-mutual, non-linear, non-tail recursion, in which the different function calls can have or depend on side effects.

The aim of this chapter is to present the analysis of recursion, and to show how to replace it with iterative constructs. Due to space restrictions we only hint at how the systematic trade-offs become possible and we point to some of the results that are got from using the method on real-life designs.

In order to keep the chapter readable, the concepts will be illustrated with code fragments. The code fragments in this chapter are written in Python, a high level object oriented language, with a syntax that is readable for people used to C++ or JAVA like languages. Python is used because it allows for more concise specification of algorithms than C++ or JAVA. One thing to keep in mind is that grouping of statements is expressed by indentation as opposed to curly brackets in C++ or JAVA. The full code illustrating the concepts and techniques and containing more examples than can be given here is available from an ftp site, found in the bibliography [129].

6.2 Basic method and extensions

6.2.1 Relation between basic method and its extensions

Consider the code fragment in figure 6.1. It shows the definition of a recursive function bR taking two arguments x,y. If a condition x¡=0 is fulfilled a function print is called. If the condition is not fulfilled the function calls itself repeatedly with arguments that are transformed. The function can be notated more generally as shown in the column next to it.

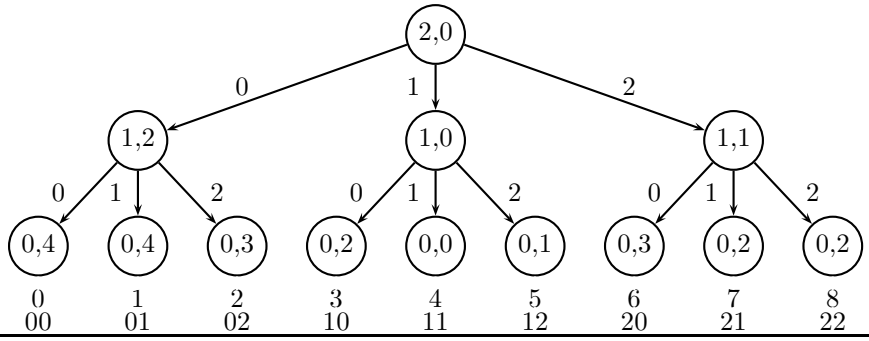
We start our discussion by looking at the call tree resulting from executing the example function with arguments x=2, y=0. The example function has some special characteristics. Because these special characteristics will be used in our observation, it will seem as if the method is usable in very special cases only. Throughout the text, however, the assumptions will be systematically relaxed, until fairly general forms of recursion can be handled. The assumptions that are true for our first example are: (1) All leaves in the call tree live at a finite, constant and known depth d in the call tree. The call tree root node has depth 0; (2) No function calls happen in between the calls of bR to itself. Such function calls will be named intermediate functions in the course of this chapter; (3) Only one base case is present, and only one recursive case. A base case is a condition under which the function returns without calling itself. Inside the base case a base case calculation can be performed using the argument values. A recursive case is a condition under which the function does call itself. Inside the recursive case intermediate functions can be activated before, in-between and after the recursive calls. These are not yet shown in the general notation of figure 6.1 in order not to overload it. (4) None of the function calls have side-effects (except for print statement in the base case calculation). (5) The function calls return no values.

By the time the end of the chapter is reached, these assumptions can be relaxed to: (1) Leaves in the call tree live at a finite depth. This depth must neither be constant nor known. The resulting code is simpler if an upper bound on the depth in the call tree is known but the technique is extensible to handle arbitrary depths with an unknown upper bound, although detailed discussion of this latter extension falls outside the scope of this text. In the latter case no guarantees can be given that the recursion will end. (2) Function calls can happen between the calls of bR to itself; these function calls can cause or depend on side-effects of other function calls. We do not allow such function calls to introduce mutual recursion. (3) Multiple base cases can be present, as well as multiple recursive cases. (4) Base case conditions and calculations, recursive case conditions and calculations, intermediate functions, argument transformation functions may all have side-effects. (5) The functions can have return values, and the return values of different recursive calls can be combined to a new return value using a return value combination function which itself can exhibit side-effects.

Many programs using non-mutual recursion can be rewritten so that our methods apply. Mutual recursion is not addressed here. Sometimes mutual recursion can be rewritten as non-mutual recursion [163, 120] though.

All this flexibility comes at a price, however. The more relaxed our assumptions are, the more bookkeeping will be needed and the less efficient the iterative version of the algorithm becomes. To get better insight in how different relaxations of assumptions influence the performance of the iterative version of the algorithm, the method is presented as a tool box consisting of a basic method and

Figure 6.2 Example call tree



a series of extensions. Each of these extensions can be used separately from the other extensions. Each extension targets the relaxation of one assumption. Each extension adds an amount of overhead to the iterative solution. It will become clear how using a particular extension impacts the performance.

6.2.2 Basic method on motivating example

Consider the call tree in figure 6.2. The recursive control flow traverses the call tree in depth-first order. The nodes show the argument values x and y as they are transformed during the recursion. As is visible in the call tree, each invocation of function bR results in either three or zero invocations of bR. Three invocations happen if a base case has not yet been reached. This shows itself as three child nodes in the call tree. If there are zero invocations of bR it means we are in a base case. This shows itself as a leaf in the call tree. Each arrow represents a function call. Every left arrow has been labeled 0, every middle arrow has been labeled 1 and every right arrow has been labeled 2. Arrows labeled 0 correspond to transforming the arguments with the functions $A[0][j]$ (see figure 6.1). Similarly arrows labeled 1 or 2 correspond to transforming the arguments with the functions $A[1][j]$, $A[2][j]$ respectively. The leaves can be numbered as well: see labels 0 to 8, and the corresponding representation in number base 3: 00 to 22. Two observations can be made:

1. The digits in the leaf number written in number base 3 indicate what argument transformations need to be applied (and in what order) to get from the root node to any leaf. As will become clear in this chapter, these digits play an important role in transforming more general forms of recursion (than our example) to iteration.

2. The argument values in different leaves needed to activate the base case calculation can be calculated independently of each other. This means that in principle each base case calculation can be activated with cor-

Figure 6.3 Iterative version of the example algorithm. Only valid within the assumptions stated in the text.

```
def bI(x0,x1):
  if bc0(x0,x1):
    bcc0(x0,x1)
  elif rc0(x0,x1):
    base, maxdepth = 3, x0
    itidx = BaseBCounter(base,maxdepth)
    for leafnr in range(0,base**maxdepth):
      tx0, tx1 = x0, x1
      for d in range(maxdepth-1,-1,-1):
        tx0,tx1 = A[itidx[d]][0](tx0,tx1),A[itidx[d]][1](tx0,tx1)
      bcc0(tx0,tx1)
      itidx.inc()
```

rect argument values in parallel. Note that side-effects of the base case calculations can introduce ordering constraints which prevent such parallelization. Such ordering constraints are present in the algorithm and are not introduced by the method described here. Suppose the base case calculation implements something like a filtering operation on a small part of an image, then such parallelization can yield tremendous speed-ups. Detailed techniques to exploit the freedom created by the transformations presented here is future work.

For now observe that an iterative version is easily constructed from the recursive example : make a counter that counts from 0 to 8 in number base 3. Then apply the correct argument transformation functions as indicated by the value of the digits in the counter. Finally call the base case calculation with these values. The iterative code corresponding to the example is shown in figure 6.3.

A BaseBCounter instance implements a counter counting in number base b with *maxdepth* digits. It implements a method `inc()` which adds 1 to a BaseBCounter object. Later in the text a method `incpow(p)` which will be used which adds $base^p$ to a given BaseBCounter number. It also implements the [] operator to access separate digits of the base b representation. The code for BaseBCounter is not given here but can be found at the ftp site (see reference section entry [129].). The $A[i][j]$ are argument transformation functions as before. The Python programming language supports so-called parallel assignment. This means that a statement like "x, y = y, x" results in the values of x, y being swapped, without the need to explicitly create a temporary intermediate variable as would be needed in a language like C or JAVA. The parallelization that was hinted at in observation 2, happens at the "for leafnr" loop level (line 7). Each iteration of this loop in principle can be done in parallel. The symbol # introduces a comment.

Note how the maximum depth in the call tree is fixed to 3. This means that the iterative implementation shown here only works for recursive algorithms in which all leaves live at depth 3. The counter counts in base 3. This means that the iterative implementation shown here only works correctly for algorithms that have three recursive calls in their single recursive case.

When the extension "leaves do not live at constant depth in the call tree" is introduced in section 6.3.1 it will be enough to know an upper bound on the depth of the call tree. Using the upper bound on the depth of the call tree guarantees that the iterative version will halt and at design-time a worst-case estimation of the execution time can be made.

An extension "unknown depth of call tree" is possible and can be used if an upper bound can not be found but discussion of this extension falls outside the scope of this chapter. Using the unknown depth extension it is no longer possible to know if/when the execution will end. Of course, in a real-life system, execution will necessarily end when the available memory is exhausted. Memory is used e.g. by the BaseBCounter to store digits. Luckily in many real-life designs, an upper bound can be found either by reasoning on the algorithm or even by profiling with worst-case examples.

First, the intuitive ideas leading to an iterative version of the recursive motivating algorithm are underpinned with some rigorously derived formulas.

6.2.3 Propagation equation and path evaluation equation

6.2.3.1 Operators

First some notations are introduced.

- Short-hand notation for a row-matrix/tuple:

$$(x_0, x_1, \ldots, x_{m-1}) \quad = \quad (x_i)_{i=0}^{m-1}$$

- Short-hand notation for a column-matrix:

$$\begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_{m-1} \end{bmatrix} = [x_i]_{i=0}^{m-1}$$

A *function composition operator* $\odot : \mathcal{F}^{N\times 1} \times \mathcal{F}^{N\times 1} \rightarrow \mathcal{F}^{N\times 1}$ is defined. $\mathcal{F}$ denotes a set of argument transformation functions A_{ij}. For the time being only algorithms with one recursive case are considered and hence the third subscript of the argument transformation functions can be left out. The result of applying operator $\odot$ to two column matrices of argument transforming functions A_{ij} is

the result of composing together the functions of the first matrix with the functions from the second matrix.

$$[A_{ip}]_{i=0}^{N-1} \odot [A_{jq}]_{q=0}^{N-1} \quad = \quad \left[A_{ip}\left((A_{jq})_{q=0}^{N-1}\right)\right]_{i=0}^{N-1}$$

Example 1. *The $\odot$ operator*

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \odot \begin{bmatrix} f_3 \\ f_4 \end{bmatrix} \quad = \quad \begin{bmatrix} f_1(f_3, f_4) \\ f_2(f_3, f_4) \end{bmatrix}$$

Using the $\odot$ operator the *call tree branch operator* $\oplus$ can be defined. Again the A_{ij} and B_{ij} are argument transformation functions.

$$\left([A_{ij}]_{j=0}^{m-1}\right)_{i=0}^{p} \oplus \left([B_{ij}]_{j=0}^{m-1}\right)_{i=o}^{q} \quad = \quad \left(\left([A_{kl}]_{l=0}^{m-1} \odot [B_{rs}]_{s=0}^{m-1}\right)_{k=0}^{p}\right)_{r=0}^{q}$$

The $\oplus$ operator can be seen as a vectorized version of the $\odot$ operator.

Example 2. *The $\oplus$ operator*

$$\begin{array}{ll} & \left(\begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix}, \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix}\right) \oplus \left(\begin{bmatrix} B_{00} \\ B_{01} \end{bmatrix}, \begin{bmatrix} B_{10} \\ B_{11} \end{bmatrix}, \begin{bmatrix} B_{20} \\ B_{21} \end{bmatrix}\right) \\ \textit{(def } \oplus\textit{)} \\ = & \left(\begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix} \odot \begin{bmatrix} B_{00} \\ B_{01} \end{bmatrix}, \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix} \odot \begin{bmatrix} B_{00} \\ B_{01} \end{bmatrix}, \begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix} \odot \begin{bmatrix} B_{10} \\ B_{11} \end{bmatrix},\right. \\ & \left. \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix} \odot \begin{bmatrix} B_{10} \\ B_{11} \end{bmatrix}, \begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix} \odot \begin{bmatrix} B_{20} \\ B_{21} \end{bmatrix}, \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix} \odot \begin{bmatrix} B_{20} \\ B_{21} \end{bmatrix}\right) \\ \textit{(def } \odot\textit{)} \\ = & \left(\begin{bmatrix} A_{00}(B_{00}, B_{01}) \\ A_{01}(B_{00}, B_{01}) \end{bmatrix}, \begin{bmatrix} A_{10}(B_{00}, B_{01}) \\ A_{11}(B_{00}, B_{01}) \end{bmatrix}, \begin{bmatrix} A_{00}(B_{10}, B_{11}) \\ A_{01}(B_{10}, B_{11}) \end{bmatrix},\right. \\ & \left. \begin{bmatrix} A_{10}(B_{10}, B_{11}) \\ A_{11}(B_{10}, B_{11}) \end{bmatrix}, \begin{bmatrix} A_{00}(B_{20}, B_{21}) \\ A_{01}(B_{20}, B_{21}) \end{bmatrix}, \begin{bmatrix} A_{10}(B_{20}, B_{21}) \\ A_{11}(B_{20}, B_{21}) \end{bmatrix}\right) \end{array}$$

6.2.3.2 Propagation equation

The following formulas allow to calculate the argument values of one specific node at a given depth d of the call tree.

Theorem 3. *A formula which produces all the argument values at depth d of the call tree is given by:*

$$N_d \quad = \quad \left(\bigoplus_{k=0}^{d}\left([A_{ij}]_{j=0}^{m-1}\right)_{i=0}^{n-1}\right) \oplus [x_p]_{p=0}^{m-1} \tag{6.1}$$

Where:

$$d \quad = \quad \textit{depth in call tree (counting from 0)}$$

Algorithm 4. *Toy example*

```
// suppose x,y is always positive or 0
def g(x,y):
 if x <= 0:
  bcc(x,y)
 else:
  g(x-1,x+y-3)
  print "x=",x,"y=",y
  g(x-y-3,x+y/2);
  print "2nd finished"
```

$$\begin{aligned} A_{ij} &= \textit{argument transforming functions} \\ x_p &= \textit{original argument values} \\ n &= \textit{number of recursive calls in recursive case} \\ m &= \textit{number of arguments of recursive function} \end{aligned}$$

Proof 4. *(By induction)*
For $d = 0$ the formula returns the original arguments (by convention). For $d = 1$ the formula returns the set of argument transformation functions which is correct.
Suppose that the formula holds for depth $d-1$, then it needs to be proved it also holds for d:

$$N_{d-1} = \left(\bigoplus_{k=0}^{d-1} \left([A_{ij}]_{j=0}^{m-1} \right)_{i=0}^{n-1} \right) \oplus [x_p]_{p=0}^{m-1}$$

To get leaves one level deeper the leaves of the current level (as given by N_{d-1}) are composed with the argument transformation functions:

$$\begin{aligned} N_d &= \left\{ \left([A_{ij}]_{j=0}^{m-1} \right)_{i=0}^{n-1} \oplus \left(\bigoplus_{k=0}^{d-1} \left([A_{ij}]_{j=0}^{m-1} \right)_{i=0}^{n-1} \right) \right\} \oplus [x_p]_{p=0}^{m-1} \\ &= \left(\bigoplus_{k=0}^{d} \left([A_{ij}]_{j=0}^{m-1} \right)_{i=0}^{n-1} \right) \oplus [x_p]_{p=0}^{m-1} \end{aligned}$$

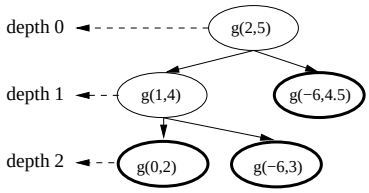
□

Formula 6.1 on the preceding page, referred to as *the propagation equation*, is illustrated on the following example. Function g(x,y) as given in algorithm 4 is used as a starting point.

Example 5. *The propagation equation*

From algorithm 4: $A_{00}(x,y) = x-1$, $A_{01}(x,y) = x+y-3$, $A_{10}(x,y) = x-y-3$, $A_{11}(x,y) = x+y/2$, $bc_0 = (x \leq 0)$. The argument values that are used in the base case can be calculated using the propagation equation 6.1 on the previous page. The propagation equation is evaluated for the function call

Figure 6.4 g(2,5) call graph. Leaves are drawn with a bold ellipse.



g(2,5). According to the propagation equation, for leaves situated at depth 1 in the call tree with $x = 2$, $y = 5$*:*

$$\begin{aligned}\left(\begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix}, \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix}\right) \oplus \begin{bmatrix} x \\ y \end{bmatrix} &= \left(\begin{bmatrix} A_{00}(x,y) \\ A_{01}(x,y) \end{bmatrix}, \begin{bmatrix} A_{10}(x,y) \\ A_{11}(x,y) \end{bmatrix}\right) \\ &= \left(\begin{bmatrix} x-1 \\ x+y-3 \end{bmatrix}, \begin{bmatrix} x-y-3 \\ x+y/2 \end{bmatrix}\right) \\ &= \left(\begin{bmatrix} 1 \\ 4 \end{bmatrix}, \begin{bmatrix} -6 \\ 4.5 \end{bmatrix}\right)\end{aligned}$$

And for leaves situated at depth 2 *in the call tree and* $x = 2$ *and* $y = 5$*:*

$$\begin{aligned}&\left(\begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix}, \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix}\right) \oplus \left(\begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix}, \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix}\right) \oplus \begin{bmatrix} x \\ y \end{bmatrix} \\ = \quad &\left(\begin{bmatrix} x-2 \\ 2\cdot x+y-7 \end{bmatrix}, \begin{bmatrix} -y-1 \\ 1.5\cdot x+0.5\cdot y-2.5 \end{bmatrix}, \right. \\ &\left. \begin{bmatrix} x-y-4 \\ 2\cdot x-0.5\cdot y-6 \end{bmatrix}, \begin{bmatrix} -1.5\cdot y-6 \\ -1.5\cdot x-0.75\cdot y-3 \end{bmatrix}\right) \\ = \quad &\left(\begin{bmatrix} 0 \\ 2 \end{bmatrix}, \begin{bmatrix} -6 \\ 3 \end{bmatrix}, \begin{bmatrix} -7 \\ -4.5 \end{bmatrix}, \begin{bmatrix} -13.5 \\ -3.75 \end{bmatrix}\right)\end{aligned}$$

This can be represented graphically as in figure 6.4. The results $\begin{bmatrix} -7 \\ -4.5 \end{bmatrix}$ *and* $\begin{bmatrix} -13.5 \\ -3.75 \end{bmatrix}$ *for leaves at depth* 2 *are not valid when* $x = 2$ *and* $y = 5$ *because they result from branching a base case.*

The propagation equation allows calculating how the argument values change throughout the recursion. This gives already a first method to remove recursion from algorithm 4 on the preceding page: evaluate the propagation equation until all leaves in the call tree are found. A node is a leaf if the base case condition returns 'true' when it is applied to the argument values in the node. Now it suffices to activate the base case calculations using the argument values found by the propagation equation.

While this method certainly works it is not efficient in terms of memory usage: all argument values at all depths in the call tree must be retained until the

leaves have been calculated. Therefore a new formula is derived in order to evaluate the argument values along one path from root to leaf node in the call tree without having to consider the argument values of the surrounding nodes.

6.2.3.3 Path evaluation equation

A formula is derived to calculate the argument values in the j-th node at depth d in the call tree. Nodes are numbered left to right on depth d. Consider the following part of the propagation equation with A_{ij} being argument transformation functions as before:

$$\begin{aligned}
& \left([A_{0j}]_{j=0}^{m-1}, \ldots, [A_{(n-1)j}]_{j=0}^{m-1}\right) \oplus \left([A_{0j}]_{j=0}^{m-1}, \ldots, [A_{(n-1)j}]_{j=0}^{m-1}\right) \\
= & \left([\ A_{0j}\ (A_{00}, A_{01}, \ldots, A_{0(m-1)})\ ]_{j=0}^{m-1}, \ldots, [\ A_{(n-1)j}\ (A_{(n-1)0}, A_{(n-1)1}, \ldots, A_{(n-1)(m-1)})\ ]_{j=0}^{m-1}\right) \\
= & \left([\alpha_{0j}]_{j=0}^{m-1}, \ldots, \left[\alpha_{(n^2-1)j}\right]_{j=0}^{m-1}\right) \qquad (6.2)
\end{aligned}$$

A shorter notation is introduced (replacing the matrices with single symbols):

$$\begin{aligned}
\overrightarrow{A_i} &= [A_{ij}]_{j=0}^{m-1} \\
\overrightarrow{\alpha_i} &= [\alpha_{ij}]_{j=0}^{m-1}
\end{aligned}$$

The equation 6.2 can be rewritten as:

$$\begin{aligned}
& \left(\overrightarrow{A_0}, \ldots, \overrightarrow{A_i}, \ldots, \overrightarrow{A_{n-1}}\right) \oplus \left(\overrightarrow{A_0}, \ldots, \overrightarrow{A_j}, \ldots, \overrightarrow{A_{n-1}}\right) \\
= & \ (\overrightarrow{A_0} \odot \overrightarrow{A_0}, \overrightarrow{A_1} \odot \overrightarrow{A_0}, \ldots, \overrightarrow{A_{n-1}} \odot \overrightarrow{A_0}, \overrightarrow{A_0} \odot \overrightarrow{A_1}, \ldots, \overrightarrow{A_{n-1}} \odot \overrightarrow{A_{n-1}}) \\
= & \ (\overrightarrow{\alpha_0}, \overrightarrow{\alpha_1}, ..., \overrightarrow{\alpha_{n^2-1}}) \qquad (6.3)
\end{aligned}$$

Obviously, $\overrightarrow{A_u} \odot \overrightarrow{A_v}$ results in $\overrightarrow{\alpha_{v \cdot n+u}}$. More interesting is to look at the inverse problem: given $\overrightarrow{\alpha_q}$, what are u, v to make $\overrightarrow{A_u} \odot \overrightarrow{A_v} = \overrightarrow{\alpha_q}$? The answer is found using straightforward algebra:

$$\begin{aligned}
u &= q \bmod n \\
v &= q \operatorname{div} n
\end{aligned}$$

Here 'div' means integer division: $a, b \in \mathbb{N}; b \neq 0 : \ a \operatorname{div} b = \left\lfloor \frac{a}{b} \right\rfloor$; 'mod' means remainder of integer division: $a, b \in \mathbb{N}; b \neq 0 : \ a \bmod b = a - (a \operatorname{div} b) \cdot b$.

Similarly, for $l \neq m$:

$$\begin{aligned}
& \left(\overrightarrow{A_0}, \ldots, \overrightarrow{A_i}, \ldots, \overrightarrow{A_{l-1}}\right) \oplus \left(\overrightarrow{B_0}, \ldots, \overrightarrow{B_j}, \ldots, \overrightarrow{B_{m-1}}\right) \\
= & \ (\overrightarrow{A_0} \odot \overrightarrow{B_0}, \overrightarrow{A_1} \odot \overrightarrow{B_0}, \ldots, \overrightarrow{B_{l-1}} \odot \overrightarrow{A_0}, \overrightarrow{B_0} \odot \overrightarrow{B_1}, \ldots, \overrightarrow{B_{l-1}} \odot \overrightarrow{B_{m-1}}) \\
= & \ (\overrightarrow{\alpha_0}, \overrightarrow{\alpha_1}, ..., \overrightarrow{\alpha_{l \cdot m-1}})
\end{aligned}$$

the relations become

$$\begin{aligned} u &= q \bmod l \\ v &= q \operatorname{div} l \end{aligned} \tag{6.4}$$

For leaves at depth 3 another term from the propagation equation 6.1 on page 103 is added:

$$\begin{aligned} & (\overrightarrow{\alpha_0}, \ldots, \overrightarrow{\alpha_{n^2-1}}) \oplus \left(\overrightarrow{A_0}, \overrightarrow{A_1}, \ldots, \overrightarrow{A_{n-1}}\right) \\ = & \left(\overrightarrow{\alpha_0} \odot \overrightarrow{A_0}, \overrightarrow{\alpha_1} \odot \overrightarrow{A_0}, \ldots, \overrightarrow{\alpha_{n^2-1}} \odot \overrightarrow{A_0}, \overrightarrow{\alpha_1} \odot \overrightarrow{A_0}, \ldots, \overrightarrow{\alpha_{n^2-1}} \odot \overrightarrow{A_{n-1}}\right) \\ = & \left(\overrightarrow{\beta_0}, \overrightarrow{\beta_1}, \ldots, \overrightarrow{\beta_{n^3-1}}\right) \end{aligned}$$

Using the equations 6.3 on the facing page and 6.4 and using some theorems from number theory: for $a, b, c \in \mathbb{N}$, $b, c \neq 0$: $(a \operatorname{div} b) \operatorname{div} c = a \operatorname{div} (b \cdot c)$, $(a \bmod (b \cdot c)) \bmod b = a \bmod b$, $(a \bmod b) \bmod (b \cdot c) = a \bmod b$, $(a \bmod (b \cdot c)) \operatorname{div} b = (a \operatorname{div} b) \bmod c$ it follows that:

$$\begin{aligned} \overrightarrow{\beta_q} &= \overrightarrow{\alpha_{q \bmod n^2}} \odot \overrightarrow{A_{q \operatorname{div} n}} \\ &= \left(\overrightarrow{A_{(q \bmod n^2) \bmod n}} \odot \overrightarrow{A_{(q \bmod n^2) \operatorname{div} n}}\right) \odot \overrightarrow{A_{q \operatorname{div} n^2}} \\ &= \left(\overrightarrow{A_{q \bmod n}} \odot \overrightarrow{A_{(q \operatorname{div} n) \bmod n}}\right) \odot \overrightarrow{A_{q \operatorname{div} n^2}} \end{aligned}$$

For a general depth $d > 0$ the following equation which calculates the argument values in the j-th node at depth d is found:

$$r_j = \bigodot_{i=0}^{d} \left(\overrightarrow{A_{(j \operatorname{div} n^i) \bmod n}}\right) \tag{6.5}$$

Here

r_j	=	argument values at depth d on j-th path from root to leaf in the call tree
j	=	the path number (iteration index)
d	=	depth of recursion
n	=	number of recursive calls in recursive case
$\overrightarrow{A_i}$	=	argument transformation functions of i-th recursive call

Equation 6.5 is important because it can be used to evaluate the equation 6.1 on page 103. Indeed, application of the equation 6.5 can calculate one specific set of node argument values in the call tree at a given depth d whereas calculating the same node directly using equation 6.1 on page 103 would require calculating *all* intermediate node argument matrices at depths $< d$. Equation 6.5 is called

the *path evaluation equation* because it is used to calculate the path from the root node to the j-th leaf of the call tree by choosing d to be (an upper bound of) the depth at which leaves live.

A particularly interesting observation is that the subscripts i of $\overrightarrow{A_i}$ form the reversed representation of index i in number base n as evidenced from the mod and div calculations. This can be exploited later on to optimize the implementation of the iterative algorithm.

Example 6. *Coming back to the toy example given in algorithm 4 on page 104:*

$$\overrightarrow{A_0} \odot \begin{bmatrix} A_{00} \\ A_{01} \end{bmatrix} = \begin{bmatrix} x-1 \\ x+y-3 \end{bmatrix} \qquad \overrightarrow{A_1} \odot \begin{bmatrix} A_{10} \\ A_{11} \end{bmatrix} = \begin{bmatrix} x-y-3 \\ x+y/2 \end{bmatrix}$$

For leaves at depth 1 *the equation 6.5 on the preceding page results with* d = *depth* = 1, m = *number of recursive calls in the recursive case* = 2 *in:*

$$\begin{aligned} r_j &= \bigodot_{i=0}^{1} \left(\overrightarrow{A_{(j \text{ div } 2^i) \bmod 2}} \right) \\ &= \overrightarrow{A_{j \bmod 2}} \odot \overrightarrow{A_{((j \text{ div } 2) \bmod 2)}} \end{aligned}$$

Hence,

$$(r_0, r_1, r_2, r_3) = (\overrightarrow{A_0} \odot \overrightarrow{A_0}, \overrightarrow{A_1} \odot \overrightarrow{A_0}, \overrightarrow{A_0} \odot \overrightarrow{A_1}, \overrightarrow{A_1} \odot \overrightarrow{A_1})$$

which indeed gives the same result as when using the propagation equation 6.1 on page 103 directly. Note how the subscripts of $\overrightarrow{A_i}$ *indeed form the reversed binary representation of the subscripts of the corresponding* r_i *values.*

Since the argument transformation functions in this example are linear combinations of the arguments it is possible to evaluate the $\odot$ *operator using matrix multiplications. Matrix multiplications are associative hence the brackets can be forgotten. In fact, the operator* $\odot$ *is associative in general as it is function composition (see e.g. [6]). It is shown how to apply* $\overrightarrow{A_0}$ *to argument matrix* $\begin{bmatrix} x \\ y \end{bmatrix}$. *The argument matrix is extended with a row containing a* 1. *Similarly, the* $\overrightarrow{A_i}$ *matrices are extended with a row containing all* 0, *and one* 1. *This also allows to include addition of constants to arguments in the matrix calculations. Then:*

$$\overrightarrow{A_0} \odot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = M_0 \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

By definition of $\overrightarrow{A_0}$*:*

$$\overrightarrow{A_0} \odot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \overset{(def\ A_0)}{=} \begin{bmatrix} x-1 \\ x+y-3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Hence,

$$M_0 = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 1 & -3 \\ 0 & 0 & 1 \end{bmatrix} \quad \textit{similarly, for } \overrightarrow{A_1}\textit{:} \; M_1 = \begin{bmatrix} 1 & -1 & -3 \\ 1 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This is used to calculate the third leaf at depth two. Reversing the binary (two recursive calls in the recursive case) representation of 3 (third leaf), using 2 bits (call tree depth two) gives us the A subscripts: $(3)_{10} = (10)_2 \rightarrow$ *subscripts* $0, 1$. *Evaluating a path evaluation expression like* $\overrightarrow{A_0} \odot \overrightarrow{A_1}$ *boils down to evaluating matrix multiplications:*

$$(\overrightarrow{A_0} \odot \overrightarrow{A_1}) \odot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = M_0 \cdot M_1 \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x - y - 4 \\ 2 \cdot x - 0.5 \cdot y - 6 \\ 1 \end{bmatrix}$$

This indeed corresponds to the third leaf at depth two that was calculated earlier in section 6.2.3.2 on page 103 using the propagation equation directly.

6.3 Comparison of iterative and recursive control flow

It is instructive at this point to compare the control flow of the recursive version to the control flow of the iterative version. The recursive version performs a depth-first traversal of the call tree, whereas our proposed iterative version follows each path from root node to leaf. Figure 6.5 shows this graphically.

The dashed lines in both the recursive and iterative version indicate argument transformations. The dotted lines in the recursive version denote back-tracking. Two conclusions can be drawn for now:

1. The edge labels indicate the ordering in time of the different argument transformations. As is visible on the call trees the recursive version actually performs less argument transformations than our iterative version. On the other hand, the recursive version stores the argument values on the stack until back-track, whereas the iterative version recalculates them as needed. The extra work will be quantified more precisely in section 6.3.6. Partial memoization will allow us to make a systematic trade-off of space versus argument transformation calculations (time).
2. It also means that if the argument transformations have side-effects, our iterative version might not yield the expected results because they are activated more often in the iterative version than in the recursive version, and the call trace looks different. Handling argument transformation functions with side-effects will be the topic of extension **4** in section 6.3.4.

Figure 6.5 Comparing the recursive control flow to the iterative control flow

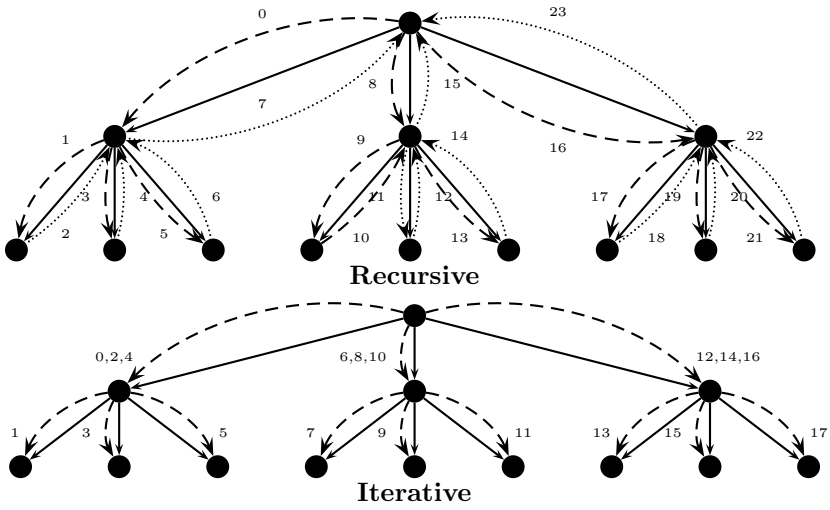
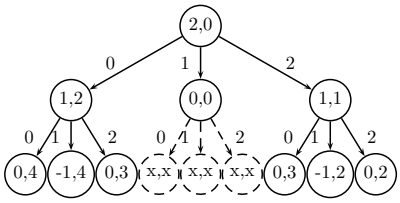


Figure 6.6 Algorithm with leaves at non-constant depth

```
1 def bR(x,y):
    if (x<=0):
3      print x, y
    elif (x>0):
5      bR(x-1,y+2);
       bR(x-2,y*2);
7      bR(x-1,y+1);
```



6.3.1 Extension 1: leaves at non-constant depth

In this section it is assumed an upper bound to the recursion depth is known. An example algorithm is depicted in figure 6.6. The general notation is the same as in figure 6.1. As is visible in the call tree, a base case is reached in node $(0, 0)$. At this time the BaseBCounter has value 10. The base case is reached while traversing the call tree from root to leaf before reaching the upper bound depth in the call tree. Increasing the BaseBCounter with 1 (from 10 to 11) at this point would result in the same base case being visited again during the next iteration. This is wrong, as in the recursive algorithm, after reaching node $(0, 0)$, the execution will back-track and continue with path 20. How to handle this in our iterative version is straightforward: while iterating from root to leaf: check for a base case. If one is found, increase the BaseBCounter with b^d, where b is number of recursive calls in the recursive case, and d is the depth in the call tree where the base case occurs. Increasing the BaseBCounter with b^d has the effect of skipping the dashed nodes in the call tree in figure 6.6. In

Figure 6.7 Iterative version with non-constant leaf depth

```
def ncdI(x0,x1):
  if bc0(x0,x1):
    bcc0(x0,x1)
  elif rc0(x0,x1):
    base, maxdepth = 3, x0
    itidx = BaseBCounter(base,maxdepth+1)
    while itidx[maxdepth]==0: # for each leaf
      tx0, tx1 = x0, x1
      for d in range(maxdepth-1,-1,-1): # follow path from root to leaf
        tx0, tx1 = A[itidx[d]][0](tx0,tx1),A[itidx[d]][1](tx0,tx1)
        if bc0(tx0, tx1):
          bcc0(tx0,tx1)
          break # value of d is retained
      itidx.incpow(d) # skip virtual leaves
```

the course of this chapter a base case occurring at a depth $d < maxdepth$ will be referred to as an **early base case**. The code for the iterative version is shown in figure 6.7. Note especially lines 7, 11 and 14 which are different from the algorithm in figure 6.3. Because sometimes the iteration counter must be increased with more than 1, the for loop is rewritten as a while loop (line 7). Because base cases can occur before reaching the upper bound on the depth of the call tree, an extra test to see whether a base case is encountered is inserted (line 11). The iteration counter is increased with more than 1 if the base case happens at a depth d smaller than the depth upper bound $maxdepth$ (line 14).

This extension can be refined for the case where no upper bound is known. While such refinement falls outside the scope of this text the basic idea is to dynamically enlarge the number of digits used in the iteration counter if the maximum depth is reached and no base case was found.

6.3.2 Extension 2: intermediate function calls

This extension handles calling functions in between the recursive calls. An example and the corresponding more general notation is given in figure 6.8.

It is instructive to annotate the call tree for the example in figure 6.8 with the activations of the intermediate functions $g[0]$ to $g[3]$. This is shown in figure 6.9. Looking at the annotated call tree two observations can be made.

1. The functions $g[0]$ to $g[2]$ are activated just before the argument transformation functions of the recursive calls. The index i of the $g[i]$, $(0 \leq i < b)$ that is activated equals the value of the corresponding digit of the BaseBCounter (i.e. the arrow label from figure 6.6). In other words: just before the $C[i][j]$ set of argument transformation functions is applied the

Figure 6.8 An example algorithm

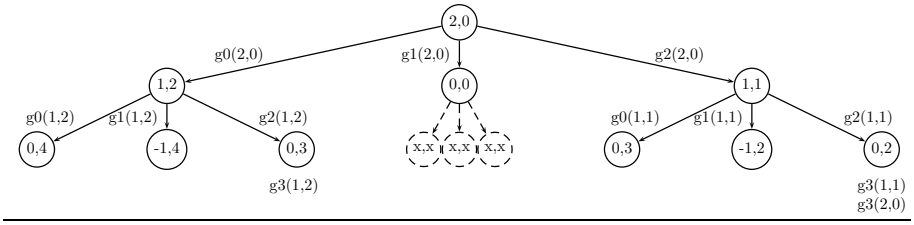
```
def if1Rexample(x,y):
   if (x<=0):
      print x,y
   elif (x>0):
      print "0:␣",x,y
      if1Rexample(x-1,y+2)
      print "1:␣",x,y
      if1Rexample(x-2,y*2)
      print "2:␣",x,y
      if1Rexample(x-1,y+1)
      print "3:␣",x,y
```

Example

```
def if1R(x0,x1):
   if bc0(x0,x1):
      bcc0(x0,x1)
   elif rc0(x0,x1):
      g[0](x0,x1)
      if1R(C[0][0](x0,x1),C[0][1](x0,x1))
      g[1](x0,x1)
      if1R(C[1][0](x0,x1),C[1][1](x0,x1))
      g[2](x0,x1)
      if1R(C[2][0](x0,x1),C[2][1](x0,x1))
      g[3](x0,x1)
```

General notation

Figure 6.9 Call tree annotated with intermediate function activation



$g[i]$ function needs to be activated with the argument values that were calculated by the previous argument transformation step.

Remember the difference in control flow between the recursive and iterative version of the algorithm as explained in figure 6.5 and section 6.3.2. In the iterative version each iteration follows a path from root to leaf in the call tree. If every activation of an argument transformation function is accompanied by an intermediate function activation erroneous results can come out if the intermediate functions have side-effects. Indeed, by following the recursive control flow (i.e. depth-first traversal) on figure 6.9 it becomes clear that the $g[i]$ $(0 \leq i < b)$ functions should only be activated for the part of the path that is not common to the previously evaluated path from root to leaf. Somewhat surprisingly, the leaf number of the current path contains all the information that is needed to find out when and what $g[i]$ functions to activate. The part of the path that is not common to the previously evaluated path when increasing the BaseBCounter with b^d (explained in section 6.3.1) corresponds to all digits from the least significant digit to the first non-zero digit (if any), going from least significant digit to most significant digit.

2. The function $g[3]$ is activated whenever all child nodes of a node have been visited. In such cases the leaf number written in base 3 ends in a series of digits 2. Stated more generally: the leaf number written in base b in such cases necessarily ends in a series of digits that equal $b-1$, where b is the number base of the BaseBCounter (i.e. the number of recursive calls in the recursive case).

Table 6.1 applies these two rules on a few example paths to determine when what $g[i]$ should be activated. We discuss the following two cases: (1) The general case is the one where the $g[i]$ functions depend on the correct argument values. In this case the activation of $g[i]$ should be scheduled just before the next argument transformation step is applied. Then the correct values are available without the need to store them somewhere. (2) A special case exists where the $g[i]$ do not take the argument values as parameters and the order of execution of the $g[i]$ with respect to the base case tests and calculations and recursive case tests does not matter. In this special case the execution of the $g[i]$ function can be rescheduled in time with respect to the argument transformation functions. The $g[i]$ activations can be scheduled after all argument transformation steps on one path from root to leaf have been evaluated. This opens possibilities for (symbolic) simplification of the path evaluation. An application of this special case taken from a real-life design problem is given in section 6.4 dealing with an MPEG4 visual texture coding application. See figures 6.10 and 6.11 for iterative implementations of the recursive code from figure 6.8 using our observations.

Figure 6.10 Python implementation for intermediate function activation without g_b

```
def if1I(x0,x1):
  if bc0(x0,x1):
    bcc0(x0,x1)
  elif rc0(x0,x1):
    base, maxdepth = 3, x0
    itidx = BaseBCounter(base,maxdepth+1)
    while itidx[maxdepth]==0:
    tx0, tx1, gcnt = x0, x1, 0
    while (gcnt < len(itidx.digits)) and (itidx[gcnt] == 0):
      gcnt = gcnt+1
    for d in range(maxdepth-1,-1,-1):
      if gcnt==d:
        g[itidx[d]](tx0,tx1)
      elif gcnt > d:
        g[0](tx0,tx1)
      tx0, tx1 = C[itidx[d]][0](tx0,tx1),C[itidx[d]][1](tx0,tx1)
      if bc0(tx0, tx1):
        bcc0(tx0,tx1)
        break
    itidx.incpow(d)
```

General case: g_i interwoven with argument transformation steps.

```
def if3I(x0,x1):
  if bc0(x0,x1):
    bcc0(x0,x1)
  elif rc0(x0,x1):
    base, maxdepth = 3, x0
    itidx = BaseBCounter(base,maxdepth+1)
    while itidx[maxdepth]==0:
      tx0, tx1, gcnt, btcnt = x0, x1, 0, 0
      while (gcnt < len(itidx.digits)) and (itidx[gcnt] == 0):
        gcnt = gcnt+1
      for d in range(maxdepth-1,-1,-1):
        tx0, tx1 = E[itidx[d]][0](tx0,tx1),E[itidx[d]][1](tx0,tx1)
        if bc0(tx0, tx1):
          bcc0(tx0,tx1)
          btcnt = gcnt
          break
      for e in range(maxdepth-1,-1,-1):
        if (gcnt == e) and (e>=d):
          g[itidx[e]]()
        elif (gcnt > e) and (e>=d):
          g[0]()
      itidx.incpow(d)
```

Special case: g_i handling separate from path evaluation.

Figure 6.11 Python implementation for intermediate function activation with g_b

Just below the bcc0() call (between line 18 and 19 in the upper part of figure 6.10), and aligned with it, insert the following code to handle the g_3 function activation. Note how handling the g_3 function makes the time needed by the iterative version larger: an extra loop is needed to calculate the argument values that are used during the back-tracking. This can be avoided by using memoization (not shown) at the expense of using more memory.

```
while btcnt < len(itidx.digits) and itidx[btcnt-gcnt+d]==itidx.base-1:
  hx0, hx1 = x0, x1
  for e in range(maxdepth-1,btcnt,-1):
    hx0, hx1 = C[itidx[e]][0](hx0,hx1),\
                    C[itidx[e]][1](hx0,hx1)
    g[itidx.base](hx0,hx1)
    btcnt = btcnt+1
```

General case

Just before the incpow(d) call (between line 21 and 22 in the lower part of figure 6.10), and aligned with it, insert the following code to handle the g_3 function activation.

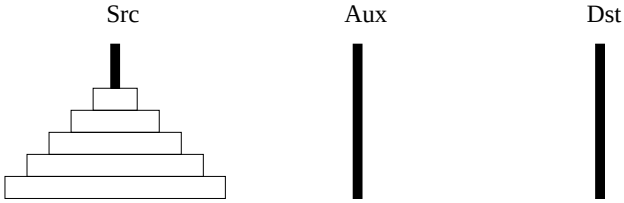
```
while btcnt < len(itidx.digits) and itidx[btcnt-gcnt+d]==itidx.base-1:
  g[itidx.base]()
  btcnt = btcnt+1
```

Special case

Table 6.1: Finding out what intermediate functions g_i to activate in the iterative version. Digits after and including last non-zero digit indicated between { }. Leaf number ending in series of digits $b-1$ indicated in [ ]. The arguments to the g_i function are available from the previous argument transformation. Canceled table items indicate paths that are not evaluated and functions that are never activated due to an early base case.

path (leaf number)	g_i $(0 \leq i < b)$ to activate	g_b to activate
{00}	$g_0(2,0)$, $g_0(1,2)$	
0{1}	$g_1(1,2)$	
0[{2}]	$g_2(1,2)$	$g_3(1,2)$
{10}	$g_1(2,0)$, ~~$g_0(1,2)$~~	
~~1{1}~~	~~$g_1(0,0)$~~	
~~1{2}~~	~~$g_2(0,0)$~~	~~$g_3(0,0)$~~
{20}	$g_2(2,0)$, $g_0(1,1)$	
2{1}	$g_1(1,1)$	
[2{2}]	$g_2(1,1)$	$g_3(1,1), g_3(2,0)$

Figure 6.12 Towers of Hanoi problem



Challenge: Move all disks from the Src pole to the Dst pole, never putting a bigger disk on top of a smaller one, in the smallest amount of moves. Disks can be put on one of three poles only.

We now proceed to illustrate this extension on the well known Towers of Hanoi algorithm.

Example 7. *Towers of Hanoi*

The theory is illustrated on a well known example: the towers of Hanoi. The problem is depicted in figure 6.12. The recursive algorithm for the solution of the problem is given in algorithm 5 on the facing page. As can be seen the algorithm is easy to analyze as it has only one recursive case and all leaves reside at a constant depth N (which is a parameter that is known at run-time, just before the Hanoi solver is called). The argument transformation functions are identified first:

$$\begin{aligned} A_{00}(N, Src, Aux, Dst) &= N - 1 \\ A_{01}(N, Src, Aux, Dst) &= Src \end{aligned}$$

Algorithm 5. *Towers of Hanoi, recursive solution*

```
def solve(N, Src, Aux, Dst):
 if N == 0:
  pass
 else:
  solve(N-1, Src, Dst, Aux)
  print "move from",Src,"to",Dst
  solve(N-1, Aux, Src, Dst)
```

$$
\begin{aligned}
A_{02}(N, Src, Aux, Dst) &= Dst \\
A_{03}(N, Src, Aux, Dst) &= Aux \\
A_{10}(N, Src, Aux, Dst) &= N - 1 \\
A_{11}(N, Src, Aux, Dst) &= Aux \\
A_{12}(N, Src, Aux, Dst) &= Src \\
A_{13}(N, Src, Aux, Dst) &= Dst
\end{aligned}
$$

Hence,

$$
\overrightarrow{A_0} \odot \begin{bmatrix} N \\ Src \\ Aux \\ Dst \end{bmatrix} = \begin{bmatrix} N-1 \\ Src \\ Dst \\ Aux \end{bmatrix} \qquad \overrightarrow{A_1} \odot \begin{bmatrix} N \\ Src \\ Aux \\ Dst \end{bmatrix} = \begin{bmatrix} N-1 \\ Aux \\ Src \\ Dst \end{bmatrix}
$$

It can be seen that application of $\overrightarrow{A_0}$ *decreases* n *with one and swaps rows* 3 *and* 4*. Similarly, application of* $\overrightarrow{A_1}$ *decreases* n *with one and swaps rows* 2 *and* 3*.*

According to the path evaluation equation, for a given value of n*, and with* $m = 2$*:*

$$
r_j = \bigodot_{i=0}^{n-1} \overrightarrow{A_{(j \text{ div } m^i) \bmod m)}}
$$

Now the evaluation of r_j *must be implemented. Applying* $\overrightarrow{A_0}$ *to a given matrix with argument values results in this matrix with swapped third and fourth row and decreased version of* n*:*

$$
\overrightarrow{A_0} \left(\begin{bmatrix} n \\ s \\ a \\ d \end{bmatrix} \right) = \begin{bmatrix} n-1 \\ s \\ d \\ a \end{bmatrix}
$$

Ignoring the decrement of n *(for no other reason than to reduce typesetting*

space), application of $\overrightarrow{A_0}$ *can be interpreted as doing a matrix multiplication:*

$$\overrightarrow{A_0}\left(\left[\begin{array}{c} s \\ a \\ d \end{array}\right]\right) = M_0 \cdot \left[\begin{array}{c} s \\ a \\ d \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] \cdot \left[\begin{array}{c} s \\ a \\ d \end{array}\right] = \left[\begin{array}{c} s \\ d \\ a \end{array}\right]$$

Similarly, applying $\overrightarrow{A_1}$ *is another permutation:*

$$\overrightarrow{A_1}\left(\left[\begin{array}{c} s \\ a \\ d \end{array}\right]\right) = M_1 \cdot \left[\begin{array}{c} s \\ a \\ d \end{array}\right] = \left[\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{c} s \\ a \\ d \end{array}\right] = \left[\begin{array}{c} a \\ s \\ d \end{array}\right]$$

$\overrightarrow{A_0} \odot \overrightarrow{A_1}$ *is then found using:*

$$(\overrightarrow{A_0} \odot \overrightarrow{A_1}) \odot \left[\begin{array}{c} s \\ a \\ d \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] \cdot \left[\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right] \cdot \left[\begin{array}{c} s \\ a \\ d \end{array}\right] = \left[\begin{array}{c} a \\ d \\ s \end{array}\right]$$

Hence the effect of $\overrightarrow{A_0} \odot \overrightarrow{A_1}$ *can be summarized as follows: the matrix elements have 'rotated' one row upwards. A list can be made of interpretations of various combinations of* $\overrightarrow{A_j}$ *operators:*

$$\begin{aligned}
\overrightarrow{A_0} &= \textit{swap rows 2 and 3} \\
\overrightarrow{A_0} \odot \overrightarrow{A_1} &= M_0 \cdot M_1 \\
&= \textit{rotate rows 1 up} \\
\overrightarrow{A_0} \odot \overrightarrow{A_1} \odot \overrightarrow{A_0} &= \overrightarrow{A_1} \odot \overrightarrow{A_0} \odot \overrightarrow{A_1} \\
&= M_1 \cdot M_0 \cdot M_1 \\
&= \textit{swap rows 1 and 3} \\
\overrightarrow{A_1} \odot \overrightarrow{A_0} \odot \overrightarrow{A_1} \odot \overrightarrow{A_0} &= \overrightarrow{A_0} \odot \overrightarrow{A_1} \\
&= \textit{rotate rows 1 up}
\end{aligned}$$

$$\begin{aligned}
\overrightarrow{A_1} &= \textit{swap rows 1 and 2} \\
\overrightarrow{A_1} \odot \overrightarrow{A_0} &= M_1 \cdot M_0 \\
&= \textit{rotate rows 1 down} \\
\overrightarrow{A_0} \odot \overrightarrow{A_1} \odot \overrightarrow{A_0} \odot \overrightarrow{A_1} &= \overrightarrow{A_1} \odot \overrightarrow{A_0} \\
&= \textit{rotate rows 1 down} \\
\overrightarrow{A_0} \odot \overrightarrow{A_0} &= \overrightarrow{A_1} \odot \overrightarrow{A_1} \\
&= \textit{do nothing (nop)}
\end{aligned}$$

The fact that $\overrightarrow{A_0} \odot \overrightarrow{A_0}$ *and* $\overrightarrow{A_1} \odot \overrightarrow{A_1}$ *have no effect can be used to simplify the path evaluation equation 6.1 on page 103. This is illustrated in the calculation of* r_{29} *for the Hanoi problem with 4 disks (hence* $j = 29$*,* $n = 4$*,* $m = 2$*):*

$$\begin{aligned}
r_{29} &= \bigodot_{i=0}^{4} \overrightarrow{A_{(29 \text{ div } 2^i) \bmod 2}} \\
&= \underbrace{\overrightarrow{A_1} \odot \overrightarrow{A_0} \odot \overrightarrow{A_1}}_{\textit{swap row 1 and 3}} \quad \underbrace{\odot \overrightarrow{A_1} \odot \overrightarrow{A_1}}_{\textit{nop}} \\
&= \textit{swap row 1 and 3}
\end{aligned}$$

Note that the subscripts of the (unsimplified) $\overrightarrow{A}$*-operator sequence form the (reversed) binary representation of* j*. The result of* r_{29} *is the net result of all*

side-effects after completing path number 29*. This is not equal to the 29th move as will become clear in the next paragraph.*

Because it depends on the values of the recursive function arguments the intermediate function g_1 (here a single 'print') must be activated at the correct moment during evaluation of each r_j. As the 'print' prints out the solution the moment of activation of the 'print' is also the moment when the next move is known. Because the argument transformation functions have no side effects and because no base case calculation is present no reason exists to further complete the full path equation for r_j after the 'print' has been activated. It suffices to stop when the 'print' is activated and to immediately move on to the next path. This way less work needs to be done in the iterative version than in the recursive version. The intermediate function activation rules are used to find out when the g_1 function must be activated. As it turns out it must be activated right before the last $\overrightarrow{A_1}$ step was applied during evaluation of the path evaluation equation. At this moment the j-th move in the problem is known.

This is now illustrated on the calculations of the 29th move in a Hanoi problem with 5 disks. First calculate r_{29}, then look when the 'print' function has to be activated. At this moment the 29*th move is known.*

$$r_{29} = \bigodot_{i=0}^{4} \overrightarrow{A_{(29 \text{ div } 2^i) \bmod 2}}$$

$$= \overrightarrow{A_1} \odot \overleftarrow{\overrightarrow{A_0} \odot \overrightarrow{A_1} \odot \overrightarrow{A_1} \odot \overrightarrow{A_1}}$$

argument values are available here ↑ *evaluation order*

Hence the 29th move is $\overrightarrow{A_0} \odot \overrightarrow{A_1} \odot \overrightarrow{A_1} \odot \overrightarrow{A_1}$. To see what this means first simplify this expression using the table of $\overrightarrow{A_j}$ operator interpretations. This results in $\overrightarrow{A_0} \odot \overrightarrow{A_1}$. The interpretation of $\overrightarrow{A_0} \odot \overrightarrow{A_1}$ is given by rotating the original arguments one step up. Hence the result of $\overrightarrow{A_0} \odot \overrightarrow{A_1}$ applied to $\begin{bmatrix} Src & Aux & Dst \end{bmatrix}^T$ — the arguments originally passed to the Hanoi solver — gives $\begin{bmatrix} Aux & Dst & Src \end{bmatrix}^T$. The 'print' function will print 'move from arg1 to arg3' meaning from Aux to Src.

Algorithm 6 on the following page summarizes how to calculate the j-th move in an N disk Hanoi tower problem.

Making an iterative version of the Hanoi solver is trivial by using these results: it suffices to synthesize a loop in which you calculate the consecutive moves using algorithm 6 on the next page. Each move can be calculated independently of the other moves thus enabling full parallelization.

6.3.3 Extension 3: multiple base cases and multiple recursive cases

Algorithm 6. *Calculating j-th move in N disk Hanoi tower problem*

To calculate the j-th move ($j = 1 \dots 2^N - 1$) in a N disk Hanoi tower problem: (*each step is illustrated on the 113th move of the 9-disk Hanoi problem*).

1. Write down the binary representation of j in reverse (i.e. from LSB to MSB). Pad with zeros in back until you have used N bits.
   Applied: Reversed Binary representation of 113 *using* 9 *bits is* 100011100.

2. Now take only the bits coming behind the first 1 bit.
   Applied: bits after first 1 *bit is* 00011100.

3. (Incidentally, the previous two steps can be calculated efficiently:
   trim first zero of the reverse N-bit representation of $j\&(j-1)$
   Applied: 113&112 = 001110001&001110000 = 001110000*, reverse and trim left-most zero:* 00011100*)*
   The left-most digit after reversing is always zero since always either j or j-1 is even.

4. Simplify repeatedly as follows:

   (a) wherever you have two consecutive 0 or 1 bits, remove them both (i.e. 00 →< empty > and 11 →< empty >)

   (b) wherever you have two consecutive 01 or 10 bit-combinations replace them with one such reversed combination (i.e. 0101 → 10, 1010 → 01)
   Applying (a) and (b) repeatedly to the 113*th move in a* 9 *disk problem results in* 01.

5. What you are left with is one of the following bit combinations:

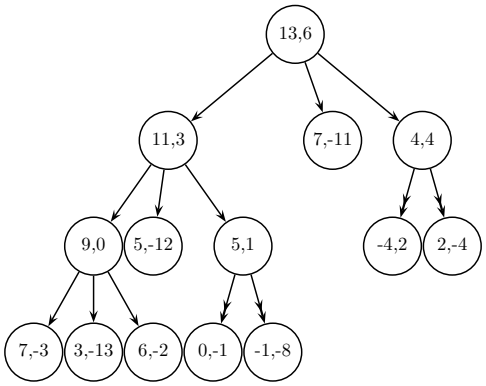
combination	action
0	Src→Aux
1	Aux→Dst
01	Aux→Src
10	Dst→Aux
101 or 010	Dst→Src
<empty>	Src→Dst

6. Incidentally, the series of substitutions has some remarkable properties:

   - It is easy to prove using modulo arithmetic that $n \bmod 3$ where $n \in \mathbb{N}$ is written in binary representation can be found by adding together the bits 0,2,... and subtracting from it the bits 1,3,5,... (all arithmetic modulo 2).
   - Then it is also easy to prove that each of the substitutions in the table above does not change the result of $j \bmod 3$.
   - Finally, by looking into the table, and combining this observation with rule 3, we can see that the "From" pole is uniquely defined by the result of $(j\&(j-1)) \bmod 3$.
   - Although the current reasoning doesn't make it clear, it is known that the "To" pole is uniquely defined by the result of $((j|(j-1)) + 1) \bmod 3$. I do not know of a reference where these remarkable results are proved.

Figure 6.13 Multiple recursive cases: nodes have different number of children depending on recursive case

```
def mbmrorig(x,y):
    if x<0:
        print "bc0:",x,y
    elif y<0:
        print "bc1:␣",x,y
    elif x-y>6:
        print "g00:",x,y
        mbmrorig(x-2,y-3)
        print "g01:",x,y
        mbmrorig(x-6,y-x-4)
        print "g02:",x,y
        mbmrorig(x-y-3,y-2)
        print "g03:",x,y
    elif x-y<=6:
        print "g10:",x,y
        mbmrorig(x-y-4,y-2)
        print "g11:",x,y
        mbmrorig(x-1,y-x-4)
        print "g12:",x,y
```



Example **Call tree mbmrorig(13,6)**

The extension to multiple base cases is almost trivial. Instead of checking for one base case during the iterations, check for multiple base cases instead, and in the same order as they were specified in the recursive algorithm. Base cases and recursive cases can be interleaved in the recursive algorithm. The recursive algorithm should be rewritten to have all base cases listed first and all recursive cases listed last. It is not clear yet if this is always possible, especially if the tests have side-effects. In practice we haven't seen any example (other than contrived ones) where the base cases and recursive cases were interleaved and where at the same time those tests had side-effects.

The extension to multiple recursive cases presents a problem not present in the multiple base case extension: depending on which recursive case is selected, the number of children can be different. Two types of arrows appear in figure 6.13. Arrows with one arrowhead indicate control flow is going through the first recursive case. Arrows with two arrowheads correspond to argument transformation steps from the second recursive case. A given node for this example now either has three children (first recursive case), two children (second recursive case) or zero children (base case). This seems to make the application of our method impossible: each digit in the BaseBCounter suddenly needs to have a different number base (depending on which recursive case is active), and this number base changes as we proceed. The problem is visible in figure 6.13.

This makes the direct application of our method impossible. For this reason, first the recursive algorithm is rewritten in such a way that the number of children for each node is a constant $b > 0$ (or zero for leaves). Luckily this is always possible, and easy. The idea is to add recursive calls which have no effect: they immediately return. This can be achieved by adding to the function a new argument fbc (i.e. **f**orced **b**ase **c**ase) which can take two values: `NORMAL` or `FORCED`, and a new base case: `if fbc==FORCED: return`. Dummy recursive calls can then be inserted by a call with argument fbc equal to `FORCED`. Normal recursive calls use fbc equal to `NORMAL`.

The technique and its effect of *padding* the call tree for the example from figure 6.13 is illustrated in figure 6.14.

The new nodes that appear in the call tree are indicated in a bold circle. Now the call tree has the same topology as previously with one recursive case. Therefore the same methods can be applied, provided that the correct set of argument transformations is applied depending on the recursive case which is active. In other words, before going one level deeper in the call tree, the recursive case must be determined and the argument transformation functions belonging to that recursive case must be applied. This is clearly visible in the code of figure 6.15. In line 5 and line 21 the code for checking multiple base cases is shown. The first time is before the iteration starts, to make sure that it is needed to start iterating. The second time is during the iteration. Line 9 iterates over the different paths from root to leaf as before. Lines 12-14 determine what intermediate functions need to be activated. Line 15 starts the evaluation of one path from root to leaf, meanwhile activating the intermediate functions as needed (lines 16-18), applying the correct argument transformation steps (lines 19-22), checking for a base case after each argument transformation step (lines 24-41). If a base case is found, the base case calculation is activated (line 25), the intermediate functions during back-tracking are activated (lines 27-38). Because in the most general case the intermediate functions depend on the values of the arguments, and because it was chosen not to store the argument values somewhere, they need to be recalculated (lines 30-36). This recalculation (which is only needed if an intermediate function exists after the last recursive call) can be replaced by a table look-up, if the argument values are stored in a table. Trade-offs that result from this will be discussed a bit in section 6.3.6.

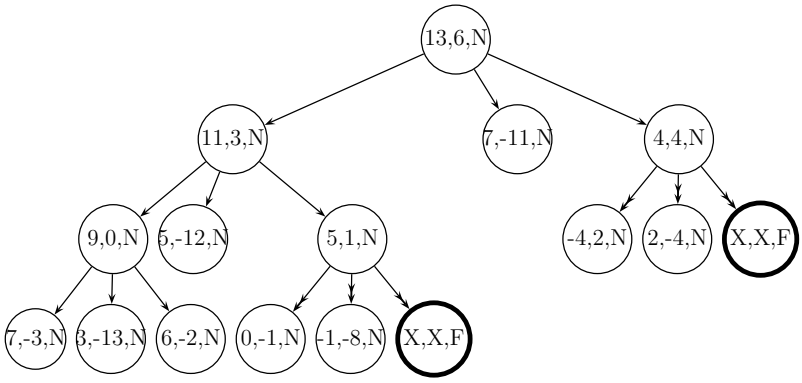
6.3.4 Extension 4: side effects

In many real-life algorithms written in imperative style, functions have side-effects. Side-effects change the global state of the program or its output. As a consequence a series of function calls with side-effects can yield different results depending on the order in which they are called. Examples of such functions are found frequently in real-life programs. Consider two functions

Figure 6.14 Multiple recursive cases: effect of adding dummy calls

```
def mbmrorig(x,y,fbc):
  if fbc==FORCED:
    return
  elif x<0:
    print "bc0:",x,y
  elif y<0:
    print "bc1:",x,y
  elif x-y>6:
    print "g00:",x,y
    mbmrorig(x-2,y-3,NORMAL)
    print "g01:",x,y
    mbmrorig(x-6,y-x-4,NORMAL)
    print "g02:",x,y
    mbmrorig(x-y-3,y-2,NORMAL)
    print "g03:",x,y
  elif x-y<=6:
    print "g10:",x,y
    mbmrorig(x-y-4,y-2,NORMAL)
    print "g11:",x,y
    mbmrorig(x-1,y-x-4,NORMAL)
    print "g12:",x,y
    mbmrorig(DONTCARE,DONTCARE,FORCED)
```

Code after inserting fbc



Effect on the call tree

Figure 6.15 An example algorithm

```
def mbmrR(x0,x1,fbc):
 if fbc==FAKE:
  pass
 elif basec[0](x0,x1,fbc):
  basecc0(x0,x1,fbc)
 elif basec[1](x0,x1,fbc):
  basecc1(x0,x1,fbc)
 elif recc[0](x0,x1,fbc):
  gr[0][0](x0,x1,fbc)
  mbmrR(F[0][0][0](x0,x1,fbc),\
    F[0][0][1](x0,x1,fbc),\
    F[0][0][2](x0,x1,fbc))
  gr[0][1](x0,x1,fbc)
  mbmrR(F[0][1][0](x0,x1,fbc),\
    F[0][1][1](x0,x1,fbc),\
    F[0][1][2](x0,x1,fbc))
  gr[0][2](x0,x1,fbc)
  mbmrR(F[0][2][0](x0,x1,fbc),\
    F[0][2][1](x0,x1,fbc),\
    F[0][2][2](x0,x1,fbc))
  gr[0][3](x0,x1,fbc)
 elif recc[1](x0,x1,fbc):
  gr[1][0](x0,x1,fbc)
  mbmrR(F[1][0][0](x0,x1,fbc),\
    F[1][0][1](x0,x1,fbc),\
    F[1][0][2](x0,x1,fbc))
  gr[1][1](x0,x1,fbc)
  mbmrR(F[1][1][0](x0,x1,fbc),\
    F[1][1][1](x0,x1,fbc),\
    F[1][1][2](x0,x1,fbc))
  gr[1][2](x0,x1,fbc)
  mbmrR(F[1][2][0](x0,x1,fbc),\
    F[1][2][1](x0,x1,fbc),\
    F[1][2][2](x0,x1,fbc))
  gr[1][3](x0,x1,fbc)
```

Recursive code (general notation)

```
def mbmrI(x0,x1,fbc):
 nrbc, base, maxdepth = 2, 3, max(x0,x1)
 itidx = BaseBCounter(base,maxdepth+1)
 if fbc==FORCED: return
 for i in range(0,nrbc):
  if basec[i](x0,x1,fbc):
   basecc[i](x0,x1,fbc)
   return
 while itidx[maxdepth]==0:
  tx0, tx1, tfbc, gcnt, btcnt, bc =\
  x0, x1, fbc, 0, 0, 0
  while gcnt < len(itidx.digits)\
        and (itidx[gcnt] == 0):
   gcnt = gcnt+1
  for d in range(maxdepth-1,-1,-1):
   crc = reccase(tx0,tx1,tfbc)
   if gcnt==d: gr[crc][itidx[d]](tx0,tx1,tfbc)
   elif gcnt > d: gr[crc][0](tx0,tx1,tfbc)
   tx0, tx1, tfbc=\
       F[crc][itidx[d]][0](tx0,tx1,tfbc),\
       F[crc][itidx[d]][1](tx0,tx1,tfbc),\
       F[crc][itidx[d]][2](tx0,tx1,tfbc)
   for e in range(0,nrbc):
    if basec[e](tx0, tx1, tfbc):
     basecc[e](tx0,tx1,tfbc)
     btcnt, crc2 = gcnt, 0
     while btcnt < len(itidx.digits)\
           and itidx[btcnt-gcnt+d]==itidx.base-1:
      hx0, hx1, hfbc = x0, x1, fbc
      for f in range(maxdepth-1,btcnt,-1):
       crc2 = reccase(hx0,hx1,hfbc)
       hx0, hx1, hfbc =\
          F[crc2][itidx[f]][0](hx0,hx1,hfbc),\
          F[crc2][itidx[f]][1](hx0,hx1,hfbc),\
          F[crc2][itidx[f]][2](hx0,hx1,hfbc)
      crc2 = reccase(hx0,hx1,hfbc)
      gr[crc2][itidx.base](hx0,hx1,hfbc)
      btcnt = btcnt+1
     bc=1
     break
    if bc: break
  itidx.incpow(d)
```

Iterative code

printing something different to the screen. Depending on the order in which they are called the output is different. Another example is a function that changes something to a global variable, and another function using that global variable to calculate a result: the order of execution matters for the result. If the change is something like "add 1 to the global variable", it also matters how many times the function gets called. For our purposes side-effects can have two consequences: (1) They introduce (hidden) data dependencies and hence impose a partial ordering on the function executions, and (2) The number of times a function with side-effects is called can matter.[1].

Looking back at figure 6.5 it becomes clear that our method breaks down if the argument transformation functions or the recursive case tests have side-effects. Indeed, in the iterative version argument calculations and recursive case tests are done more often than in the recursive version, in order to save memory. Storing the needed values in memory instead (a form of memoization) can make the required number of tests equal in both cases. In the case of side-effects memoization is needed to get correct results. In the absence of side-effects memoization is a design choice. The results of the functions with side-effects can be stored in a memoization table, and reused from the table when needed. In the absence of side-effects it is possible to choose how many calculation results to store in the memoization table, and until what depth in the call tree. The result is a systematic way of trading off memory space for calculations.

Storing and reusing values from a table requires the use of a key into the table. The amount of memory needed to store the argument values (or recursive case test results) can be chosen by defining a suitable key into the table. Concentrating on the argument transformation functions: if we chose as key into the memoization table e.g. a tuple (BaseBCounter,depth in the call tree), each node in the call tree gets a separate entry in the memoization table. This requires a lot of memory (see section 6.3.6.2 for a quantification), but has the advantage that full parallelization remains possible (as far as side-effects do not prohibit it). If on the other hand we use as key into the table only the depth in the call tree, entries in the table are overwritten while traversing the call tree, enforcing a sequential execution of the recursive algorithm or its iterative implementation. This is called in-place optimization. Defining a suitable key into the memoization table will thus allow to create a systematic trade-off between amount of memoization memory needed versus amount of parallelization opportunities.

The recursive algorithm provides implicit memoization by means of keeping the variables on the stack frame: the stack frame is used to hold the results as long as they are needed. The iterative implementation of the recursive

[1] Algorithms can be devised where the output depends on the absolute time at which it is called. It is safe to assume that such functions will not yield correct results in combination with our method. Such functions would also yield different results when used with other recursion removal methods, like tail recursion removal or incrementalization.

algorithm has to provide explicit memoization. The definition of the key into the memoization table trades off the required memory versus the potential for parallelizing the execution. For argument transformation functions with side-effects, the argument transformations in lines 19-22 and lines 33-35 of figure 6.15 are replaced with their memoized versions. The first time they are executed, their results are stored into the memoization table. The next time their results are needed, they are retrieved from the table. In the case of table entries that are overwritten, also a mechanism is necessary to decide whether an entry in the table is valid, or needs to be overwritten. While handling the back-tracking to activate intermediate functions, the argument values can also be taken from the memoization table, avoiding the expensive extra loop (line 30). For recursive case tests with side-effects, lines 16, 31, 36 are replaced with calls to a memoized version of getreccase. As with the argument transformations, also a mechanism is needed to invalidate entries in the table if in-place optimization is used. Implementation details are worked out in the code from [129].

6.3.5 Extension 5: functions with return values

The final extension presented here is one to handle recursive functions with return values. Similar as was done for memoization, the return values will be stored in memory and used when needed. Again each return value can either be stored in a separate memory location, or memory locations can be reused. The return values of different recursive calls in a given recursive case usually are combined into a result at the end of the recursive case. This combination of return values into a result is a form of intermediate function coming after the last recursive call. A possible scheme that works for the most general recursive algorithms that can be handled by our technique in its current form is one where for each level in the call tree b memory locations `rv[level][branch]`, each one large enough to hold one return value, are reserved. b here is the number of recursive calls in the recursive case (after making that number equal in each recursive case). Given a node at depth d of the call tree. Its n-th child ($0 \leq n < b$) writes its return value into `[d+1][n]`. While back-tracking the results written by the children of the n-th child of a node at depth d are combined to a new return value. More in detail: the results `rv[d+2][0], rv[d+2][1]`, ...,`rv[d+2][b-1]` can be combined to a result `rv[d+1][n]`

Consider as an example the calculation of a Fibonacci number. In its simplest form the result of two recursive calls is added together. We will not make an attempt to optimize this naive algorithm. We just show what our method results in. For normal Fibonacci calculations there are better algorithms than the one that will result from our transformation: a closed mathematical solution exists which yields the Nth Fibonacci number in O(1) time and space. The important point is that the same method applies to much more complex algorithms. The code at the ftp site [129] contains example code for an algorithm having leaves

Figure 6.16 An example algorithm

```
def fibo_rec_model(n):
 if fibobc0(n):
  return fibobcc0(n)
 elif fibobc1(n):
  return fibobcc1(n)
 elif fiborc0(n):
  r1 = fibo_rec_model(n-1)
  r2 = fibo_rec_model(n-2)
  return fibog2(n, r1, r2)
```

Recursive code

```
def fibo_it(n):
 nrbc, base, maxdepth, A = 2, 2, n, 0
 rv = [[0]*base for i in range(maxdepth)]
 itidx = BaseBCounter(base,maxdepth+1)
 for i in range(0,nrbc):
  if fibobc[i](n):
   return fibobcc[i](n)
 while itidx[maxdepth] == 0:
  tn, gcnt, btcnt = n, 0, 0
  while gcnt < len(itidx.digits) and\
            itidx[gcnt] == 0:
   gcnt = gcnt+1
  for d in range(maxdepth-1,-1,-1):
   level, prevA, A, bc =\
   maxdepth-d-1, A, itidx[d], 0
   tn = fiboF[A](tn)
   if gcnt >= d:
    for e in range(0,nrbc):
     if fibobc[e](tn):
      rv[level][A] = fibobcc[e](tn)
      bc = 1
      break
   if bc: break
  if bc:
   btcnt, hn = gcnt, n
   while btcnt<maxdepth-1 and\
    itidx[btcnt-gcnt+d]==itidx.base-1:
    l2 = maxdepth-btcnt-1
    for f in range(maxdepth-1,btcnt,-1):
     hn = fiboF[itidx[f]](hn)
    prevA = itidx[btcnt+1]
    (rv[l2][0],rv[l2][1],rv[l2-1][prevA]) =\
    fibogb2(hn,rv[l2][0],rv[l2][1],rv[l2-1][prevA])
    btcnt = btcnt+1
   if btcnt==maxdepth-1 and\
      itidx[btcnt-gcnt+d]==(base-1):
    (rv[0][0],rv[0][1],rv[0][0])=\
    fibogb2(hn,rv[0][0],rv[0][1],rv[0][0])
  itidx.incpow(d)
 return rv[0][0]
```

Iterative code

at non-constant depth, multiple base cases, multiple recursive cases and intermediate functions, where all functions have side-effects and return values. The code for the iterative implementation of this much more complex algorithm looks almost the same as the one shown above for the unoptimized Fibonacci calculation. It effectively combines all the extensions discussed in this text. Incidentally the code in figure 6.16 can be simplified and optimized by inlining function calls and propagation of constants. Measurements on C equivalents of the Fibonacci code results in somewhat worse performance for the iterative implementation of Fibonacci calculation (despite using full memoization and in-place optimization which results in the same time and space complexity as the recursive algorithm), indicating that in the most general cases considered in this chapter the transformation using our method as-is is not necessarily useful for reasons of performance. The difference in performance is due to the bookkeeping needed in the iterative version. Indeed, in the recursive version memoization is implicitly provided by the (run-time) stack. In the iterative version a key is explicitly used. Many optimizations are possible in the iterative version however. On similar applications, once they are in explicit loop nest form, large factors of gain in memory accesses and energy have been observed during source code transformation steps [37]. We expect that such optimizations will compensate for this loss but a quantitative evaluation is left for future work.

As part of future work the required bookkeeping can be further optimized. Even today this method can already be useful for algorithms with return values where the recursion proceeds to very deep levels: the iterative implementation by construction can not result in run-time stack overruns.

6.3.6 Trade-offs and performance impact of using the different extensions

We have discussed a basic idea and then presented a number of extensions to that basic idea to replace recursion with iteration. The question arises what impact the different extensions have in terms of number of calculations and memory usage. Calculating an accurate number of additions and multiplications is difficult because compiler specific. Instead, a high-level comparison is presented of the impact of different extensions on memory usage, number of argument transformation calculations and opportunities for parallelization. Without loss of generality, the comparison will be made for the case where all leaves live at a constant depth in the call tree. This assumption eases our performance analysis and represents a worst-case with respect to the considered costs associated to executing an implementation of an algorithm.

6.3.6.1 Execution time

First consider the comparison between a recursive execution and an iterative execution where no functions are called in between the recursive calls and in which no memoization is done. Consider a call tree where each node has either 0 or b children of depth $maxdepth$. The root node lives at depth 0. This situation was already depicted in figure 6.5. Each arrow in the call tree corresponds to a recursive call preceded by a transformation of argument values to a new value. Each arrow in the iterative implementation corresponds to an transformation of argument values, without the function call to itself. Summing how often each arrow is evaluated in the call tree is used as a rough estimate of the number of calculations that need to be done for a given implementation.

The recursive execution traverses the call tree in depth-first order. Each arrow is followed once. An arrow has depth d, if it connects a node from depth d to depth $d+1$. Then, at depth d, within our assumptions, exactly $b^{(d+1)}$ arrows exist. The number of arrows in the call tree therefore is $A_{rec}(b, maxdepth) = \sum_{d=0}^{maxdepth-1} b^{d+1} = b.\frac{b^{maxdepth}-1}{b-1}$ if $b \neq 1$, $A_{rec}(1, d) = maxdepth$. The number of leafs in the call tree is $L(b, maxdepth) = b^{maxdepth+1}$. If evaluating an arrow costs time t_A and evaluating a base case costs time t_B the recursive version is estimated to take $t_{rec} = A_{rec}(b, maxdepth) * t_A + L(b, maxdepth) * t_B$.

The iterative execution follows each path from root to leaf. An arrow at depth d therefore is evaluated $b^{(maxdepth-d-1)}$ times. Summing the number of evaluated arrows at all depths d gives $A_{it}(b, maxdepth) = \sum_{d=0}^{maxdepth-1}(b^{d+1}.b^{maxdepth-d-1})$ $= maxdepth.b^{maxdepth}$. The iterative version of the algorithm therefore is estimated to take $t_{it} = A_{it}(b, maxdepth) * t_A + L(b, maxdepth) * t_B$ time.

The ratio t_{it}/t_{rec} is close to one except when both b is small and t_A is smaller than or of same order of magnitude as t_B. For $b = 2$ (which represents a worst-case situation), the ratio t_{it}/t_{rec} is bounded by $maxdepth$. For $b = 2$ and $t_A = t_B$, $t_{it}/t_{rec} \approx maxdepth/4$. A detailed analysis of this ratio falls outside the scope of this chapter.

If a function call is made after the last recursive call, recall from figure 6.11 that an extra loop was needed to avoid using memory for storing arrow results. How does this influence the t_{it}/t_{rec} ratio? The extra loop causes an extra $A_{it2}(b, maxdepth) = \sum_{d=0}^{maxdepth-1}(d+1).b^d$

arrows to be evaluated. Then, $t_{it} = (A_{it}(b, maxdepth) + A_{it2}(b, maxdepth)) * t_A + L(b, maxdepth) * tB$. The ratio $t_{it}/t_{rec} \approx maxdepth/b$ if $t_A \approx t_B$, becoming smaller if $t_A << t_B$, and bounded by $maxdepth$ if $t_A >> t_B$.

When full memoization is done $A_{it}(b, d)$ becomes equal to $A_{rec}(b, d)$, and therefore $t_{it} = t_{rec}$. Differences between the recursive version and the iterative version that were neglected in this analysis are: (1) evaluating an arrow in the recursive version not only requires transforming arguments, but also a recursive call combined with the allocation and creation of a stack frame that holds all

variable values until the call returns. (2) similarly, evaluating an arrow in the iterative version with memoization not only requires transforming arguments, but also accessing a memoization table; the time to evaluate a base case in the iterative version is extended with the time needed to calculate (or update) a number base b representation of the path counter. This last cost can be very low at the expense of using some memory to store the already calculated digits. These differences can be significant when profiling code. For recursion, the error will be compiler and computer architecture specific and nothing much can be said about it in general. Presumably the implementation of recursion in production quality compilers will be done efficiently. The optimization of the iterative implementations presents possibilities for future work. It is certain that the current proof-of-concept implementations of the memoization tables and path counters are not the most optimal ones.

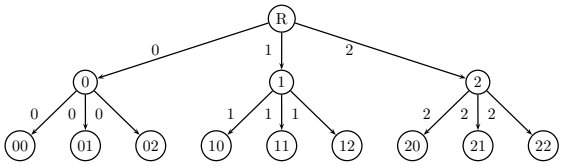
6.3.6.2 Memory footprint

Again we start with the situation where no function calls happen in between the recursive calls and in which no memoization is done. The recursive version of the algorithm needs to store stack frames during execution. The number of stack frames must hold at once is equal to the depth of the recursion. In the iterative implementation in principle no memory is needed proportional to the depth of the recursion or proportional to the number of recursive calls in the recursive case, since all needed values can be calculated on the fly. In the code shown in this chapter a BaseBCounter object was used which does need memory proportional to the maximal recursion depth to store the representation of the digits of the path counter in base b. This was merely done for speed optimization purposes, but can easily be replaced with div and mod calculations.

This situation does not change when a function is called after the last recursive call.

When memoization is done, either to handle side-effects or as a deliberate design choice, one is free to choose how many results to cache. A cached result from near the top of the call tree will be reused from memory more often than a result from near the bottom of the call tree. Indeed, the results of calculations associated to an arrow at depth d in the call tree are needed $b^{maxdepth-d-1}$ times during the execution of the algorithm. Since the results associated to arrows at depth $maxdepth - 1$ are needed only once, it does not make sense to memoize (cache) them. Deciding until what depth in the call tree results are cached, and from what depth results are recalculated on the fly gives a method to trade-off the required memory space versus the number of calculations done. At depth d, b^{d+1} arrows exist. Caching an arrow at depth d effectively reduces the number of arrow evaluations from $b^{maxdepth-d-1}$ to 1. It follows that by applying full memoization (except for the bottom layer of arrows in the call

Figure 6.17 Assignment of recursive algorithm with b=3 to 3 processors. Arrows are labeled with the processor they are assigned to.



tree which are evaluated only once) the same number of arrows are evaluated in the recursive version as in the iterative version. Caching the results associated to an arrow - if implemented efficiently - hence never needs more memory than the memory needed for holding the corresponding stack frame in the recursive version.

By storing each result associated to every arrow in a separate memory location, all results can be accessed simultaneously and therefore full parallelization remains possible, but at the cost of a large memory footprint: all results need to be available at the same time. In practice perhaps full parallelism is not needed because not enough processors are available. In that case memoization memory locations can be reused over time, leading to in-place optimization. This results in a smaller memory footprint but at the cost of reduced potential for parallelism. The overwriting of memoization memory locations indeed introduces ordering constraints because results that are required for correct execution are not available all the time. In the limit where full memoization is performed and then full in-place optimization is done, the amount of memory needed is proportional to $maxdepth - 1$. The bottom layer of arrows in the iterative version does not need to be cached leading to a slightly better memory footprint for the iterative implementation than for the recursive algorithm. And just like in the recursive algorithm the opportunities for parallelism are lost. For every extra processor in use, an extra amount of memory proportional to $maxdepth - 1$ is required.

6.3.6.3 Parallelism

Suppose n identical processors are available. Calculating an optimal assignment of arrows and base cases to processors for a given value of n, b and $maxdepth$ is not detailed here. In case n is a power of b, it is intuitively clear how to parallels in order to use $(maxdepth - 1) * n$ memory and $t_{it_par} = t_{it_seq}/n$ time. This is indicated in figure 6.17.

6.4 Practical results and discussion

We now present a case study demonstrating the result of applying the techniques explained in this chapter on an MPEG 4 visual texture coder (VTC). The VTC application described below is used as a driver application for research in the context of low-power embedded systems.

6.4.1 MPEG4 visual texture coding (VTC)

MPEG4 [111] promotes an object oriented approach to representing multimedia information. Scene objects consist of meshes onto which textures can be mapped. Efficiently transmitting and decoding textures in different qualities in information bitstreams requires advanced compression techniques. The code shown in figure 6.18 represents a time consuming inner loop of a visual texture coding decoder implementation and its iterative implementation. Some details of how to apply the method described in this text to convert from the recursive version to the iterative version can be found in [102] . In the iterative version the successive argument transformation calculations are visualized as a sum. As mentioned in section 6.3.2 the equations needed to evaluate the transformation of arguments along a path from root node to leaf node can be written down and manipulated (simplified). Just applying the method described in this chapter results in the following argument values in the base case for iteration j: $x_{bc} = \sum_{i=0}^{n-1} 2^i((j \operatorname{div} 2^{2i}) \bmod 2)$ and $y_{bc} = \sum_{i=0}^{n-1} 2^i((j \operatorname{div} 2^{2i+1}) \bmod 2)$. Looking at the sum shows that xbc is in fact a number that is got from j by leaving out all the odd bits of j, and interpreting the remaining bits as a binary number. Similarly, ybc is got from j by leaving out the even bits from j, and interpreting the remaining bits as a binary number. Such an operation can very easily and extremely efficiently be done in hardware, but is written as a sequential loop here (after doing some optimizations). No memoization is done. The fact that several arrows need to be re-evaluated every time is visible as the same terms in the sum being calculated in different loop instances. Adding memoization would mean to store these common intermediate results in memory, and reuse them from there.

6.4.2 Profiling results for vtc

It should be clear from the source code that the iterative version uses no stack to hold simulate recursion. It is interesting to compare the execution times of the recursive and the iterative version of the VTC decoder. Normally the base case calculation (`Decode_EQ_pixel` and `InvQuantPixel`) dominates the execution time, i.e. $t_B >> t_A$ and $b = 4 > 1$. According to the analysis in section 6.3.6.1, we expect the execution times to be more or less equal in the recursive and the iterative version.

Figure 6.18 Recursive MPEG4 texture decoder.

(recursive code)

```
def RecDecode(x,y,n):
 if n == 0:
  Decode_pixel_EQ(x,y)
  InvQuantPixel(x,y)
 else:
  RecDecode(x,y,n-1)
  if n-1==4: Check_eos()
  RecDecode(x+pow(2,n-1),y,n-1)
  if n-1==4: Check_eos()
  RecDecode(x,y+pow(2,n-1),n-1)
  if n-1==4: Check_eos()
  RecDecode(x+pow(2,n-1),\
            y+pow(2,n-1),n-1)
  if n-1==4: Check_eos()
```

(iterative code)

```
def ItDecode(x,y,n):
 for j in range(pow(4,n)):
  xbc = x
  ybc = y
  mask = 1
  f = j
  for i in range(n):
   xbc = xbc+(f & mask)
   f = f >> 1
   ybc = ybc+(f & mask)
   mask = mask << 1
  Decode_pixel_EQ(xbc,ybc)
  InvQuantPixel(xbc,ybc)
  if n > 4 and (j+1) % 256 == 0:
   Check_eos()
```

b=4, maxdepth=n

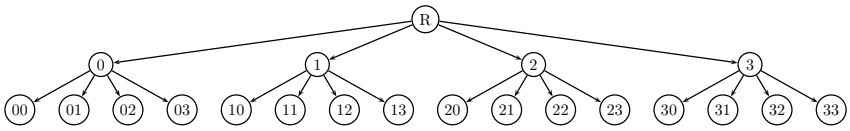
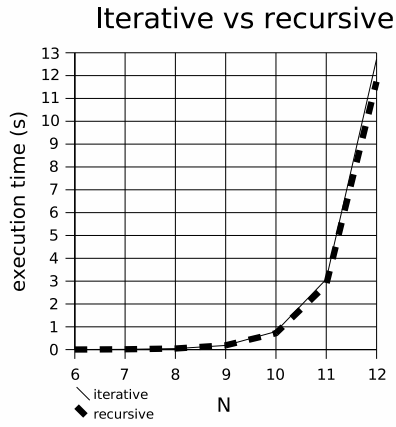


Table 6.2: Time measurement VTC decoder implementations

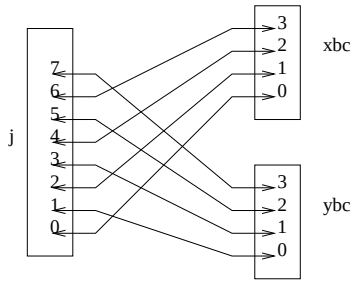
n	iterative (s)	recursive (s)
6	0.004	0.005
7	0.013	0.014
8	0.046	0.047
9	0.184	0.189
10	0.794	0.718
11	3.084	2.833
12	12.720	11.736



Time measurement is done using the unix command "time" on a mobile Pentium IV 1.6GHz with 256 MB ram system, on the C equivalent of the Python code shown in figure 6.18. Profiling the Python implementations and draw conclusions from the results is difficult due to hidden costs: e.g. the high cost of method invocation and array (list) access in Python. The former is expensive because of the need to type-check arguments at run-time. The latter is expensive because of built-in array bounds checking, and interpretation of negative indices as counting from the back to the front. The profiling shows that despite the recalculation of argument transformations the iterative version has comparable speed to the recursive version.

Knowing that in the context of VTC the recursion depth is never large (the examples we examined never needed a recursion depth larger than 5) it is concluded from the time measurement that the sequential iterative version of the VTC application in practice is better in terms of both execution time and (especially) memory than its recursive counterpart. If the base case calculations do not contain side-effects that introduce ordering constraints, each iteration of the iterative VTC is completely decoupled from the other ones, leading to a large potential for exploiting parallelism. The latter characteristic implies potentially large gains in energy consumption [225] on voltage-scalable multiprocessor platforms. This is of importance for example on battery-operated embedded platforms. An early version of the techniques described here and applied to the VTC example already lead to a energy saving of 20% [141]. As indicated, further improvement is possible in the sequential version also: mapping the `for i` loop on a specialized functional unit e.g. can do the complete loop using nothing but a reorganization of wires (see figure 6.19).

Figure 6.19 Hardware implementation of the argument transformations.



calculating xbc and ybc from j
example where j is represented using 8 bits

6.5 Conclusions

This chapter presented a new method for replacing non-linear recursion with iteration in imperative-style programs. Systematic trade-offs can be made between how many calculations are done, versus how much memory is used, and between how much memory is used versus how much parallelism is present. Performance gain is not guaranteed. The method is general enough to handle non-tail recursion and non-linear recursion. The methods presented in this chapter result in iterative implementations of recursively specified algorithms that do not use a stack to simulate the recursion.

A subset of the techniques explained in this chapter were demonstrated on a real-life MPEG4 VTC application, and on the Towers of Hanoi algorithm.

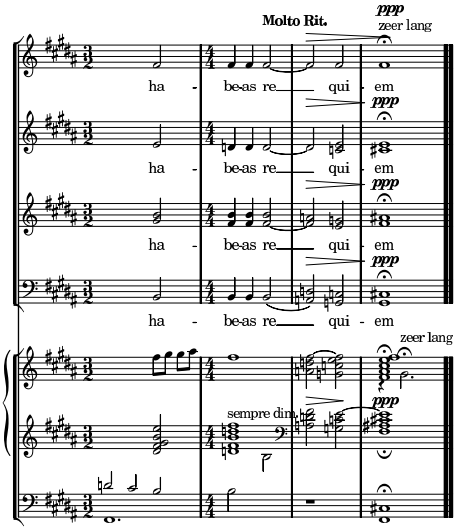
Mutual recursion has not yet been addressed in this work.

Part IV

Conclusions

Chapter 7

Conclusions and Future work



Maurice Duruflé, Requiem op. 9, part IX (In Paradisum)

7.1 Conclusions

Part I of this thesis gave an introduction and described motivation and context for this research. The research fits in the TCM design flow which is being instantiated the work of a team of PhD students mainly located at IMEC. TCM aims at reducing energy consumption in dynamic applications with user interaction, dynamic creation and deletion of tasks and data.

Part II of this dissertation has shown how applications are modeled using the MTG* and the gray box model in the task concurrency management (TCM) methodology. Chapter 2 first looked at related work with respect to modeling. For each of the aspects discussed it was explained how and why our model is different from the others. Some of these differences have technical reasons, while others are there to maintain compatibility with the existing MTG model.

The MTG* and gray box models briefly described in chapter 3 of this thesis can be seen as extensions of the older existing MTG model. Originally MTG* and gray box were used in TCM mainly for reasons of design-time scheduling and run-time scheduling preparation. This thesis on the other hand concentrated on the application modeling and source-to-source transformation stages of TCM.

Part III of this thesis shows how the MTG* and gray box models can be used to our advantage while doing other steps of the TCM design flow. In particular, it was shown to be useful while doing TCM transformations that aim for increasing the exposed parallelism, or creating freedom for parallelizing transformations. Two such transformations were looked at in detail in Part III:

- Chapter 5 uses the gray box model as a starting point for detecting certain patterns that enable semaphore removal.

  The importance of the semaphore removal technique, while advantageous for reducing resource consumption in itself, was explained to be mainly creation of more optimization freedom which can lead to significantly larger energy reductions Measurements on a JAVA application were presented to show how the semaphore removal technique can halve the number of byte code instructions that need to be executed in the method that contains the synchronization. The global impact was much smaller, namely around 4%. The reason is that synchronisation does not dominate this application. But as a side effect of the synchronisation removal more freedom is available for subsequent transformations. Due to limited time the exploitation of this freedom is considered future research.

- Chapter 6 describes a transformation in which recursion is replaced with iteration. We started by creating a framework in which to model recursion. After that, the framework was applied to the problem of replacing very general recursive algorithms with iteration. Mutual recursion is not

looked at and is considered future work. The replacement of recursion to iteration provides for systematic trade-offs with respect to the amount of memory that will be needed versus the number of calculations that will have to be done, the amount of memory that will be used versus the resulting parallelism. Using this freedom to gain energy is left as future work.

The framework results in a formula which allows symbolic simplification of the resulting iterative implementations. As a result the iterative implementations of recursive algorithms sometimes can yield simple solutions. As a result of the application of this symbolic simplification a closed formula resulted which finds the source pole in the j-th move in an Towers of Hanoi problem: $(j\&(j-1)) \bmod 3$ uniquely determines this pole. We also find a way to determine both source and destination pole from the binary representation of the move number. By applying the framework on an MPEG4 visual texture coding application, after conversion from recursion to iteration a simple hardware interpretation is found for the resulting operations in the inner loop of the iterative algorithm. Exploiting this could lead to very efficient implementations of VTC, but quantification of this gain is left as future work.

In the appendices that follow the following aspects are described:

- Appendix A presents a gray box model of QoS-enabled application. This gray box model served as input for the first TCM demonstrator application ever. The demonstrator was shown at the Design Automation Conference 2001 to lead to a significant energy saving (around 50% on a single ARM processor platform using voltage scaling) as compared to state of the art techniques. The experiments and results are extensively discussed in [222].
- Appendix B describes how the research in TCM transformations was initiated. It explains how potentially useful transformations, and a hypothetical transformation script were identified and classified by abstraction and re-instantiation in a different context some of the existing ideas in IMEC. The transformations discussed in this thesis so far have shown no indication that this theoretically derived script is incorrect, but it is not possible to proof it is correct in general.
- Because TCM is still in the process of being defined, ideas, concepts and definitions are bound to change over time. Appendix C brings together a number of currently known issues that can lead to reconsideration of existing TCM related ideas and methods. The same appendix also contains directions for future research.
- Appendix D finally presents some of the source code that was not presented in chapter 6 in order not to clutter the exposition.

7.2 Future work

At the start of our research only a few principles existed according to which TCM had to achieve an overall improvement of system cost. However, as soon as we started to try something it became clear that these principles were indeed just abstract principles, and it was far from trivial to actually start implementing something. During our research slowly well-defined problems started to emerge. Identifying and clarifying the problems indeed took most of our time and involved many discussions with the thesis promoters and members of the PhD team. Given the limited time and people resources, only a fraction of these problems could be solved by our PhD team. Some of the issues related to this thesis which are on the TCM TODO list are described here.

7.2.1 Future work related to modeling

Some heuristics currently are used to extract a gray box model from a given piece of source code. No systematic study, however, has been done to find out what the best thread node boundaries are. Some topics which deserve further attention include:

- In this thesis it was assumed that it was always possible to do task-level transformations on a suitably extracted gray box model. We did not address the question of how to extract this gray box model from a piece of C++ code. While some heuristics to do so exist and are used in IMEC, more research is needed to come to a formal definition of different kinds of thread nodes which are in use in TCM: scheduler thread node, transformation thread node and primitive thread node.
- While it is clear that choosing different thread node boundaries can influence the result system cost, no systematic characterization these effects exists.
  - It would be interesting to more explicitly characterize the influence of changing the thread node boundaries on the Pareto curves that result from the design time scheduler.
  - From the results of this characterization possibly better steering can be derived of the intra task transformations, which leads to improved overall system cost after applying TCM.
  - Reducing design time and improving time-to-market is also important. It should be investigated where tools can help to reduce the amount of work that needs to be done manually. Where tools can not help, investigate the trade off between the amount of design time needed to analyze and prepare a system for use with TCM and the

cost reduction that results from it. Can non optimal solutions yield acceptable results with a much reduced design time?

- How and when can Pareto curves that characterize thread frame performance be combined for several thread frames? Some initial material is presented in section B.4.2.
- Investigate which subsets of MTG* can be formalized and analyzed rigorously and what conclusions can be drawn from that analysis.

7.2.2 Future work related to transformations

This thesis described two transformations which can be used in the context of TCM. Possible extensions related to TCM transformations are virtually unlimited:

- The most obvious extension is to identify and work out new TCM transformations. When a number of transformations are available, it can be investigated how to order the transformations in time. We expect that the Pareto curves which are now mainly used for TCM design time scheduling can also be used to characterize the effect of TCM transformations, and to steer the order of TCM transformations.
- When we consider creating more run-time freedom by keeping several transformed versions of an algorithm available, investigate how to efficiently switch implementations. Should the different transformed versions of the code be combined in one function using conditions? Should efficient code be generated at run-time?
- Even on simple examples it is clear that non-Pareto optimal points on the design time scheduler generated Pareto curves can become Pareto optimal after transformation (and vice versa). The need exists to keep a number of non-Pareto optimal points available while transforming, leading to the concept of a *fat* Pareto curve. Currently it is not clear just how fat a Pareto curve should be.

More future work related to TCM, but not directly related to this thesis is summarized in appendix C.

Curriculum Vitae

Stefaan Himpe

Vroentestraat 20
B-3270 Scherpenheuvel-Zichem (Schoonderbuken)
België
tel: +32 477 36 79 29
e-mail: stefaan.himpe@gmail.com

Geboren in Brugge op 13 februari 1975

1980-1995 Lager onderwijs en secundair onderwijs, Latijn-Wiskunde, Sint-Jozefscollege, Tielt

1995-1996 Eerste kandidatuur Burgerlijk Ingenieur Gent

1996-1999 Industrieel Ingenieur Elektronica (Master of Science in de Industriële Wetenschappen), Katholieke Hogeschool Brugge-Oostende, eindwerk: "Muzieksynthese met behulp van digitale golfgeleidertheorie"

1999-2000 Burgerlijk Ingenieur Elektrotechniek (Master of Science in de Ingenieurswetenschappen), eindwerk: "Toepassen van de Data Transfer en Storage Exploratiemethodologie op een Advanced Audio Decoder en optimisatie voor een Trimedia processor"

2000-2006 Doctoraatsstudent in de Ingenieurswetenschappen, K.U.Leuven, Departement Elektrotechniek, Groepen ACCA, SISTA, ELECTA

2005- Software Ingenieur, ICOS Vision Systems NV, Haasrode

Part V

Appendices

Appendix A

Gray box model of MPEG4 QoS demonstrator

Figure A.1 Notation summary

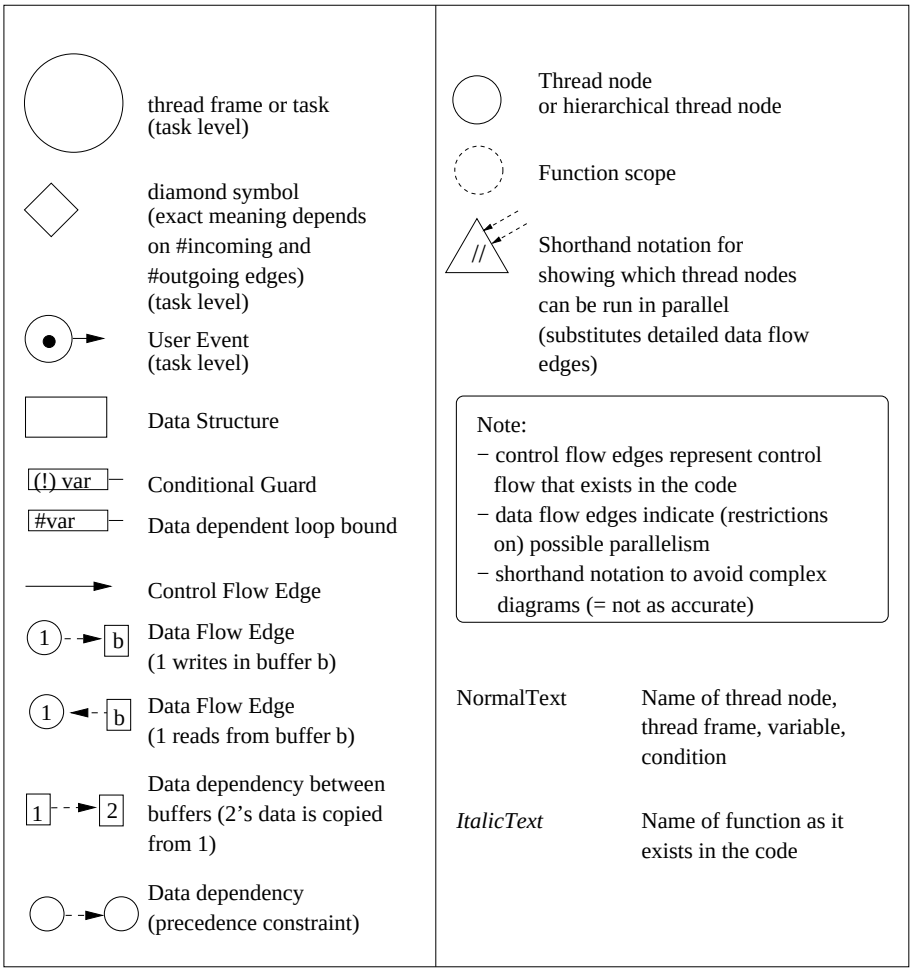


Figure A.2 MTG* task level model of MPEG4 QoS demonstrator.

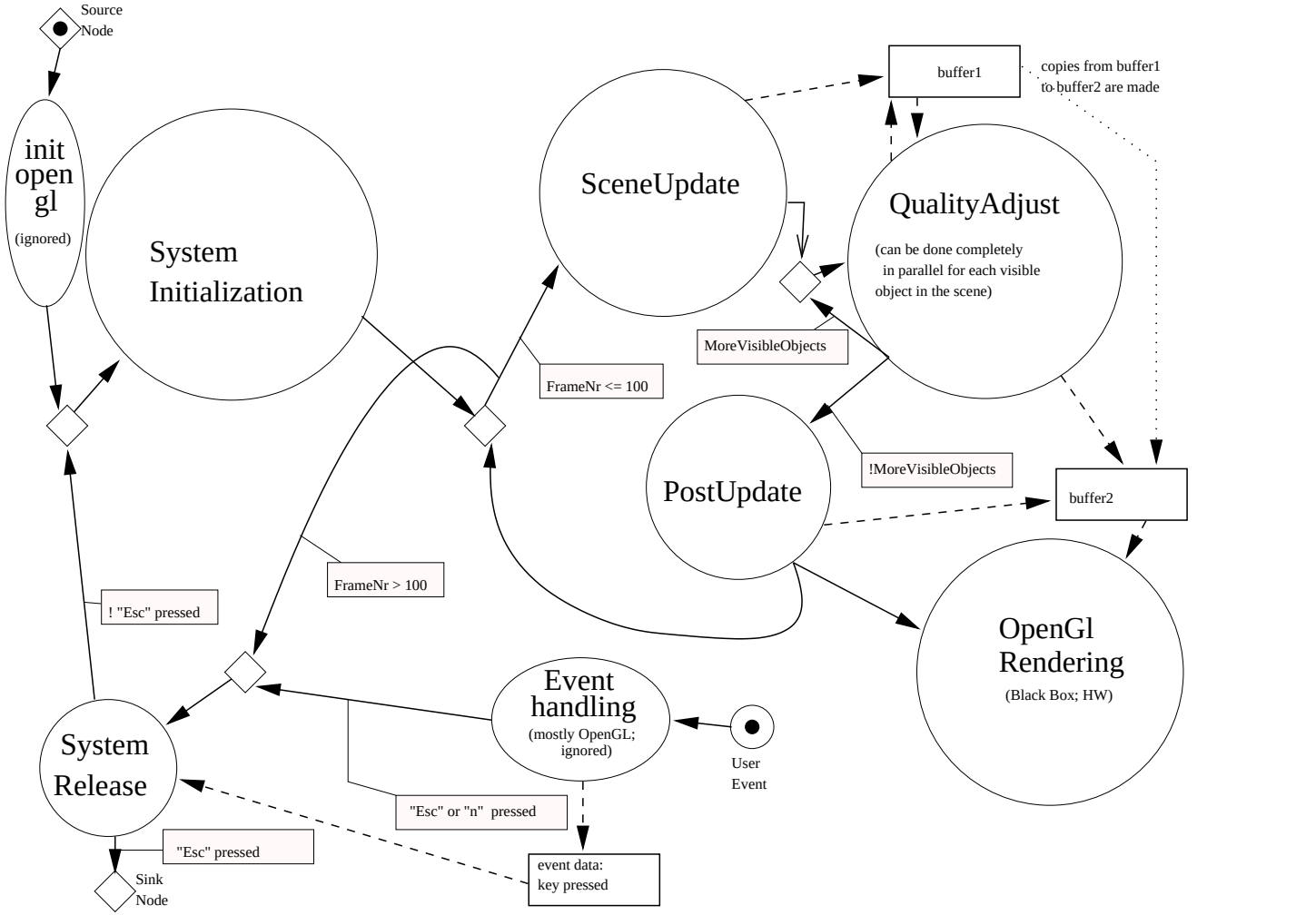


Figure A.3 Gray box model of Quality Adjust thread frame.

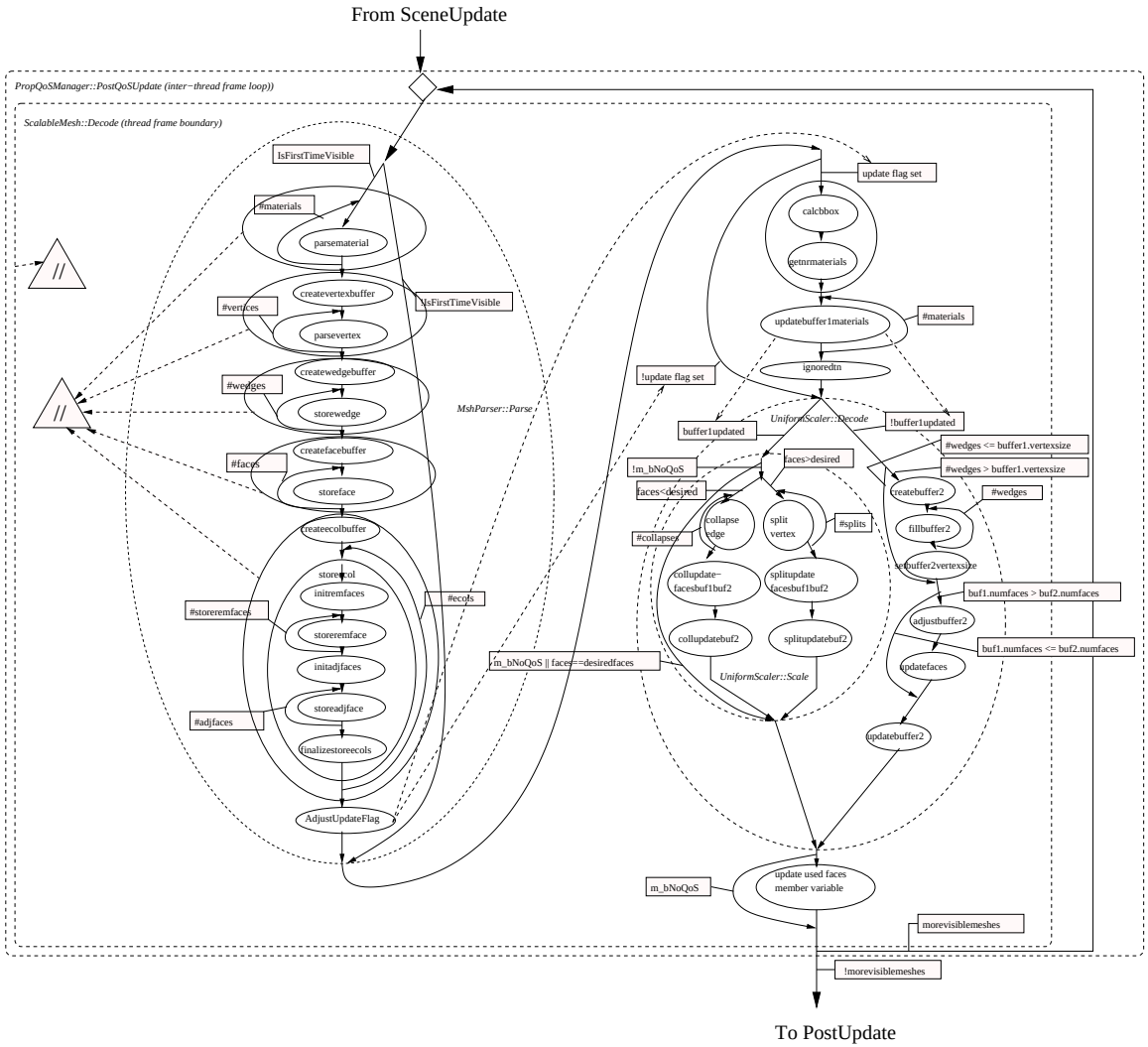


Figure A.4 Gray box model of Scene Update thread frame.

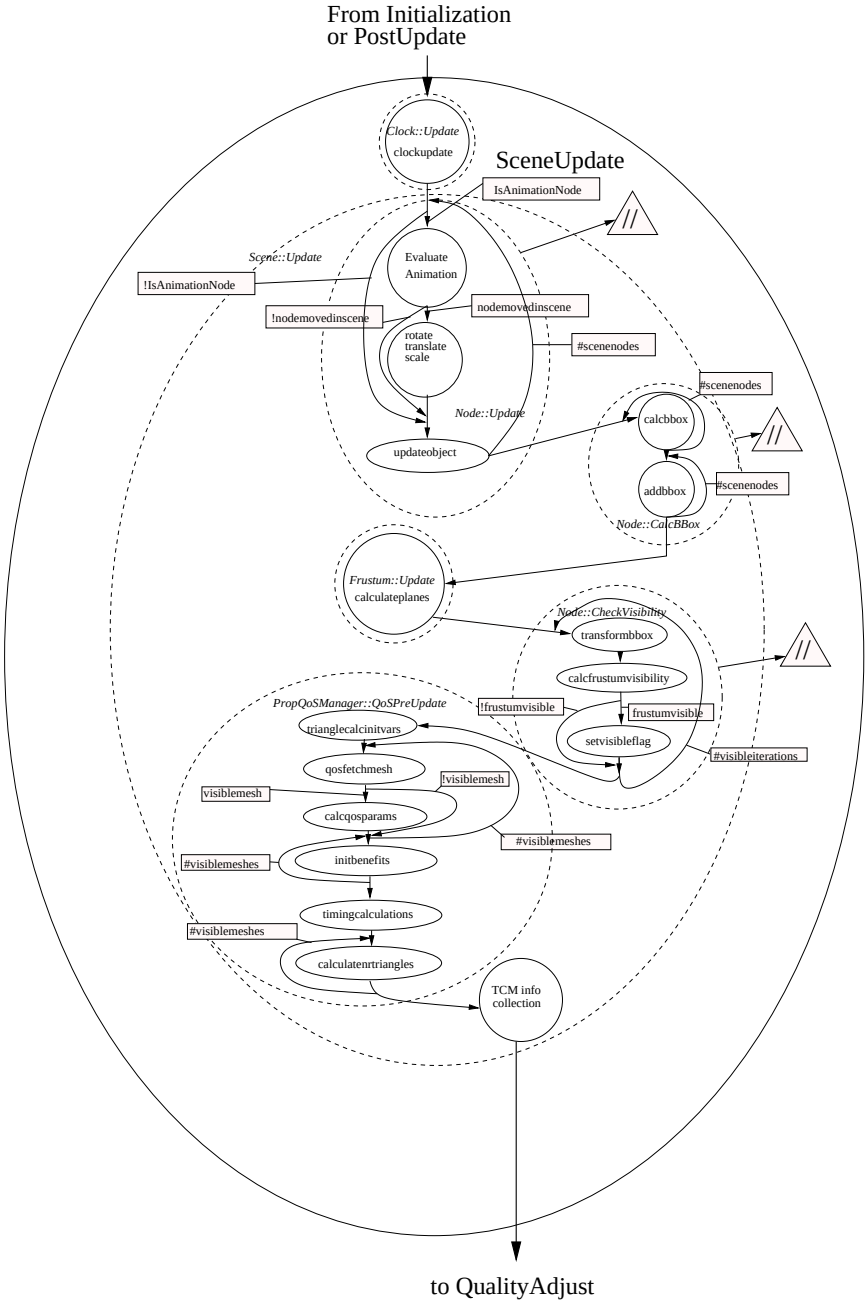


Figure A.5 Gray box model of Post Update thread frame.

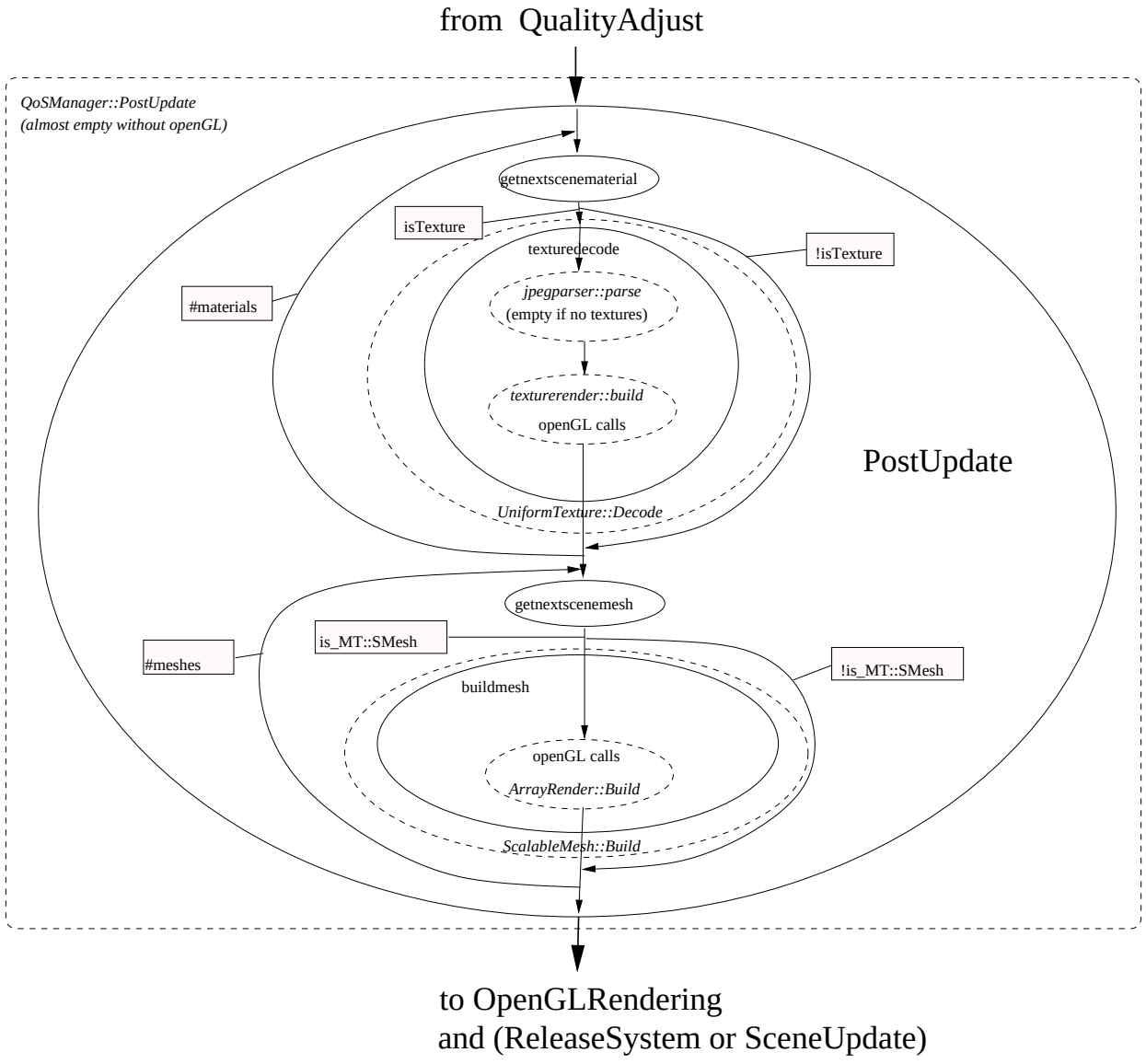


Figure A.6 Gray box model of Initialization thread frame.

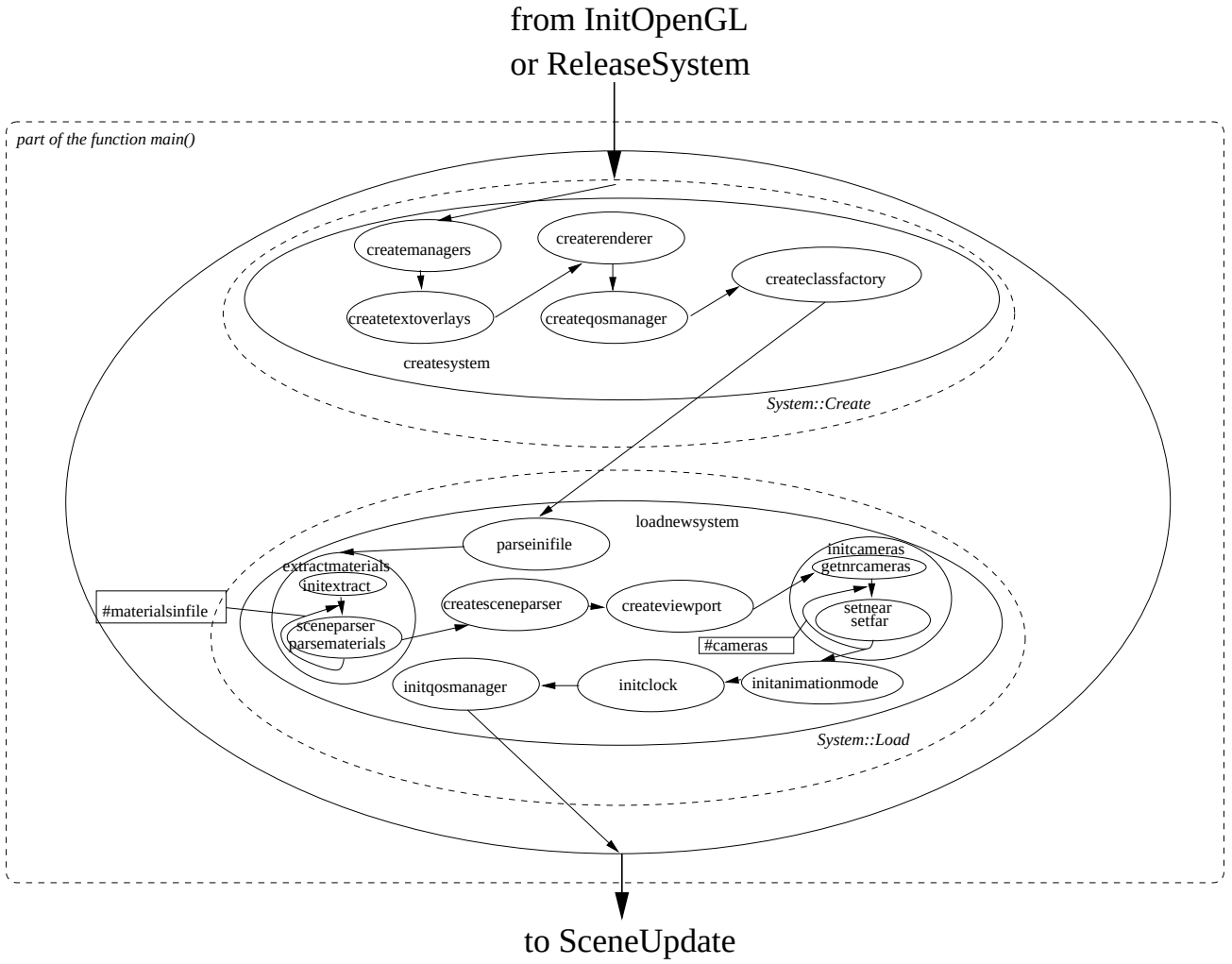
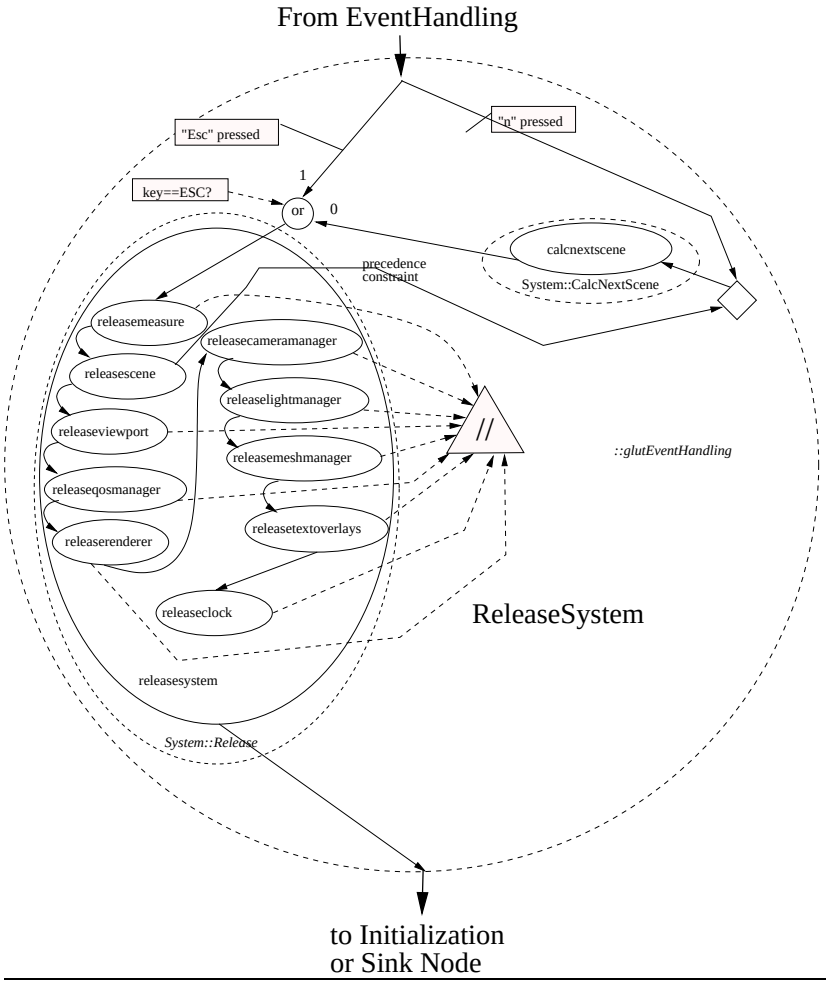


Figure A.7 Gray box model of System Release thread frame.



Appendix B

TCM transformation classification

B.1 Introduction

This chapter develops some ideas for using Pareto curves in multiple dimensions, and applications of such curves to estimating transformation impact. Chapters 5 and 6 develop and study effects of some transformations in more detail.

In this section a number of potentially useful TCM transformations are identified. In order to identify new transformations previous work done in IMEC to identify data transfer and storage exploration (DTSE) transformations [38] is reused. In this section the existing DTSE transformations are listed and classified according what purpose they serve. Starting from this abstract classification of the DTSE transformations, TCM transformations are instantiated that are expected to have beneficial effects in TCM context.

The transformations mentioned in the DTSE script are performed in two phases. The first phase is a pre-processing phase. In this phase the code is prepared for exploration by doing transformations that have a positive effect on all costs or which only introduce localized trade-offs with little impact not resulting in system-wide trade-off. As a result they can be more easily steered. Their actual implementation on the source code level can be quite complex though. The second phase is an exploration phase in which transformations are done which can introduce system-wide trade-offs. Transformation primitives used to implement pre-processing transformations are expected to be reusable while implementing exploration transformations.

The DTSE pre-processing phase proceeds in the following order:

1. abstraction transformations

   Abstraction in this context means hiding information which is not relevant at a certain moment. Examples of simplification transformations in DTSE include

   - hierarchical rewriting in 3 layers (layer 1= control flow, layer 2= loop nests, layer 3= rest)
   - hiding of undesired constructs (data dependent loop bounds, non-linear array index expressions)

2. freedom creation transformations

   Freedom creation transformations can temporarily make the implementation have a higher cost because they increase the freedom to trade-off costs later. Freedom created by freedom creation transformations is undone by subsequent bottleneck removal or trade-off transformations.

   - rewriting layer 2 in static single assignment form and replacing pointers with arrays
   - selective inlining

3. bottleneck identification and removal

   Bottlenecks are identified and removed if it is possible to remove them without introducing system-wide trade-offs. In some cases it is also possible to move bottlenecks to less harmful places.

   - data flow analysis and signal partitioning

4. complexity reduction

   In the complexity reduction step the amount of information to take into account in the next step of the script is again reduced. This reduction can lead to less optimal results but at the advantage of less information to take into account at once.

   - reduce model graph complexity for chains, recursions and conditions
   - weight-based removal of optimization candidates (classification of bottlenecks)
   - partition in independently optimized graphs (divide and conquer)

The DTSE exploration phase proceeds in the following order:

1. abstraction transformations

- global access removing data flow transformations like advanced resubstitution including conditional scopes, modifying computation orders by using algebraic laws, shifting delay lines through the algorithm
- global data dependence removal transformations like eliminating signals which are not needed for the result

2. freedom creation transformations
   - global loop and control flow transformations: improving data access regularity
3. bottleneck identification and trade-offs. Bottlenecks can be removed or moved to less harmful locations by means of look-ahead and software pipelining transformations.
   - global loop and control flow transformations: improving data locality
   - deciding on recomputation of result versus reusing it from memory
   - data reuse
   - storage cycle budget distribution
   - memory hierarchy layer assignment
   - data layout in memory
4. complexity reduction
   - no examples in DTSE, because all such transformations have been applied in the pre-processing part

For TCM similar steps can be instantiated leading to a TCM transformation script.

B.1.1 TCM instantiation of abstracted DTSE steps

For TCM it seems reasonable to follow a similar script as for DTSE. It is proposed to have the following steps. The same two-phase approach is proposed.

First a TCM pre-processing phase is proposed. During pre-processing transformations are performed that do not result in system-wide trade-offs. A more precise definition of pre-processing transformation which allows for automated classification of transformations is given in section B.3 on page 170.

1. abstraction transformations

- code clean-up: removing unused code or debug code and correction of certain software design decisions

  The software decisions that are cleaned up usually are the result of trying to keep the implementation maintainable. To keep code maintainable often a small price has to be paid in terms of costs like energy consumption and execution time. Since TCM eventually tries to optimize as good as possible, such overhead, where possible, should be removed. Some related work dedicated to OO languages is published in [77]. Higher-level decisions with respect to what algorithms, protocols or abstract data types are used are considered to be already fixed in this stage of the optimization process. Some related work is published in [227, 54].
- rewriting in two layers by extraction of initial gray box model from source code: thread frames and thread nodes

  Initial thread frame extraction is based on looking for dynamic task creation and deletion, acquiring and releasing of semaphores or other synchronization operations and event handling. Initial thread node extraction uses information gained by profiling to identify data dependent conditions and loop bounds that cause a large variation in execution time for different invocations. It also uses recognition of patterns in the source code that indicate potential for parallelization. Candidate thread nodes during an initial gray box extraction include function calls or loop bodies inside function calls. An ideal thread node can be accurately characterized by an average execution time and energy consumption, and doesn't hide loop-level or higher level parallelism inside.
- get an initial Pareto curve for each thread frame from the design-time scheduler. This Pareto curve can be used as reference to estimate the impact of transformations to come.

2. freedom creation transformations

   - removal of concurrency inhibiting constructs (if no system-wide trade-offs occur)

     Examples include semaphore removal for semaphores that can be removed by sequentialization (see chapter 5) or replacing dynamic creation of a task with a simple function call if the task always needs to be created.
   - program slicing techniques [216, 197, 24] can be used to reveal additional parallelism in thread nodes that dominate execution time.

3. bottleneck identification and removal

   - data dependence and side-effect analysis explicitly show the available parallelism and identify bottlenecks which cannot be parallelized as is

- reduction of critical path
- replace certain design patterns with other design patterns (see e.g. [25, 193])

4. complexity reduction
   - replacing task-creation chains (tasks creating tasks creating tasks...) with one-time task creation
   - weight-based selection of constructs being considered for further transformation
   - graph splitting: split large model in smaller separately optimized models

Next a system-wide TCM exploration phase is proposed. Transformations in this phase will typically decrease some costs at the expense of increasing other costs.

1. abstraction transformations
   - signal substitution
   - algebraic transformations on control flow
2. freedom creation transformations

   Transformations that increase the freedom for other transformations or improve the freedom of the design-time (e.g. revealing and increasing parallelism) and/or run-time scheduler (e.g. annotating possibilities for interleaving thread frames at run-time).
   - selective code expansion (inlining)

     Inlining is considered a freedom creation transformation because it increases code size but allows for specialization of a specific function invocation.
   - The recursion removal technique developed in chapter 6 of this thesis fits here. Without recursion removal a non-linear recursive function is contained in its own thread node. After recursion removal the iterations are shown explicitly and the thread node can be made finer-grained. Moreover the method does not introduce iterations dependencies that were not yet present in the original recursive specification. Systematic trade-offs that are supported by this transformation include needed memory versus amount of calculations and fully orthogonal to that also needed memory versus amount of parallelism.
   - replace certain design patterns with other design patterns (see e.g. [25, 193])

3. bottleneck identification and trade-offs

   - scope transformations
     - changing scope of semaphores
       Semaphore scope has an influence on synchronization overhead and available parallelism (see chapter 5). Trade-offs can result: coarser locking results in less synchronization overhead, but at the same time can also result in less parallelism.
     - choosing granularity of created tasks
       Consider a system running in a long loop, e.g. a slide-show viewer. Each picture in the slide-show needs decoding. Design options could include creating a new decoder task for each picture (can be tailored to the needs of a particular picture) or creating a decoder task only once for all pictures. Trade-offs include more run-time overhead (more tasks to manage) versus more run-time freedom (more possibilities to map such tasks on the platform).
   - trade-off task arithmetic complexity versus task control complexity
     - choosing block size for streaming computations
       Larger block sizes can avoid significant function calling overhead but require larger buffers to hold intermediate results and can have an impact on timing related issues like latency.
     - lazy evaluation as opposed to eager evaluation
       Lazy evaluation potentially removes many unnecessary calculations but requires insertion of a lazy evaluation implementation.
     - memoization can trade off memory space for calculations
       Memoization is a technique whereby the result of a calculation that needs to be redone often is stored and reused from memory[1].
   - software pipelining
     Pipelining increases the available parallelism but can introduce the need for larger buffers to hold intermediate results or the need for speculative execution and roll-back which can make energy consumption higher.

4. complexity reduction

   - determine final design-time scheduler thread nodes
     The topology and (execution time, energy consumption) characteristics of the thread node graphs determine the freedom of the TCM design-time scheduler and hence the quality of the resulting Pareto curve. A good quality Pareto curve is a Pareto curve with a large

[1] Decisions about using extra buffers in case of large data types are constrained by decisions previously made in task-level DTSE which normally precedes TCM.

(2,24)	(2.5,18)	(3,15)	(4,6)

Table B.1: 2D (t, e) example Pareto points

constraint (execution time) range with a number of Pareto points evenly distributed over the range and is shifted as much as possible towards the origin.

B.2 Pareto curves and surfaces

Chapter 4 already introduced the concepts of design space, design space point, Pareto optimality and Pareto point, cost and constraint. Now we look in more detail what a Pareto curve represents.

B.2.1 Anatomy of a Pareto curve

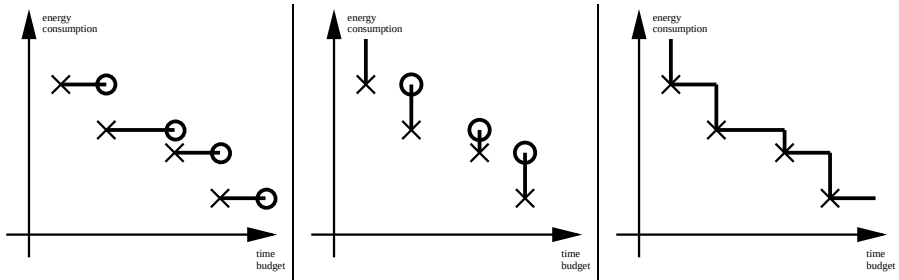
Consider the set of Pareto points characterized by (execution time, energy) in table B.1.

Suppose time budget is chosen to be a constraint and energy consumption to be a cost. This means time budget is chosen as large as allowed by constraints to minimize energy consumption. If the external timing constraint dictates that the execution at all times must finish within 2.7 time-units the design space point $(2.5, 18)$ is chosen thereby fixing the energy consumption to 18 energy-units. The same design space point (and hence the same energy consumption) is chosen for each timing constraint in the half-open interval $[2.5, 3[$. This leads to a horizontal surface (line) segment from $(2.5, 18)$ to $(3, 18)$. Mathematically speaking the point $(3, 18)$ is not included in this horizontal Pareto surface (line) segment. The point $(3, 18)$ is included, however, in another adjacent Pareto surface segment.

If on the other hand energy consumption is chosen to be a constraint and time budget to be a cost the situation changes. Now energy consumption can chosen as large as possible (decided by whether the device is powered by a battery, a fuel cell or from the electricity outlet) in order to minimize the execution time. Hence if at some point in the life cycle of the device the power supply imposes that energy consumption at all times should not exceed 16 energy units the design space point $(3, 15)$ is selected. The same design space point (and hence the same time budget) is chosen for each energy constraint in the half-open interval $[15, 18[$. This leads to a vertical surface segment from $(3, 15)$ to $(3, 18)$. See figure B.1 on the following page for clarification.

Both Pareto surfaces can be superimposed into one Pareto surface which still

Figure B.1 Influence of cost factor classification on Pareto surface



constraint: time; cost: energy | constraint: energy, cost: time | superposition of Pareto segments

has all necessary information. To find the best matching design space point it suffices to draw a line corresponding to the constraint. The design space point that belongs to the segment that is crossed is the one that best matches the given constraint. If no intersection is found no design space point exists that fulfills the constraint. If the constraint is $t < 2.7$ draw the vertical line $t = 2.7$, look at which surface segment is intersected and use the design space point that generates that segment. If the constraint is $e < 16$, draw the horizontal line $e = 16$, look in which segment an intersection occurs and use the design space point that generates this segment.

B.2.2 Multiple constraints and multiple costs

Introducing extra cost factors beyond execution time and energy consumption increases the number of dimensions of the design space points.

In the presence of multiple constraints a value needs to be chosen for each constraint without violating one of the constraints, and which results in the lowest total cost. In the case of multiple costs, an new cost function can be introduced which assign a relative importance to different costs and allows to find the best combination of values for the constraints.

In this thesis it is assumed that always at least one constraint and at least one cost axis are always present. Otherwise there are no trade-offs to be made. This assumption is realistic in embedded systems with real-time characteristics.

In the presence of multiple costs, after choosing a value for the constraint(s) sometimes multiple design space points can be found that satisfy the constraints. In such a case a design space point which minimizes $\sum_{i=1}^{N} \lambda_i c_i$, subject to $\sum_{i=1}^{N} \lambda_i = 1$ can be selected

- The c_i are the costs (not: constraints); λ_i are the relative importance of

(3,2,3)	(3,1,4)	(2,1,5)	(2,2,4)	(1,3,2)	(1,2,5)

Table B.2: 3D (t, e, s) example Pareto points

the costs.

- Each λ_i shows how important cost factor c_i is considered as compared to the other c_j's.
- The λ_i's can be decided at design-time or (at least in theory) varied at run-time. For fixed λ_i's the Pareto surfaces are fixed and can be calculated completely at design-time. If the λ_i's are varied at run-time the Pareto surfaces can change at run-time but the λ_i-values for which this happens and the different resulting Pareto surfaces can easily be calculated at design-time. This is clarified in the examples.

Example 8. *A three-dimensional example will illustrate the ideas presented here. Consider the Pareto-optimal design space points in table B.2. Each point can be interpreted as representing (execution time, energy consumption, code size) of a particular thread frame implementation.*

First consider execution time to be the constraint and energy and code size to be costs. This corresponds to how TCM would classify cost factors as constraints and costs. Choosing execution time to be constraint implies that execution time is chosen as large as possible in order to minimize some function of energy consumption and code size. More specifically the problem is:

> *Choose t (as large as possible as dictated by external factors) to minimize the function* $\lambda e + (1 - \lambda)s$*, where* $0 \leq \lambda \leq 1$ *is a measure for the* relative importance *of cost factor e with respect to cost factor s.*

For $\lambda = 0$ *no attention is paid to the energy consumption; optimization is done for code size only. For* $\lambda = 1$ *no attention is paid to the code size. All optimization is done for energy consumption only. For* $\lambda = 0.5$ *energy consumption and code size are considered equally important.*

The Pareto surface which shows for each time-value what energy consumption and code sizes are available is different for different values of λ*. To get some feeling for how the Pareto surface changes with* λ *the surfaces for different values of* λ *for the Pareto points in table B.2 is constructed.*

Since execution time t is the factor that is externally constrained and which is chosen as large as possible to minimize $\lambda e+(1-\lambda)s$ *four cases are distinguished:*

1. *execution time* < 1

- *this is not possible with the given design space points. No solutions exist with* $t < 1$.

2. $1 \leq$ *execution time* < 2

   - *in this case one of the points* $(1, 3, 2)$ *or* $(1, 2, 5)$ *is chosen depending on the value of* λ. *For* $\lambda = 0$ *the only factor that is important is code size hence the design space point* $(1, 3, 2)$ *is chosen. For* $\lambda = 1$ *the only factor that is important is energy consumption hence the design space point* $(1, 2, 5)$ *is chosen. The value of* λ *for which the choice of design space point switches is easily calculated from the cost function:*

$$\begin{aligned} 3\lambda + 2(1-\lambda) &= 2\lambda + 5(1-\lambda) \\ \lambda &= 0.75 \end{aligned}$$

   A special case (in this example) exists if value $\lambda = 0.75$*: both design space points have an equal cost. One can then choose to introduce extra constraints to select one design space point or just pick a point at random.*

3. $2 \leq$ *execution time* < 3

   - *In this case one of the points* $(2, 1, 5)$ *or* $(2, 2, 4)$ *is chosen. The value of* λ *for which the choice switches:*

$$\begin{aligned} 1\lambda + 5(1-\lambda) &= 2\lambda + 4(1-\lambda) \\ \lambda &= 0.5 \end{aligned}$$

   For $\lambda < 0.5$, *the point* $(2, 2, 4)$ *is chosen because (choose* $\lambda = 0$ *for easy calculation: this means that energy consumption is disregarded)* $4 < 5$. *Similarly, for* $\lambda > 0.5$, *the point* $(2, 1, 5)$ *is chosen because (choose* $\lambda = 1$ *for easy calculation: this means that code size is disregarded)* $1 < 2$. *For* $\lambda = 0.5$ *either point can be chosen.*

4. *execution time* ≥ 3

   - *In this case one of the points* $(3, 2, 3)$ *or* $(3, 1, 4)$ *is chosen. The value of* λ *for which the choice of design space point switches:*

$$\begin{aligned} 2\lambda + 3(1-\lambda) &= \lambda + 4(1-\lambda) \\ \lambda &= 0.5 \end{aligned}$$

   For $\lambda < 0.5$, *the point* $(3, 2, 3)$ *is chosen, for* $\lambda > 0.5$, *the point* $(3, 1, 4)$ *is chosen. For* $\lambda = 0.5$ *either point can be chosen.*

Now the 3D Pareto curve can be constructed. Four 'regions' dictated by execution time are distinguished.

- *In region* $(t < 1)$ *no solutions exist.*
- *In region* $(1 \leq execution\ time < 2)$ *three possibilities exist, depending on the value of* λ. *For* $\lambda < 0.75$ *the Pareto surface in this region looks like a line segment from* $(1, 3, 2)$ *to* $(2, 3, 2)$. *The segment is a line because it is generated by varying* t *only. For* $\lambda > 0.75$ *the Pareto surface in this region looks like a line segment from* $(1, 2, 5)$ *to* $(2, 2, 5)$. *For* $\lambda = 0.75$ *the Pareto surface in this region consists of both line segments.*
- *In region* $(2 \leq execution\ time < 3)$ *again three possibilities exist depending on the value of* λ. *For* $\lambda < 0.5$ *the Pareto surface in this region looks like a line segment from* $(2, 2, 4)$ *to* $(3, 2, 4)$. *For* $\lambda > 0.5$ *the Pareto surface in this region looks like a line segment from* $(2, 1, 5)$ *to* $(3, 1, 5)$. *For* $\lambda = 0.5$ *the Pareto surface in this region consists of both line segments.*
- *In region* $(execution\ time > 3)$ *three possibilities exist depending on the value of* λ. *For* $\lambda < 0.5$ *the Pareto surface in this region looks like a line segment from* $(3, 1, 4)$ *to* $(+\infty, 1, 4)$. *For* $\lambda > 0.5$, *the Pareto surface in this region looks like a line segment from* $(3, 2, 3)$ *to* $(+\infty, 2, 3)$. *For* $\lambda = 0.5$ *both line segments coexist.*

The same procedure can be repeated by choosing each of the other cost factors to be constraint while making the two others costs.

Classifying energy as constraint and classifying execution time and code size as cost makes sense in a situation where energy consumption is chosen as large as possible (constrained by possibilities for cooling or depending on whether the device is powered from a battery or a transformer) in order to minimize the execution time (the opposite of what TCM normally wants to achieve). Starting from energy again four regions can be defined (see the design space points in table B.2 on page 165). The extra cost function is $\beta.t + (1 - \beta)s$

1. $energy\ consumption < 1$
   - *No solution possible.*
2. $1 \leq energy\ consumption < 2$
   - *Either design space point* $(3, 1, 4)$ *or* $(2, 1, 5)$ *can be chosen.* β *for which one point or the other is chosen is easily calculated:* $3\beta + 4(1 - \beta) = 2\beta + 5(1 - \beta)$, *or* $\beta = 0.5$. *For* $\beta < 0.5$ *the preferred design space point is* $(3, 1, 4)$. *For* $\beta > 0.5$ *the preferred design space*

point is $(2, 1, 5)$. *For* $\beta = 0.5$ *both points have equal cost. Other constraints can then be used to decide on a final point (e.g. discard the point with the highest coordinate).*

3. $2 \leq$ *energy consumption* < 3

   - *Either design point* $(3, 2, 3)$ *or* $(2, 2, 4)$ *or* $(1, 2, 5)$ *is chosen. Special values for* β *can be computed:*

     (a) $3\beta + 3(1 - \beta) = 2\beta + 4(1 - \beta)$ *or* $\beta = 0.5$.
     (b) $3\beta + 3(1 - \beta) = 1\beta + 5(1 - \beta)$ *or* $\beta = 0.5$.
     (c) $1\beta + 5(1 - \beta) = 2\beta + 4(1 - \beta)$ *or* $\beta = 0.5$.

   - *for* $\beta = 0.5$, *the three points* $(3, 2, 3)$, $(2, 2, 4)$ *and* $(1, 2, 5)$ *are equivalent; for* $\beta < 0.5$ *point* $(3, 2, 3)$ *is optimal; for* $\beta > 0.5$, *point* $(1, 2, 5)$ *is optimal.*

4. *energy consumption* ≥ 3

   - *only point* $(1, 3, 2)$ *can be chosen.*

The Pareto surface hence looks as follows:

- *In region* (*energy consumption* < 1)

  - *No solution; empty halfspace.*

- *In region* ($1 \leq$ *energy consumption* < 2)

  - *Three possibilities depending on the value of* β. *For* $\beta < 0.5$ $(3, 1, 4)$ *is preferred so the Pareto surface segment is a line segment from* $(3, 1, 4)$ *to* $(3, 2, 4)$. *For* $\beta > 0.5$ $(2, 1, 5)$ *is the best choice. The Pareto surface segment is a line segment from* $(2, 1, 5)$ *to* $(2, 2, 5)$. *For* $\beta = 0.5$ *both points are equally suitable. Both line segments then coexist.*

- *In region* ($2 \leq$ *energy consumption* < 3)

  - *Three possibilities depending on the value of* β. *For* $\beta < 0.5$ *point* $(3, 2, 3)$ *is best. For* $\beta > 0.5$ *point* $(1, 2, 5)$ *is best. For* $\beta = 0.5$ *the three points* $(3, 2, 3)$, $(1, 2, 5)$ *and* $(2, 2, 4)$ *are equivalent.*

- *In region* (*energy consumption* ≥ 3)

  - *The Pareto surface segment is a line segment from* $(1, 3, 2)$ *to* $(1, +\infty, 2)$.

Finally choosing code size as a constraint and both execution time and energy consumption as costs the extra cost function is $\gamma t + (1-\gamma)e$. *The regions become:*

1. $code\ size < 2$
   - *no solution.*
2. $2 \leq code\ size < 3$
   - *only* $(1,3,2)$ *is possible; Pareto surface is line segment from* $(1,3,2)$ *to* $(1,3,3)$.
3. $3 \leq code\ size < 4$
   - *only* $(3,2,3)$ *is possible; Pareto surface is line segment from* $(3,2,3)$ *to* $(3,2,4)$.
4. $4 \leq code\ size < 5$
   - *either* $(3,1,4)$ *or* $(2,2,4)$ *can be chosen. For* $\gamma = 0$ *both points are equivalent; for* $\gamma > 0$ *point* $(2,2,4)$ *is preferred. For* $\gamma > 0$ *the Pareto surface is a line segment from* $(2,2,4)$ *to* $(2,2,5)$. *For* $\gamma = 0$ *the Pareto surface consists of two line segments; one from* $(2,2,4)$ *to* $(2,2,5)$ *and one from* $(3,1,4)$ *to* $(3,1,5)$.
5. $code\ size \geq 5$
   - *either* $(2,1,5)$ *or* $(1,2,5)$ *can be chosen. For* $\gamma < 0.5$ *point* $(2,1,5)$ *is chosen; the Pareto surface is a line segment from* $(2,1,5)$ *to* $(2,1,+\infty)$. *For* $\gamma > 0.5$ *point* $(1,2,5)$ *is chosen; the Pareto surface is a line segment from* $(1,2,5)$ *to* $(1,2,+\infty)$. *For* $\gamma = 0.5$ *both segments coexist.*

Example 9. *Now the construction of the Pareto surface for the case with* two constraints *and* one cost *is explained. In our three dimensional design space the situation is simplified because only one cost is present and hence no need exists to introduce extra cost functions to select one from multiple possible design space points.*

The number of cost factors corresponding to constraints imposes the dimension of the Pareto surface segments. Since two constraints are present now the Pareto surface segments are 2 dimensional (i.e. rectangles) – unlike the ones in the first part of this example which were only 1-dimensional. The reason is that for every combination of values for the two constraints a design-space point must be found by reading the Pareto surface.

Table B.3: t,e regions and resulting s for given design space points

time budget	energy consumption	Pareto surface segment corners
$t < 1$	any	no solutions
$1 \leq t < 2$	$e < 2$	no solutions
	$2 \leq e < 3$	$(1,2,5)(1,3,5)(2,3,5)(2,2,5)$
	$e \geq 3$	$(1,3,2)(1,+\infty,2)(2,+\infty,2)(2,3,2)$
$2 \leq t < 3$	$e < 1$	no solutions
	$1 \leq e < 2$	$(2,1,5)(2,2,5)(3,2,5)(3,1,5)$
	$2 \leq e < 3$	$(2,2,4)(2,3,4)(3,3,4)(3,2,4)$
	$e \geq 3$	$(1,3,2)(1,+\infty,2)(3,+\infty,2)(3,3,2)$
$t \geq 3$	$e < 1$	no solutions
	$1 \leq e < 2$	$(3,1,4)(3,2,4)(+\infty,2,4)(+\infty,1,4)$
	$2 \leq e < 3$	$(3,2,3)(3,3,3)(+\infty,3,3)(+\infty,2,3)$
	$e \geq 3$	$(1,3,2)(1,+\infty,2)(+\infty,+\infty,2)(+\infty,3,2)$

The case where execution time and energy consumption are considered to be constraints and the code size is accepted as a resulting cost is presented. The Pareto segments are calculated in table B.3. Starting from the design space points a number of constraint regions are extracted. In each constraint region the design space point with lowest cost is chosen.

Similar surfaces can be derived by classifying two other factors as constraints and choosing the third one to be cost. The Pareto segments are calculated in tables B.5 on the next page, B.4 on the facing page and B.5 on the next page. A picture of the resulting Pareto surface can be found in figure B.2 on page 172.

B.3 Transformation classification

During pre-processing the source code is transformed in such a way that none of the considered costs become worse. Functional behavior can potentially be changed as long as the expected end results are still the same. Applying this in a TCM context implies that the Pareto curves as derived by the TCM design-time scheduler shift towards the origin. TCM pre-processing transformations were previously defined as transformations that do not globally trade-off two or more costs. Another way of putting it: pre-processing transformations perform advanced cleaning up of an initial implementation. Looking at the Pareto curves before and after the pre-processing transformation reveals no intersection between the two. The shape of the Pareto curves before and after pre-processing transformations is not necessarily identical because smaller local trade-offs are allowed as long as they do not result in intersecting Pareto curves. Moreover the transformations often create new possibilities to map the thread nodes on

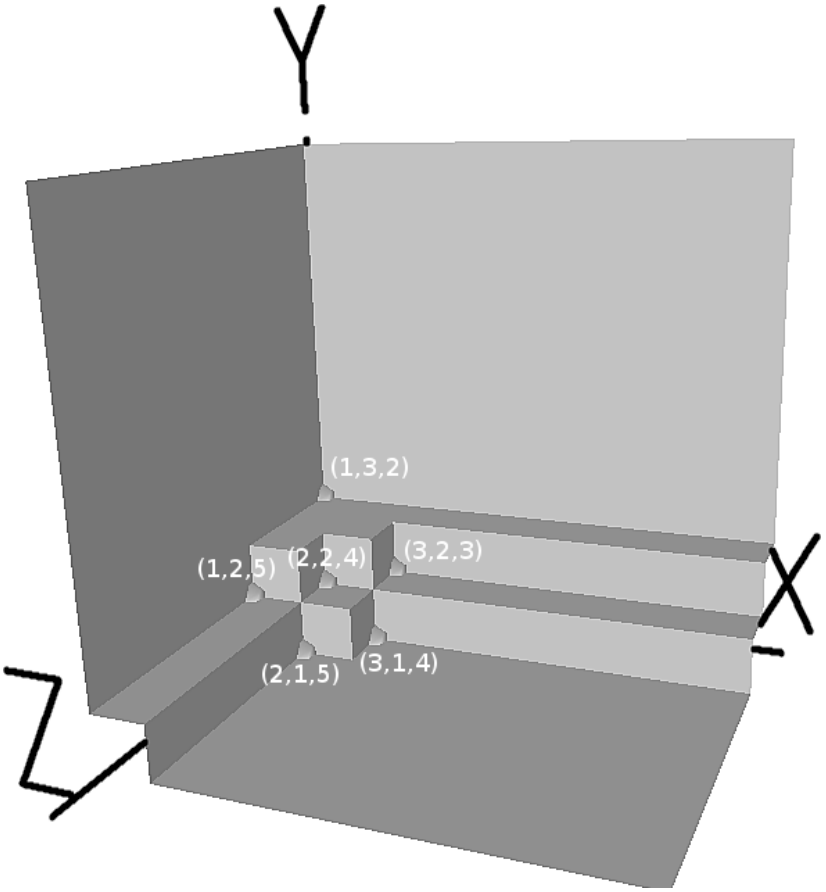
Table B.4: t,s regions and resulting p for given design space points

time budget	code size	Pareto surface segment corners
$t < 1$	any	no solutions
$1 \leq t < 2$	$s < 2$	no solutions
	$2 \leq s < 5$	$(1,3,2)(1,3,5)(2,3,5)(2,3,2)$
	$s \geq 5$	$(1,2,5)(1,2,+\infty)(2,2,+\infty)(2,2,5)$
$2 \leq t < 3$	$s < 2$	no solutions
	$2 \leq s < 4$	$(1,3,2)(1,3,4)(3,3,4)(3,3,2)$
	$4 \leq s < 5$	$(2,2,4)(2,2,5)(3,2,5)(3,2,4)$
	$s \geq 5$	$(2,1,5)(2,1,+\infty)(3,1,+\infty)(3,1,5)$
$t \geq 3$	$s < 2$	no solutions
	$2 \leq s < 3$	$(1,3,2)(1,3,3)(+\infty,3,3)(+\infty,3,2)$
	$3 \leq s < 4$	$(3,2,3)(3,2,4)(+\infty,2,4)(+\infty,2,3)$
	$4 \leq s < 5$	$(3,1,4)(3,1,5)(+\infty,1,5)(+\infty,1,4)$
	$s \geq 5$	$(3,1,4)(3,1,+\infty)(+\infty,1,+\infty)(+\infty,1,4)$

Table B.5: p,s regions and resulting t for given design space points

energy	code size	Pareto surface segment corners
$e < 1$	any	no solutions
$1 \leq e < 2$	$s < 4$	no solutions
	$4 \leq s < 5$	$(3,1,4)(3,1,5)(3,2,5)(3,2,4)$
	$s \geq 5$	$(2,1,5)(2,1,+\infty)(2,2,+\infty)(2,2,5)$
$2 \leq e < 3$	$s < 3$	no solutions
	$3 \leq s < 4$	$(3,2,3)(3,2,4)(3,3,4)(3,3,3)$
	$4 \leq s < 5$	$(2,2,4)(2,2,5)(2,3,5)(2,3,4)$
	$s \geq 5$	$(1,2,5)(1,2,+\infty)(1,3,+\infty)(1,3,5)$
$e \geq 3$	$s < 2$	no solutions
	$2 \leq s < 3$	$(1,3,2)(1,3,3)(1,+\infty,3)(1,+\infty,2)$
	$3 \leq s < 4$	$(1,3,2)(1,3,4)(1,+\infty,4)(1,+\infty,2)$
	$4 \leq s < 5$	$(1,3,2)(1,3,5)(1,+\infty,5)(1,+\infty,2)$
	$s \geq 5$	$(1,3,2)(1,3,+\infty)(1,+\infty,+\infty)(1,+\infty,2)$

Figure B.2 3D pareto surface calculated from the given design space points.



the platform and hence new Pareto points can be created. So a new definition for TCM pre-processing transformations can be given.

A *TCM pre-processing transformation* is a transformation that does not result in a new Pareto curve intersecting with the old Pareto curve.

A second phase in the DTSE script is the application of transformations that introduce system-wide trade-offs. A system-wide trade-off involves making one cost better at the expense of another cost. In a TCM context this causes the Pareto curve after transformation to be rotated as compared to the Pareto curve before transformation. Comparing the Pareto curves before transformation and after transformation reveals one (or more) intersection points.

A *TCM exploration transformation* is a transformation whose Pareto curves before and after transformation show an intersection. See figure B.3 on the next page for an illustration.

Note that practical Pareto curves necessarily are discrete step functions because the design-time scheduler can only generate a finite amount of different mappings of thread nodes on a real platform.

B.4 Getting transformation feedback from Pareto curves

B.4.1 Cdfg transformations

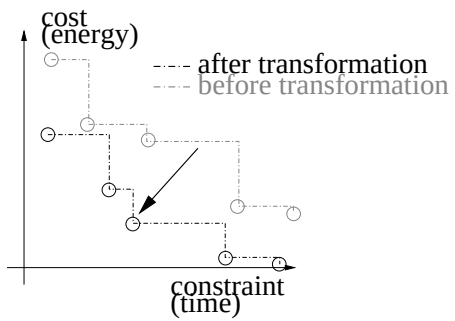
First the intra thread frame transformations are looked at (cdfg transformations). Getting an idea of the impact of one or more transformations involves looking at the Pareto curves extracted by the design-time scheduler. Each point on such a Pareto curve has coordinates indicating the costs (e.g. energy) associated to a certain value for the constraints (e.g. execution time). Each point on the Pareto curve corresponds to a different mapping of the same thread node graph inside a given thread frame on the target platform.

B.4.1.1 Pre-processing transformations

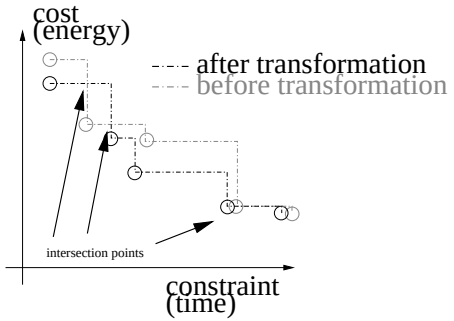
Pre-processing transformations were defined as transformations that result in a new Pareto curve not intersecting with the old Pareto curve. To get an estimation of the effect of a pre-processing transformation it is proposed to calculate the average amount of shifting towards the origin. A practical way to get an estimate is to calculate the displacement of the center of gravity of the Pareto points on the old Pareto curve to the new Pareto curve:

Given two sets of Pareto points P_1 and P_2 corresponding to the Pareto curve

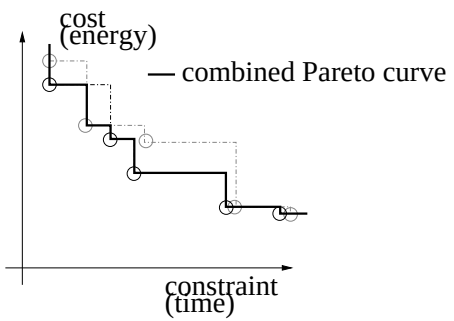
Figure B.3 Pre-processing and transformation phases in TCM transformation flow.



Pre-processing transformation:no intersection



Trade-off transformation: intersection



Combine Pareto curves

before pre-processing transformation and after pre-processing transformation respectively. Suppose P_1 contains N Pareto points and P_2 contains M points. Each Pareto point is characterized by its coordinates corresponding to constraints and costs. As an example, a Pareto point $\overrightarrow{p}$ from Pareto curve P_1 with coordinates $(3ms, 10mW)$ corresponds to a mapping of the thread nodes inside the thread frame from which P_1 is derived to the target platform which executes the thread frame in at most 3ms and which requires at most 10mW of energy. The center of gravity of P_1, $\overrightarrow{z}_{P_1}$, can be calculated as well as the center of gravity of P_2, $\overrightarrow{z}_{P_2}$.

$$\overrightarrow{z}_{P_1} = \frac{\sum_{i=1}^{N} \overrightarrow{p}_{1i}}{N} \qquad \overrightarrow{z}_{P_2} = \frac{\sum_{j=1}^{M} \overrightarrow{p}_{2j}}{M} \tag{B.1}$$

The displacement of the center of gravity is given by $\Delta\overrightarrow{z} = \overrightarrow{z}_{P_1} - \overrightarrow{z}_{P_2}$. Each coordinate of $\Delta\overrightarrow{z}$ gives an indication of the effect of the pre-processing transformation on that particular cost.

The transformation is said to have a positive effect iff all coordinates of $\Delta\overrightarrow{z}$ are positive or zero and at least one coordinate is not zero.

B.4.1.2 Exploration transformations

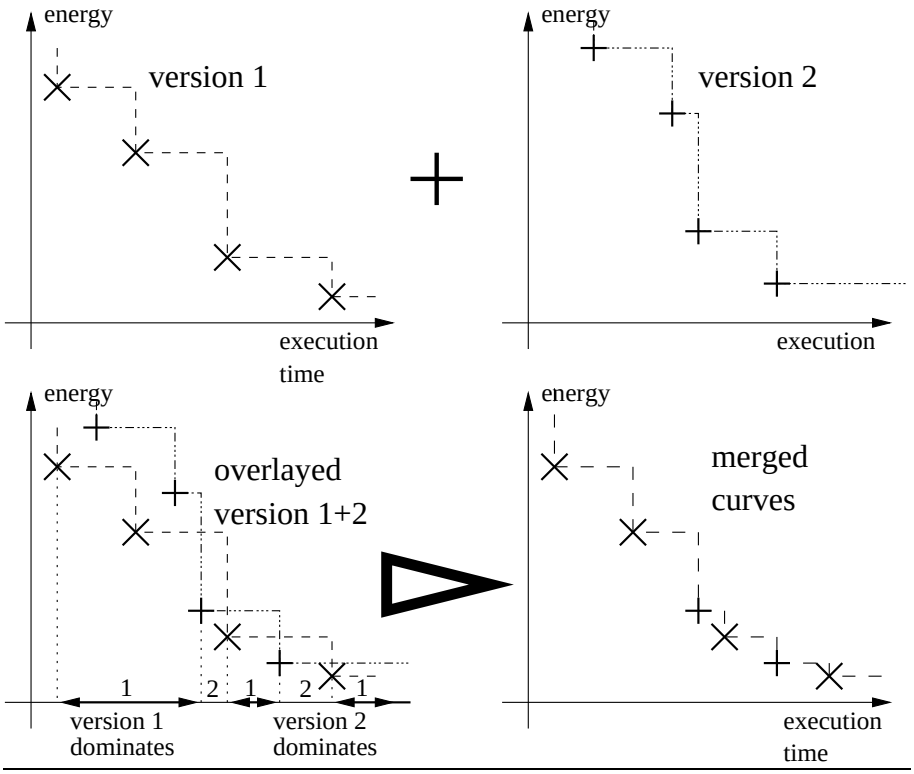
Exploration transformations were defined as transformations that introduce intersection points. Exactly those intersection points are interesting to know. They show the boundaries of the regions where either the original or transformed code performs equally or better. By keeping the two versions of the code available the best version can be chosen at run-time depending on the workload detected in the system. In terms of the Pareto curves all Pareto points from both curves are merged in a new Pareto point cloud and then non Pareto optimal points are filtered away from the cloud. This results in a combined Pareto curve. See figure B.4 on the following page for an illustration.

B.4.2 Task-level transformations

Estimating the effect of a task-level transformation by means of Pareto curves is slightly more involved. Task level transformations change the task-level of an MTG* model. This can result in a different number of thread frames or a different granularity of thread frames. The Pareto curves are extracted by the design-time scheduler for each thread frame separately.

One possibility to use the Pareto curves for transformation impact estimation is to first add up the Pareto curves of all thread frames into a system-wide global Pareto curve. Additional complications as compared to the intra task case exist.

Figure B.4 Intersection points and Pareto curve merging.



First, not every thread frame is active at the same time. To get a view of the impact of a task level transformation it is necessary to define a number of inter-thread frame scenarios — similar to how scenarios help handle data dependent behavior inside thread frames. Such scenarios indicate which thread frames can be active at the same time. Only Pareto curves of simultaneously active thread frames need to be merged to get a meaningful system-wide Pareto curve. A second problem that is still under investigation is how to handle different thread frames that share functional units on the platform. Such thread frames cannot be run in parallel due to platform constraints. Similar problems exist when two thread frames need to communicate data. Both problems cause simple addition of Pareto curves to get a global curve to generate inaccurate or incorrect results.

Combining thread frame Pareto curves in a system-wide Pareto curve and using such curves to do estimation of the impact of task-level TCM transformations is a topic of further research.

B.5 Conclusions

In this chapter it was first shown how a possible TCM script was derived from the existing DTSE script. We first abstracted the existing DTSE script and then reinstantiated a corresponding TCM script. Proof that this resulting TCM script is indeed correct is future work. The work presented in this thesis does not contradict this TCM script but also is not sufficient to prove it.

In a second part of the chapter some explanation about constructing and using multi-dimensional Pareto curves and surfaces was given. Application of Pareto curves to the problem of transformation impact estimation was proposed. Because not enough transformations have been worked out yet it is too early to try and see if this is a promising way of working.

Appendix C

Issues and future work

This appendix compiles a number of issues that have come up in TCM research recently. A list of future work is also given, both as related to this thesis and to TCM in general.

C.1 Current issues

This section discusses a number of issues that have come up in TCM research recently. *The issues in this section are not my contributions.* They serve as TCM documentation and can be skipped on first reading.

C.1.1 Application domain type

T-DTSE and TCM can be used in two types of application domains.

1. type 1 dynamic applications with

   - multiple sequentialisable tasks
   - data dependent loop bounds (while loops) with not too large range of possible loop bound values
   - real-time constraints (e.g. frame rate)

   Such applications are usually frame-based, i.e. there is a notion of a time frame in which some results have to be ready. As an example: in a video decoder application with a frame-rate of 25 images/s an image must be ready for display every 40i ms.

Typically such applications are mapped on platform completely at design-time by making a worst-case design. The advantage of this approach is that the application is guaranteed to finish in time without the need for run-time managing of tasks. The disadvantage of this approach is that at any point in time the platform is usually over-dimensioned and hence too energy-consuming.

In TCM, at *design time* applications are analyzed to identify thread frames. For each thread frame one (or more: see section C.1.3 on the next page about scenarios) Pareto curves are extracted. Each Pareto point represents a Pareto optimal mapping of thread nodes inside the thread frame to the platform with respect to the cost factors execution time and energy consumption. The Pareto curves may be augmented with more cost factors (see appendix B about multi-dimensional Pareto surfaces and section C.1.3 on the facing page about integration of TCM with quality of service.) At *run-time* a suitable Pareto point (= mapping) is chosen depending on the workload in the system. As a result a slower lower power solution can be chosen if the system workload is not high, or a faster high power solution can be chosen if more tasks need to complete in time. The execution time is parametrized by the values of the data dependent loop bounds which are known (possible estimated at the start of the thread frame) at run-time.

In this type of applications, at run-time, once a working (Pareto) point is chosen for a thread frame the thread frame continues to work at that point until it finishes. Thread frame working points are not changed while they are running.

2. type 2 dynamic applications with

   - multiple concurrent tasks that are event triggered
   - potentially very data dependent (non-manifest loop bounds that take values from a large range of possible values, and which have large impact on resulting execution time)
   - combined with soft timing and quality of service management

   Different tasks can have different frame rates, and tasks can be triggered by user interaction (so essentially, there is no concept of frame rate anymore).

   In a traditional approach the tasks are scheduled by an operating system, often using a system of priorities. Providing real-time guarantees in such environments is not easy. It typically requires mechanisms like pre-emption which cause context switching overhead at run-time.

   In TCM, at design-time applications are analyzed and partly explored. At run-time working points are chosen for the tasks that need to execute. Choosing a Pareto working point for a thread frame is something

more general than assigning a traditional task priority. It allows to better exploit application and platform knowledge at an acceptable run-time overhead. Given the very dynamic nature of this second class of applications some of the exploration is postponed until run-time. These are the applications where subscenarios are expected to be necessary (see section C.1.3).

The biggest difference for TCM between this class of applications and the first class of applications is that to take into account the very dynamic behavior of type 2 applications, working points of thread frames are allowed be shifted while the thread frames are executing. Allowing this shift will increase run-time overhead and it yet remains to be seen if the gains will be high enough to justify this extra overhead.

C.1.2 Thread frame boundaries

With the definitions of thread frame in this thesis blocking reads introduce a thread frame boundary because of the implied synchronization. In applications where blocking reads are used extensively this results in a fine-grained task-level model which easily results in hundreds of trivial Pareto curves.

The original solution to this problem was to group such fine-grained thread frames in thread clusters. The draw-back of this solution is that a very worst-case behavior of the thread cluster has to be assumed to get some meaningful performance estimates, needed to select a working point.

The current solution to this problem is to allow blocking reads inside a thread frame without introducing a thread frame boundary, and to let the run-time scheduler re-evaluate a thread frame's working point while it is running if needed. See section C.1.3, especially the part on subscenarios.

C.1.3 Scenarios and subscenarios

C.1.3.1 Intra thread frame scenarios

As is mentioned in the thesis data dependencies do not introduce thread frame boundaries. Yet data dependent values can activate code which increase execution time with an order of magnitude. A data dependent loop bound causes the execution time of a loop to vary with the actual value of the loop bound. When making a performance model of a thread frame such data dependencies must be taken into account.

In the gray box model data dependencies that can cause a large variation in execution time are visualized in between the design-time scheduler thread nodes. The performance of a thread frame can be estimated from the performance and interconnection of the thread nodes, weighted by these visualized data

dependent values.

In principle each combination of data dependent parameter values results in a different execution time, energy consumption, and possibly other costs which are taken into consideration. Fortunately it is not needed to discern all possible combinations. Based on profiling the data dependent values can be lumped in ranges. Selecting a range of values for each visualized data dependent value defines an intra thread frame scenario. The idea is to apply a clustering/pruning of possible data dependent values and thus reduce the number of possible scenarios. For each scenario inside each thread frame a Pareto curve can be extracted by the design-time scheduler. At run-time the correct Pareto curve can be chosen depending on the actual or estimated values of the data dependent values.

Describing detailed methods for clustering data dependent values in scenarios fall outside the scope of this thesis. The current approach is rather ad hoc, based on profiling and statistics. The idea is to profile an application and make a histogram of the values that the parameters can take. A percentile ranking is then used on each parameter to define ranges. All combinations of ranges of different parameters are then considered and Pareto curves are extracted by the design-time scheduler. Pareto curves from different combinations that look similar are again lumped together to further reduce the number of scenarios. This principle of clustering data dependencies in scenarios has been demonstrated in the MPEG21 DAC QoS demo [222].

Intra thread frame scenarios are used for handling data dependencies inside thread frames in type 1 applications.

C.1.3.2 Intra thread frame subscenarios

In type 2 applications the data dependent values can vary a lot and too many scenarios result due to combinatorial explosion.

Initially it was proposed to split thread frames in such cases in two separate thread frames, allowing the run-time scheduler to re-evaluate the working point based on actual values or more up-to-date estimated values of the performance parameters. The drawback of this solution is that in type 2 applications it leads to many fine-grained thread frames (each of which potentially defines scenarios). This in turn results in many Pareto curves to be stored, and considerable run-time scheduling (point selection) overhead.

The currently proposed solution is to let the run-time scheduler adapt a thread frame working point while it is running. The re-evaluation of the working point refines a scenario by deciding about a subscenario. A subscenario hence is a subset of a scenario. A subscenario's parameter values (needed to decide best working point) can be known accurately enough only when the thread frame is

already running (and hence was already assigned a worse-case working point).

Subscenarios are meant for use in type 2 applications. They complicate the run-time scheduling because shifting working points of existing thread frames must be possible. For this reason they are currently not be used in type 1 applications.

C.1.3.3 Thread cluster scenarios

This subsection is relevant for type 2 applications only. If two or more thread frames are *always* activated simultaneously the gray box model has a possibility for putting these thread frames in a thread cluster. The Pareto curves for the thread frames in the thread clusters are then combined to a global Pareto curve (see section B.4.2 on page 175 for some discussion about issues with respect to combining Pareto curves of different thread frames into one more global Pareto curve).

If two or more thread frames are *often* activated simultaneously thread cluster scenarios can be defined. As an example: if it is detected that two thread frames (combined to a thread cluster at design time) are simultaneously activated, the combined Pareto curve of the thread cluster (calculated at design time as well) can be used to reduce the run-time overhead in working point selection. If one of the two thread frames is separately activated its thread frame Pareto curve can be used instead.

C.2 Future work

This section describes a number of possible future work extensions to TCM, which are not immediately related to the rest of this thesis.

C.2.1 Future TCM work

1. two-phase scheduling approach with higher-dimensional Pareto surfaces:
    - how to quickly and accurately select a Pareto working point at run-time
    - gradual improvement of working point selection by shifting working points as thread frames run
    - different run-time scheduler algorithms (evaluating trade offs between effort done by run-time scheduler and quality of solution)
    - integration with quality of service

Quality can be modeled as the sum of a fixed hard minimal quality constraint and inverse of a variable quality cost:

$$q_{\text{tot}} = q_{\text{fixed}} + \frac{1}{c_{\text{variable}}}$$

where

$$c_{\text{variable}} \sim \frac{1}{\text{variable quality level}}$$

The Pareto curves can then be extended to contain a constraint axis representing q_{fixed} and a cost axis representing $c_{variable}$. After assigning a value for the constraints (execution time and minimal quality) the variable quality cost still can be minimized together with the other costs under consideration, as discussed in appendix B. A minimal variable quality cost then corresponds to a extra maximal variable quality.

- how to handle functionality (plug-ins) which is downloaded from a network? embed Pareto meta-information?

2. run-time scheduling: better exploit concurrency of tasks:

   - run-time scheduling with overlapping thread frames. What are the gains compared to non-overlapping thread frames? What can be done about overlapping thread frames needing common platform resources?
   - run-time scheduling with interleaving thread frames. Where to interleave? When to interleave? Can pre-emption and the resulting context switching overhead be avoided by defining suitable interleaving points? What are the gains in comparison to just overlapping thread frames due to better platform utilization?
   - interaction of e.g. timing constraints on possibilities for overlapping and interleaving?
   - shifting Pareto points of thread frames while they are running
     - in theory voltage can be scaled continuously, and speed is linearly dependent on voltage so one can generate a working point that lies on a theoretically continuous Pareto curve
     - in practice the voltage of a processor can be one of a few discrete values leading to discrete Pareto points
     - to better approximate an ideal working point it can be decided to run a thread frame at higher speed for a while, then switch to a lower speed, and possibly switch to a third or more speeds later during running

– however some care must be taken as switching the voltage too often results in overhead. Indeed, from statistics it is known that

$$\frac{\sum(x_i^2) - N * (\text{avg}(x_i)^2)}{n-1} = \text{stddev}^2 >= 0, \forall x_i$$

. Now substitute for x_i the different voltages to which the thread frame switches in order to approximate the ideal working point. Then x_i is approximately a measure for the execution time, and x_i^2 is a measure of the energy consumption. Interpreting the formula gives:

$$\sum(\text{energy at each point}) = \text{total energy consumption for TF}$$
$$>= N * (\text{energy consumed at ideal working point})$$

The energy at the ideal voltage can never be equaled by discrete voltage step approximation. The more times the voltage is changed (e.g. to come to a better approximation of the ideal working point), the worse this overhead becomes.

C.2.2 Interaction with other parts of the unified meta flow

1. task-level DTSE (T-DTSE) generates a number of constraints that must be obeyed by the subsequent TCM.
   - is enough freedom left for TCM after T-DTSE?
   - can T-DTSE be done completely before TCM or do interactions exist which require mixing T-DTSE and TCM steps?

Appendix D

Recursion removal source code

D.1 BaseBCounter implementation

```
class BaseBCounter:
 "A Base B Counter class"
 def __init__(self,B,D):
  "Create Base B number with D digits,init to 0"
  self.base=B
  self.digits=[0]*D
 def __repr__(self):
  "Return string representation of Base B digit"
  rep=self.digits[:]
  rep.reverse()
  return rep.__repr__()+" Base: "+self.base.__repr__()
 def __getitem__(self,I):
  "Return digit I from Base B digit"
  return self.digits[I]
 def __setitem__(self,I,value):
  "Set digit I to Base B digit"
  if value < self.base and value >= 0:
   self.digits[I]=value
  else:
   print "Warning: error in digit assignment."
   print "Setting to zero instead."
   self.digits[I]=0
 def inc(self):
  "Increase base B number with 1"
  carry,b,depth=0,self.base,len(self.digits)
  digit=self.digits[0]+1
  if digit >= b :
   carry,digit=1,digit-b
  self.digits[0]=digit
```

```
30   if carry:
      for i in range (1,depth):
32     digit=self.digits[i] + 1
       if digit >= b :
34       digit=digit-b
         self.digits[i]=digit
36     else :
         self.digits[i]=digit
38       break
    def incpow(self,pow):
40   "Increase␣base␣B␣number␣with␣B^pow"
     carry,b,depth=0,self.base,len(self.digits)
42   digit=self.digits[pow]+1;
     if digit >= b:
44    carry,digit=1,digit-b
     self.digits[pow]=digit
46   if carry:
      for i in range(pow+1,depth):
48     digit=self.digits[i]+1
       if digit >= b:
50       digit=digit-b
         self.digits[i]=digit
52     else:
         self.digits[i]=digit
54       break
```

D.2 General recursion removal example

```
1 # define base case conditions and base case calculations
  def bcrv0(x,y,fbc):
3  if fbc==FAKE:
     return 1
5  else:
     print "bcrv0"
7    return (x<=0)
  def bccrv0(x,y,fbc):
9  if fbc==FAKE:
     return DONTCARE
11 else:
     print "first␣base␣case:␣x=␣",x,",y=␣",y
13   return 2
  def bcrv1(x,y,fbc):
15 if fbc==FAKE:
     return 1
17 else:
     print "bcrv1"
19   return (y<=0)
  def bccrv1(x,y,fbc):
21 if fbc==FAKE:
     return DONTCARE
23 else:
     print "second␣base␣case:␣x␣",x,",y=␣",y
25   return 5
  bcrv=[bcrv0,bcrv1]
27 bccrv= [bccrv0,bccrv1]
```

```
  # define recursive case conditions
29 def rcrv0(x,y,fbc):
    print "rcrv0"
31  return x+y > 5
   def rcrv1(x,y,fbc):
33  print "rcrv1"
    return 1
35 rcrv=[rcrv0,rcrv1]
   # some functions with side effects
37 def verbose_decrement (arg,factor,amount):
    print "␣␣***␣Arg␣trafo␣arg=",arg,",factor=",factor,",amount=",amount
39  return arg / factor - amount
   def combine_sub (x,y):
41  print "␣␣Combining␣return␣values␣with␣subtraction!␣x=␣",x ,",y=␣",y
    return x - y
43  # define intermediate functions
   def grv00(x,y,(r1,r2,r3,result)):
45  print "g00␣entering␣first␣recursive␣case,x=␣",x,",y=␣",y
    return (r1,r2,r3,result)
47 def grv01(x,y,(r1,r2,r3,result)):
    print "g01␣-␣x=␣",x,"␣y=␣",y
49  return (r1,r2,r3,result)
   def grv02(x,y,(r1,r2,r3,result)):
51  print "g02␣-␣x=␣",x,"␣y=␣",y
    return (r1,r2,r3,result)
53 def grv03(x,y,(r1,r2,r3,result)):
    print "g03"
55  return (r1,r2,r3,combine_sub(r1,r2))
   def grv10(x,y,(r1,r2,r3,result)):
57  print "g10␣starting␣second␣recursive␣case,x=␣",x,",y=␣",y
    return (r1,r2,r3,result)
59 def grv11(x,y,(r1,r2,r3,result)):
    print "g11␣␣second␣part␣second␣recursive␣case,x=␣",x,",y=␣",y
61  return (r1,r2,r3,result)
   def grv12(x,y,(r1,r2,r3,result)):
63  print "g12␣␣third␣part␣second␣recursive␣case,x=␣",x,",y=␣",y
    return (r1,r2,r3,result)
65 def grv13(x,y,(r1,r2,r3,result)):
    print "g13␣␣now␣combining␣results␣in␣2nd␣rec␣case....,x=␣",x,",y=␣",y
67  return (r1,r2,r3,r1+r2+r3)
   grv=[[grv00,grv01,grv02,grv03],[grv10,grv11,grv12,grv13]]
69 #define argument transformation functions
   def arv000(x,y):
71  print "a000",x,y
    return verbose_decrement(x,1,1)
73 def arv001(x,y):
    print "a001",x,y
75  return verbose_decrement(y,2,1)
   def arv002(x,y):
77  print "a002",x,y
    return NORM
79 def arv010(x,y):
    print "a010",x,y
81  return verbose_decrement(x,1,2)
   def arv011(x,y):
83  print "a011",x,y
```

```
 return verbose_decrement(y,1,3)
def arv012(x,y):
 print "a012",x,y
 return NORM
def arv020(x,y):
 print "a020",x,y
 return DONTCARE
def arv021(x,y):
 print "a021",x,y
 return DONTCARE
def arv022(x,y):
 print "a022",x,y
 return FAKE
def arv100(x,y):
 print "a100",x,y
 return verbose_decrement(x,1,3)
def arv101(x,y):
 print "a101",x,y
 return verbose_decrement(y,1,2)
def arv102(x,y):
 print "a102",x,y
 return NORM
def arv110(x,y):
 print "a110",x,y
 return verbose_decrement(x,1,4)
def arv111(x,y):
 print "a111",x,y
 return verbose_decrement(y,3,4)
def arv112(x,y):
 print "a112",x,y
 return NORM
def arv120(x,y):
 print "a120",x,y
 return verbose_decrement(x,1,1)
def arv121(x,y):
 print "a121",x,y
 return verbose_decrement(y,1,1)
def arv122(x,y):
 print "a122",x,y
 return NORM
arv=[[[arv000,arv001,arv002],
       [arv010,arv011,arv012],
       [arv020,arv021,arv022]],
      [[arv100,arv101,arv102],
       [arv110,arv111,arv112],
       [arv120,arv121,arv122]]]
# a recursive algorithm showing off all possible difficulties
def allrec(x,y,fbc):
 if fbc==FAKE:
  return DONTCARE
 elif bcrv[0](x,y,fbc):
  return bccrv[0](x,y,fbc)
 elif bcrv[1](x,y,fbc):
  return bccrv[1](x,y,fbc)
 elif rcrv[0](x,y,fbc):
  r1,r2,r3,result=DONTCARE,DONTCARE,DONTCARE,DONTCARE
```

```
    (r1,r2,r3,result)=grv[0][0](x,y,(r1,r2,r3,result))
141 arg1,arg2,arg3=arv[0][0][0](x,y),arv[0][0][1](x,y),arv[0][0][2](x,y)
    r1=allrec(arg1,arg2,arg3)
143 (r1,r2,r3,result)=grv[0][1](x,y,(r1,r2,r3,result))
    arg1,arg2,arg3=arv[0][1][0](x,y),arv[0][1][1](x,y),arv[0][1][2](x,y)
145 r2=allrec(arg1,arg2,arg3)
    (r1,r2,r3,result)=grv[0][2](x,y,(r1,r2,r3,result))
147 arg1,arg2,arg3=arv[0][2][0](x,y),arv[0][2][1](x,y),arv[0][2][2](x,y)
    r3=allrec(arg1,arg2,arg3)
149 (r1,r2,r3,result)=grv[0][3](x,y,(r1,r2,r3,result))
    return result
151 elif rcrv[1](x,y,fbc):
    r1,r2,r3,result=DONTCARE,DONTCARE,DONTCARE,DONTCARE
153 (r1,r2,r3,result)=grv[1][0](x,y,(r1,r2,r3,result))
    arg1,arg2,arg3=arv[1][0][0](x,y),arv[1][0][1](x,y),arv[1][0][2](x,y)
155 r1=allrec(arg1,arg2,arg3)
    (r1,r2,r3,result)=grv[1][1](x,y,(r1,r2,r3,result))
157 arg1,arg2,arg3=arv[1][1][0](x,y),arv[1][1][1](x,y),arv[1][1][2](x,y)
    r2=allrec(arg1,arg2,arg3)
159 (r1,r2,r3,result)=grv[1][2](x,y,(r1,r2,r3,result))
    arg1,arg2,arg3=arv[1][2][0](x,y),arv[1][2][1](x,y),arv[1][2][2](x,y)
161 r3=allrec(arg1,arg2,arg3)
    (r1,r2,r3,result)=grv[1][3](x,y,(r1,r2,r3,result))
163 return result
# an iterative implementation without stack
165 # this shows how all difficulties can be handled:
#  non-linear recursion
167 #  multiple base cases with base case and base case calculation side effects
#  multiple recursive cases with side effects (this requires a memoization table)
169 #  different number of recursive calls in different recursive cases
#  -> first made equal using fbc=forced base case trick.
171 #  intermediate functions with side effects
#  return values with return value combination functions with side effects
173 #
# return values are kept in an array until needed (not a stack!)
175 def reccaserv(x,y,fbc):
 if (rcrv[0](x,y,fbc)):
177  return 0
 elif (rcrv[1](x,y,fbc)):
179  return 1
def memo_reccaserv(memo_table,key,(x,y,fbc)):
181 "Wrapper␣around␣reccaserv␣that␣memoizes␣the␣reccaserv␣results"
 if memo_table.has_key(key):
183  (k,valid)=memo_table[key]
  if valid:
185   return (memo_table[key],memo_table)
  else:
187   memo_table[key]=(reccaserv(x,y,fbc),1)
   return (memo_table[key],memo_table)
189 else:
  memo_table[key]=(reccaserv(x,y,fbc),1)
191  return (memo_table[key],memo_table)
def invalidate_reccase_memo_table(memo_table,key):
193 """When␣backtracking␣out␣of␣a␣certain␣lvl␣in␣the␣call␣tree,
␣␣␣␣the␣determined␣recursive␣case␣becomes␣invalid"""
195 (a,valid)=memo_table[key]
```

```
    memo_table[key]=(a,0)
197 def memo_F(i,j,memo_table,key,(x,y,fbc)):
    "Wrapper␣around␣F[i][j]␣that␣memoizes␣the␣F[i][j]␣results"
199 if memo_table.has_key(key):
     (a,b,c,valid)=memo_table[key]
201  if valid:
      return (memo_table[key],memo_table)
203  else:
      memo_table[key]=(arv[i][j][0](x,y),arv[i][j][1](x,y),arv[i][j][2](x,y),1)
205   return (memo_table[key],memo_table)
    else:
207  memo_table[key]=(arv[i][j][0](x,y),arv[i][j][1](x,y),arv[i][j][2](x,y),1)
    return (memo_table[key],memo_table)
209 def invalidate_F_memo_table(memo_table,itidx,olditidx):
    """For␣every␣digit␣that␣changes␣after␣updating␣itidx,
211 ␣␣␣␣F[i][j]␣result␣becomes␣invalid␣(new␣part␣of␣next␣call␣tree␣path)"""
    for key in range(len(itidx.digits)):
213  if itidx.digits[key] <> olditidx.digits[key]:
      if memo_table.has_key(key):
215    (x,y,z,valid)=memo_table[key]
      memo_table[key]=(x,y,z,0)
217   else:
      break
219 return memo_table
   def allit(x0,x1,fbc):
221 "Iterative␣version␣of␣the␣recursive␣algorithm"
    #initialize some variables
223 nrbc,base,maxdepth,Asubscript,prevAsub,memo_table,crc_table =\
    2,3,max(x0,x1)+1,0,0,{},{}
225 rv=[[0]*base for i in range(maxdepth)]
    itidx=BaseBCounter(base,maxdepth+1)
227 #check for immediate base cases
    if fbc==FAKE:
229  return
    for i in range(0,nrbc):
231  if bcrv[i](x0,x1,fbc):
      return bccrv[i](x0,x1,fbc)
233 #if none found,start iterating
    while itidx[maxdepth]==0: # for each leaf
235  tx0,tx1,tfbc,gcnt,btcnt=x0,x1,fbc,0,0
     # find out which gij to activate
237  while (gcnt < len(itidx.digits)) and (itidx[gcnt] == 0):
      gcnt=gcnt+1
239  # follow path from root to leaf
     for d in range(maxdepth-1,-1,-1):
241   bc,prevAsub,Asubscript,lvl=0,Asubscript,itidx[d],maxdepth-d-1
      ((crc,valid),crc_table)=memo_reccaserv(crc_table,lvl,(tx0,tx1,tfbc))
243   #find out the j in gij
      if gcnt==d:
245    gnumber=Asubscript
      elif gcnt > d:
247    gnumber=0
      #initialize return value array reply level
249   if (lvl<maxdepth) and (lvl<>0):
       reslev,resbranch=lvl-1,prevAsub
      else:
251
```

```
    reslev,resbranch=0,0
253  #if some gij must be activated,now is THE time
     if gcnt >= d:
255   (rv[lvl][0],rv[lvl][1],rv[lvl][2],rv[reslev][resbranch]) =\
      grv[crc][gnumber](tx0,tx1,(rv[lvl][0],rv[lvl][1],rv[lvl][2],rv[reslev][resbranch]))
257  # apply argument transformation functions
     ((tx0,tx1,tfbc,valid),memo_table) =\
259    memo_F(crc,Asubscript,memo_table,d,(tx0,tx1,tfbc))
     # check for early base cases
261  if gcnt >= d:
      for e in range(0,nrbc):
263    if bcrv[e](tx0,tx1,tfbc):
        rv[lvl][Asubscript]=bccrv[e](tx0,tx1,tfbc)
265     bc=1
        break
267  if bc:
      break
269 # if early base case found,handle backtracking if needed
    if bc:
271  btcnt,crc2=gcnt,0
     while btcnt < maxdepth-1 and itidx[btcnt-gcnt+d]==itidx.base-1:
273   # start backtracking; instead of recalculating the argument values as we did
      # before, we reuse them from the memoization table
275   lvl2=maxdepth - btcnt - 1
      (hx0,hx1,helpfbc,helpvalid)=memo_table[btcnt+1]
277   ((crc2,valid),crc_table)=memo_reccaserv(crc_table,lvl2,(hx0,hx1,helpfbc))
      invalidate_reccase_memo_table(crc_table,lvl2)
279   prevAsub=itidx[btcnt+1]
      (rv[lvl2][0],rv[lvl2][1],rv[lvl2][2],rv[lvl2-1][prevAsub]) =\
281   grv[crc2][base](hx0,hx1,(rv[lvl2][0],rv[lvl2][1],rv[lvl2][2],rv[lvl2-1][prevAsub]))
      btcnt=btcnt+1
283  if (btcnt==maxdepth-1) and (itidx[btcnt-gcnt+d]==(base-1)):
      # here we arrive only if we are executing the very last step
285   ((crc,valid),crc_table)=memo_reccaserv(crc_table,0,(x0,x1,fbc))
      (rv[0][0],rv[0][1],rv[0][2],rv[0][0]) =\
287   grv[crc][base](x0,x1,(rv[0][0],rv[0][1],rv[0][2],rv[0][0]))
    # invalidate the argument trafo function memoization table
289 # note: much more efficiently when incorporated in the itidx.incpow method,
    # not done here for reasons of clarity
291 olditidx=copy.deepcopy(itidx)
    itidx.incpow(d) # skip virtual leaves
293 memo_table=invalidate_F_memo_table(memo_table,itidx,olditidx)
   return rv[0][0]
```

Bibliography

[1] S. Adve, D. Burger, R. Eigenmann, A. Rawsthorne, M. D. Smith, C. Gebotys, M. Kandemir, D. J. Lilja, A. Choudhary, J. Fang, and P.-C. Yew. The interaction of architecture and compilation technology for high-performance processor design. *IEEE Computer Magazine*, 30(12):51–58, December 1997.

[2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222, October 1999.

[3] J. Aldrich, E. G. Sirer, C. Chambers, and S. Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, May-June 2003. Also appeared as "Static Analyses for Eliminating Unnecessary Synchronization from JAVA programs", in Proceedings of the 6th International Symposium on Static Analysis, pages 19–38, 1999, Springer-Verlag.

[4] P. Amagbégnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 163–173, La Jolla, California, United States, 1995. ACM Press.

[5] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[6] T. Apostel. Composite functions and continuity. In *Calculus, 2nd ed., Vol. 1: One-Variable Calculus, with an Introduction to Linear Algebra*, pages 140–141. John Wiley & Sons, June 1967.

[7] ARM. http://www.arm.com.

[8] J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, 1996.

[9] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. ISSN: 0001-0782.

[10] J. Backus. From function level semantics to program transformation and optimization. In *Proceedings of the international joint conference on theory and practice of software development (TAPSOFT)*, volume 1: Colloquium on trees in algebra and programming (CAAP'85), pages 60–91, Berlin, Germany, March 1985.

[11] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 33(6) of *ACM SIGPLAN Notices*, pages 258–268, Montreal, Canada, June 1998.

[12] H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3), March 1992.

[13] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, June 1997.

[14] E. Barros and A. Sampaio. Towards provably correct hardware/software partitioning using OCCAM. In *Proceedings of the Third International Workshop on Hardware/-Software Codesign*, pages 210–217, 1994.

[15] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Texts and Monographs in Computer Science. Springer Verlag, ISBN 0-387-11148-4, 1982.

[16] R. Ben-Natan. *CORBA: A guide to Common Object Request Broker Architecture*. McGraw-Hill, 1995.

[17] J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

[18] A. Benveniste and G. Berry. *Readings in hardware/software co-design*, chapter The synchronous approach to reactive and real-time systems, pages 147–159. Kluwer Academic Publishers, 2002.

[19] J. Bergstra and J. Klop. The algebra of recursively defined processes and the algebra of regular processes. In A. Ponse, C. Verhoef, and S. van Vlijmen, editors, *Algebra of Communicating Processes*, pages 1–25. Springer-Verlag, 1995.

[20] G. Berry. The foundations of esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[21] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2001.

[22] S. Bhattacharya, S. Dey, and F. Brglez. Performance analysis and optimization of schedules for conditional and loop-intensive specifications. In *Design automation conference*, 1994.

[23] S. S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java. Technical report, EECS, University of California, Berkeley, CA USA 94720, July 2003. Memorandum UCB/ERL M03/27.

[24] D. Binkley and K. B. Gallagher. Program slicing. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 43, pages 1–50. Academic Press, 1996.

[25] M. Boekhold, I. Karkowski, and H. Corporaal. Transforming and parallelizing ANSI c programs using pattern recognition. In *Proceedings of the High Performance Computing and Networking (HPCN) Europe'99*, Amsterdam, Nederland, April 1999.

[26] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, Colorado, November 1999.

[27] B. Bose and S. D. Johnson. *Derivation of a verified microprocessor*. PhD thesis, University of Indiana, 1994. a summary appears in: Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. An exercise in integrating verification with formal derivation. In G. Milne and L. Pierre, editors, Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods, pages 191–202. Springer, LNCS 683, 1993.

[28] W. Brauer, W. Reisig, and G. Rozenberg, editors. *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[29] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in el greco. In *Proceedings of the 8th International Workshop on Hardware/-Software Codesign (CODES'00)*, San Diego, California, May 2000.

[30] J. T. Buck and E. A. Lee. The token flow model. In L. Bic, G. Gao, and J.-L. Gaudiot, editors, *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Computer Society Press, 1993.

[31] H. J. Burkhardt, P. Ochsenschläger, and R. Prinoth. Product nets - a formal description technique for cooperating systems. *GMD - Studien*, 165, September 1989.

[32] F. P. Burns, D. Kinniment, and A. Koelmans. STRIDE: a tool for formal interactive systems synthesis. *IEE Proceedings on Computers and Digital Techniques*, 141(6):347–355, 1994.

[33] F. P. Burns, D. J. Kinniment, and A. M. Koelmans. Correct interactive transformational synthesis of DSP hardware. In *proceedings of the European Design Automation Conference*, pages 16–21, Amsterdam, 1991. IEEE Computer Society Press.

[34] R. M. Burstall and J. Darlington. A transformation system for developping recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[35] L. Cardelli. Type systems. In A. B. Tucker Jr., editor, *The Computer Science and Engineering Handbook*. CRC Press, Boca Raton, FL, 1997.

[36] C. Cassandras. *Discrete Event Systems, modeling and performance analysis*. Irwin: Homewood, Illinois, 1993.

[37] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, Boston, 2002.

[38] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1st edition, 1998.

[39] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. Optimizing power using transformations. *IEEE Transactions on Computer Aided Design*, pages 12–31, January 1995.

[40] A. P. Chandrakasan, M. Potkonjak, J. Rabaey, and R. W. Brodersen. HYPER-LP: A system for power minimization using architectural transformations. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 300–303, Santa Clara, California, United States, November 1992. IEEE Computer Society Press, Los Alamitos, CA, USA.

[41] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. In *Design Automation Conference*, 1993.

[42] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for DOACROSS loops. *IEEE Transactions on Parallel and Distributed Systems*, 10(5):459–470, May 1999.

[43] T. K.-Y. Cheung and G. Hellestrand. Multi-level equivalence in design transformation. In *Proceedings of International Conference on Computer Hardware Description Languages*, pages 559–566, Chiba Japan, September 1996.

[44] T. K.-Y. Cheung, G. Hellestrand, and P. Kanthamanon. A transformational codesign methodology. In *proceedings of the ASP-DAC '97 Asia and South Pacific*, pages 299–305, 1997.

[45] T. K.-Y. Cheung, G. Hellestrand, and P. Kanthamanon. A multi-level transformation approach to HW/SW codesign: A case study. In *Proceedings of 4th International Workshop on Hardware/Software Codesign, Codes/CASHE '96*, pages 10–17. IEEE Computer Society Press, 1998.

[46] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

[47] T. A. C. M. Claasen. High speed: Not the only way to exploit the intrinsic computational power of silicon. In *Proceedings of the International Solid-State Circuits Conference*, pages 22–25, San Fransisco, CA, February 1999.

[48] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):265–299, July 1983.

[49] J.-F. Collard. *Reasoning about Program Transformations: imperative programming and flow of data*. Springer Verlag ISBN 0-387-95391-4, 2003.

[50] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press and McGraw Hill, 2001.

[51] L. A. Cortés, P. Eles, and Z. Peng. A survey on hardware/software codesign representation models. Technical report, SAVE Project Report, Dept. of Computer and Information Science, Linköping University, Linköping, June 1999.

[52] N. Cossement, R. Lauwereins, and F. Catthoor. DF*: An extension of synchronous data flow with data dependency and non-determinism. In *Proceedings of Forum on Design Languages 2000, FDL00/SSDL Workshop*, Tübingen, Germany, 2000.

[53] J. Darlington and R. M. A. Burstall. A system which automatically improves programs. In *proceedings of the third international joint conference on artifficial intelligence*, pages 479–485, Stantord, 1973.

[54] E. G. Daylight, D. Atienza, A. Vandecappelle, F. Catthoor, and J. Mendias. Memory-access-aware data structure transformations for embedded software with dynamic data accesses. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):269–280, March 2004.

[55] E. A. de Kock, G. Essink, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, and K. A. Vissers. YAPI: Application modeling for signal processing systems. In *Proceedings of the 37th Design Automation Conference (DAC00)*, pages 402–405, Los Angeles, California, June 2000.

[56] H. De Man. System design challenges in the post-PC era. In *Proceedings of the 37th Design Automation Conference*, Los Angeles, 2000.

[57] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Science/Engineering/Math, January 1994.

[58] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The olympus synthesis system. *IEEE Design & Test archive*, 7(5):37–53, September 1990.

[59] M. V. der Beeck. A comparison of statechart variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148. Berlin Springer, 1994.

[60] E. W. Dijkstra. Cooperating sequential processes. Technical report, University of Eindhoven, The Netherlands, 1965. Technical Report EWD-123.

[61] P. C. Diniz and M. C. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, March 1998.

[62] C. J. Fidge. A comparative introduction to CSP, CCS and LOTOS, technical report 93-24. Technical report, Software Verification Research Centre, University of Queensland, January 1994.

[63] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for java. *Software - Practice and Experience*, 30(3):199–232, March 2000.

[64] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, June 1999.

[65] D. P. Friedman. Applications of continuations: Invited tutorial, January 13–15 1988. Fifteenth Annual ACM Symposium on Principles of Programming Languages.

[66] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 2 edition, 2001.

[67] D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE Design and Test of Computers*, 11(4):44–54, 1994.

[68] D. D. Gajski, F. Vahid, S. Narajan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of reusable object-oriented software*. Addison-Wesley Professional Computing Series, 1995.

[70] D. Genin, E. Hilfinger, J. Rabaey, C. Scheers, and H. De Man. DSP specification using the SILAGE language. In *proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1057–1060, 1990.

[71] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, February 1991.

[72] E. Girczyc. Loop winding - a data flow approach to functional pipelining. In *International Symposium of Circuits and Systems*, 1987.

[73] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent shared-memory multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 64–75, April 1989.

[74] G. Goossens, J. Vandewalle, and H. De Man. Loop optimization in register-transfer scheduling for DSP-systems. In *Design automation conference*, 1989.

[75] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, Massachusetts, 1994.

[76] W. Grass, C. Grobe, S. Lenk, W. D. Tiedemann, C. D. Kloos, A. Marin, and T. Robles. Transformation of timing diagram specifications into VHDL code. In *Proceedings of the 12th IFIP International Conference on Computer Hardware Description Languages and their Applications CHDL'95*, pages 659–668, Chiba, Japan, 1995.

[77] D. P. Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.

[78] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design*, 2003.

[79] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Design Automation Conference*, 2002.

[80] S. Gupta, M. Miranda, F. Catthoor, and R. K. Gupta. Analysis of high-level address code transformations for programmable processors. In *Design, Automation and Test in Europe*, 2000.

[81] G.Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Hardware/-software partitioning for embedded systems in OCAPI-xl. In *Proceedings of the 9th international symposium on hardware/software codesign (CODES'01)*, Copenhagen, Denmark, April 2001.

[82] A. Hafid, G. v. Bochmann, and R. Dssouli. Distributed multimedia applications and quality of service: a review. *Electronic Journal on Network and Distributed Processing*, 6:1–50, February 1998.

[83] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[84] H. Han, C.-W. Tseng, and P. Keleher. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International Journal of Parallel Programming*, 26(5):591–612, October 1998.

[85] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[86] P. G. Harrison and H. Koshenvisan. A new approach to recursion removal. *Theoretical Computer Science*, 93(1):91–113, 1992.

[87] P. G. Harrison and H. Koshnevisan. Linearisation: An optimization for nonlinear functional programs. *Science of Computer Programming*, 10:218–318, 1998.

[88] W. L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, October 1989.

[89] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, Massachusetts, 1991.

[90] S. Himpe. Modeling systems using MTG* and gray box and classification of TCM transformations, July 2001. project deliverable, Natlab-K.U.Leuven overeenkomst RWR-061-JR-01001-jr, project: Dynamic scheduling for embedded multiprocessor architectures.

[91] S. Himpe. Modeling systems using MTG* and gray box model, classification of TCM transformations, July 2001. project deliverable, IWT-ITA-II GBO330 Nr. 980302, SuperVisie: Beeldverwerking voor geïntegreerde en immersieve visualisatie (Final Deliverable).

[92] S. Himpe. MTG* and gray box, 2001. Extended abstract and poster PA3CT symposium 2001.

[93] S. Himpe. Gray box model extraction and TCM transformations on JPEG and MPEG21 demonstrators, July 2002. project deliverable, Natlab-K.U.Leuven overeenkomst RWR-061-JR-01001-jr, project: Dynamic scheduling for embedded multiprocessor architectures.

[94] S. Himpe. MTG* and gray box, modeling of dynamic multimedia applications, 2002. Extended abstract and poster at KUL PhD symposium 2002.

[95] S. Himpe. XML format to store MTG* model. Technical report, Katholieke Universiteit Leuven, November 2002.

[96] S. Himpe. Control flow analysis for recursion removal, 2003. Extended abstract and poster at PA3CT symposium 2003.

[97] S. Himpe. Semaphore transformations for task concurrency management. In *proceedings of PA3CT 2004*, Edegem, September 2004. Extended abstract and poster.

[98] S. Himpe, F. Catthoor, and G. Deconinck. Removing recursion from imperative programs. *Software - Practice and Experience (SPE)*, 2005. Submitted for review.

[99] S. Himpe, F. Catthoor, G. Deconinck, and J. van Meerbergen. MTG* and gray box: modeling dynamic multimedia applications with concurrency and non-determinism. In *proceedings of the forum on specification and design languages (FDL02) on CDROM*, Marseille, France, September 2002.

[100] S. Himpe, F. Catthoor, G. Deconinck, and J. van Meerbergen. *System Specification and Design Languages: Best of FDL02*, chapter MTG* and gray box: modeling dynamic multimedia applications with concurrency and non-determinism. Kluwer Academic Publishers, 2003.

[101] S. Himpe, G. Deconinck, and F. Catthoor. TCM transformation classification. Technical report, Katholieke Universiteit Leuven, 2001.

[102] S. Himpe, G. Deconinck, and F. Catthoor. Control flow analysis for recursion removal. In A. Krall, editor, *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2003)*, volume Lecture Notes in Computer Science 2826, pages 101–116, Vienna, Austria, September 2003. Springer Verlag.

[103] S. Himpe, G. Deconinck, and F. Catthoor. Control flow analysis for recursion removal. Technical report, Katholieke Universiteit Leuven, 2004.

[104] S. Himpe, G. Deconinck, F. Catthoor, and J. van Meerbergen. MTG* and gray box modeling: a modern bible for TCM methodologists. Technical report, Katholieke Universiteit Leuven, 2002.

[105] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[106] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[107] C. A. R. Hoare, editor. *Occam 2 Reference Manual*. International Series in Computer Science. Prentice Hall, 1988. Inmos Limited.

[108] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *European Design and Test Conference*, 1995.

[109] IEEE. *IEEE Standard for Verilog Hardware Description Language*. IEEE, 2001. IEEE Std 1364-2001.

[110] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, 2002. IEEE Std 1076-2002.

[111] ISO/IEC JTC1/SC29/WG11/N4668. Overview of the MPEG-4 standard, March 2002. Ed.: Rob Koenen.

[112] R. Jagannathan. Dataflow models. In E. Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*. McGraw-Hill, 1995.

[113] J. W. Janneck and R. Esser. Higher-order petri net modeling—techniques and applications. In *Proceedings of the Workshop on Software Engineering and Formal Methods, Petri Nets*, Adelaide, Australia, 2002.

[114] K. Jensen. Coloured petri nets: A high level language for system design and analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag, Berlin, Germany, 1991.

[115] N. K. Jha. Low power system scheduling and synthesis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 259–263, San Jose, CA, November 2001.

[116] W. Jiacun. *Timed Petri Nets*. Kluwer Academic Publishers, Boston, 1998.

[117] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, December 1997.

[118] B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java(TM) Language Specification (2nd Edition)*. Addison-Wesley Pub Co, June 2000.

[119] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[120] O. Kaser, S. Pawagi, and C. R. Ramakrishnan. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1–4):151–164, 1993.

[121] R. Kelsey and P. Hudak. Realistic compilation by program transformation. In *proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 281–292, Austin, Texas, January 1989.

[122] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, August 2004.

[123] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

[124] A. M. Koelmans, F. P. Burns, and D. J. Kinniment. Use of a theorem prover for transformational systhesis. In *proceedings of the Colloqium on Formal and Semi-formal Methods for Digital Systems Design*, London, 1991. IEE Press.

[125] A. Kountouris and C. Wolinski. High level pre-synthesis optimization steps using hierarchical conditional dependency graphs. In *Euromicro Conference*, 1999.

[126] A. Krall and M. Probst. Monitors and exceptions: How to implement java efficiently. In S. Hassanzadeh and K. Schauser, editors, *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 15–24, Palo Alto, March 1998.

[127] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message-passing and shared-memory: early experience. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 54–63, San Diego, California, United States, 1993. ACM Press.

[128] J. D. Kubiatowicz. *Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, 1998. MIT/LCS/TR-744.

[129] KULFTP. Ftp site, 2005. ftp://ftp.esat.kuleuven.be/pub/acca/recursion_to_iteration.

[130] D. Lanneer. *Design Models and Data Path Mapping for Digital Signal Processing Architectures*. PhD thesis, Katholieke Universiteit Leuven, March 1993.

[131] M. Larsson. A transformational approach to formal digital system design. Thesis no 378, May 1993.

[132] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete. Geometric parallelism and cyclo-static data flow in GRAPE-II. In *Proceedings of the IEEE Workshop on Rapid System Prototyping*, pages 90–107, Grenoble, France, June 1994.

[133] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, November 1999.

[134] B. Lee and E. A. Lee. Interaction of finite state machines and concurrency models. In *Proceedings of the 32nd Annual Asilomar Conference on Systems, Signals and Computers*, Pacific Grove, California, November 1998.

[135] E. Lee and D. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9), September 1987.

[136] E. A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, London, 2002.

[137] J. Li and R. K. Gupta. HDL optimizations using Timed Decision Tables. In *proceedings of the Design Automation Conference*, 1996.

[138] J. Li and R. K. Gupta. Decomposition of Timed Decision Tables and its use in presynthesis optimizations. In *International Conference on Computer Aided Design*, 1997.

[139] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley & Sons, 1998.

[140] Y. A. Liu and S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, volume 34 of *SIGPLAN Notices*, pages 73–82, Boston, Massachusetts, USA, November 1999. ACM Press, ISBN 1-58113-201-8.

[141] Z. Ma, C. Wong, S. Himpe, E. Delfosse, F. Catthoor, and G. Deconinck. Task concurrency analysis and exploration of visual texture decoder on a heterogeneous platform. In *Proceedings of the 2003 IEEE WORKSHOP ON SiGNAL PROCESSING SYSTEMS (SiPS03)*, Seoul, South Korea, August 2003.

[142] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[143] P. Marchal, C. Wong, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, and H. De Man. Dynamic memory oriented transformations in the MPEG4 IM1-player on a low power platform. In *Proceedings of the International Workshop on Power Aware Computing Systems (PACS)*, Cambridge MA, November 2000.

[144] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, April 1960.

[145] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1962.

[146] http://www.memorymanagement.org.

[147] P. F. A. Middelhoek. Transformational design of digital circuits. In *Proceedings of the Seventh Computersystems Workshop*, pages 57–69, Eindhoven, November 1993.

[148] P. F. A. Middelhoek. *Transformational Design: an Architecture Independent Interactive Design Methodology for the Synthesis of Correct and Efficient Digital Systems*. PhD thesis, University of Twente, April 1997.

[149] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, 1987.

[150] R. Milner. *Communicating and Mobile Systems - The Pi Calculus*. Cambridge University Press, June 1999.

[151] Y. Minamide. A functional representation of data structures with a hole. In *Conference Record of POPL '98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, January 1998. ACM Press.

[152] M. Miranda, F. Catthoor, M. Janssens, and H. De Man. High-level address optimization and synthesis techniques for data-transfer intensive applications. *IEEE transactions on VLSI systems*, 6(4):677–686, December 1998.

[153] S. Mostert. *A Computational Architecture for Real-Time sytems*. PhD thesis, University of Stellenbosch, Stellenbosch, South-Africa, 2001.

[154] S. Mostert, N. Cossement, R. Lauwereins, and J. L. van Meerbergen. DF*: Modeling dynamic process creation and events for interactive multimedia applications. In *IEEE International Workshop on Rapid System Prototyping 2001*, pages 122–127, 2001.

[155] Mozart Consortium. http://www.mozart-oz.org.

[156] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[157] Object Managment Group. http://www.omg.org/uml.

[158] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 223 – 237, Denver, Colorado, United States, 1999. ACM Press New York, NY, USA.

[159] P. Op de Beeck, F. Barat, M. Jayapala, and R. Lauwereins. CRISP: A template for reconfigurable instruction set processors. In *proceedings of FPL 2001*, pages 296–305, Belfast, North Ireland, August 2001.

[160] Open Systems Interconnection (OSI). LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 2000. technical committee/subcommittee JTC 1/SC 7, ISO 8897:1989.

[161] S. L. Pandey, K. Umamageswaran, and P. A. Wilsey. VHDL semantics and validating transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):936–955, July 1999.

[162] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999.

[163] H. A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Texts and monographs in Computer Science. Springer Verlag, ISBN 0-38752-356-1, 1990.

[164] O. Peñalba, J. Mendías, and R. Hermida. Maximizing conditional reuse by presynthesis transformations. In *Design, Automation and Test in Europe*, 2002.

[165] Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer level implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:150–166, 1994.

[166] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1981.

[167] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Universität Darmstadt, Fakultt Matematik und Physik, 1962.

[168] M. Philippsen and E. A. Heinz. Automatic synchronization elimination in synchronous FORALLs. In *Frontiers '95. The 5th Symposium on the Frontiers of Massively Parallel Computation*, Mc Lean, VA 1995. S. 350-357. Universität Karlsruhe; Institut für Programmstrukturen und Datenorganisation, 1995.

[169] Philips. http://www.semiconductors.philips.com/platforms/nexperia.

[170] J. Plevyak, X. Zhang, and A. A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 311–321, San Francisco, California, 1995.

[171] J. Plosila. *Self-Timed Circuit Design, The Action Systems Approach*. PhD thesis, University of Turku, Department of Physics, Finland, June 1999.

[172] I. Radivojevic and F. Brewer. Analysis of conditional resource sharing using a guard-based control representation. In *International Conference on Computer Design*, 1995.

[173] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[174] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[175] J. C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6:233–248, 1993.

[176] K. H. Rosen. *Discrete Mathematics and its Applications*. McGraw Hill Text, 1998.

[177] A. Sabry. What is a purely functional language? *The Journal of Functional Programming*, 8(1):1–22, January 1998.

[178] H. Samsom, L. Claesen, and H. De Man. Synguide: an environment for doing interactive correctness preserving transformations. In *proceedings of the IEEE Workshop on VLSI Signal Processing*, Veldhoven, The Netherlands, October 1993. Also in VLSI Signal Processing VI, L.Eggermont, P.Dewilde, E.Deprettere, Jef van Meerbergen (eds.), IEEE Press, New York, pages 269–277, 1993.

[179] I. Sander and A. Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings of IEEE Workshop on VLSI 99 (WVLSI 99)*, pages 32–39, Orlando, Florida, USA, April 8-9, 1999.

[180] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[181] S. Saracco, J. R. W. Smith, and R. Reed. *Telecommunications Systems Engineering Using SDL*. North-Holland - Elsevier, 1989.

[182] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[183] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[184] R. Sharp and O. Rasmussen. Transformational rewriting with ruby. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 231–248, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[185] S. K. Singhal, B. Q. Nguyen, R. Redpath, M. Fraenkel, and J. Nguyen. Building high-performance applications and services in JAVA: An experiential study, July 1997. used to be online at the IBM Developerworks web site.

[186] G. Smolka. A foundation for higher-order concurrent constraint programming. In J.-P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 50–72, München, Germany, September 1994. Springer-Verlag.

[187] J. Sobel and D. P. Friedman. Recycling continuations. In *proceedings of the International Conference on Functional Programming 1998 (ICFP '98)*, Baltimore, Maryland, 1998.

[188] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[189] E. Stoy. A petri net based unified representation for hardware/software systems, 1995. Licentiate Thesis LiU-Tek-Lic 1995:21.

[190] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. Funstate - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[191] B. Stroustup. *C++ Programming Language (Special 3rd Edition)*. Addison-Wesley Pub Co, 2000.

[192] http://www.systemc.org.

[193] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, pages 203–215, San Diego, CA, June 2003.

[194] A. S. Tanenbaum. *Modern Operating Systems, Second Edition*. Prentice Hall, 2001.

[195] Texas Instruments. http://www.ti.com.

[196] F. Thoen and F. Catthoor. *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*. Kluwer Academic Publishers, 2000.

[197] F. Tip. A survey of program slicing techniques. Technical report, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994. Report CS-R9438, Also appeared in Journal of Programming Languages, 3(3):121–181, September 1995.

[198] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, 1995.

[199] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *2*, pages 230–265, 1936.

[200] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. In *Proceedings of the London Mathematical Society*, volume 43 of *2*, pages 544–546, 1937.

[201] C. Valderrama, M. Romdhani, J. Daveau, G. Marchioro, A. Changuel, and A. Jerraya. Cosmos: A transformational co-design tool for multiprocessor architectures. In J. Staunstrup and W. Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[202] P. H. A. van der Putten, J. P. M. Voeten, M. C. W. Geilen, and M. P. J. Stevens. System level design methodology. In *Proceedings of IEEE Computer Society Workshop on System Level Design*, pages 11–16, 1998.

[203] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[204] T. Vander Aa. *Instruction Transfer and Storage Exploration for Low Energy Embedded VLIWs*. PhD thesis, September 2005.

[205] M. Varea and B. Al-Hashimi. Dual transitions petri net based modelling technique for embedded systems specification. In *Proceedings of the conference on Design, automation and test in Europe*, pages 566–571, Munich, Germany, 2001.

[206] A. H. Veen. Dataflow machine architecture. *ACM computing surveys*, 8(4):366–396, December 1986.

[207] D. Verkest, J. Cockx, and F. Potargent. On the use of c++ for system-on-chip design. In *Proceedings of the IEEE Workshop on VLSI (IWV)*, pages 42–47, Orlando, Florida, April 1999.

[208] J. P. M. Voeten, P. H. A. van der Putten, and M. P. J. Stevens. Behaviour-preserving transformations in SHE: a formal approach to architecture design. In *Beyond 2000: Hardware and Software Design Strategies, Proceedings of the 22nd EUROMICRO Conference*, pages 19–27, 1996.

[209] J. Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966.

[210] R. Walker and D. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Transactions on Computer Aided Design*, October 1989.

[211] X. Wang and E. P. Stabler. Formalization of VHDL synthesis procedure in higher-order logic. In *International Workshop on the HOL Theorem Proving System and Its Applications*, pages 106–119, 1991.

[212] M. P. Ward. A recursion removal theorem - proof and applications. Technical report, Durham University, 1991.

[213] M. P. Ward. A recursion removal theorem. In *Proceedings of the 5th refinement workshop*, pages 155–218. Springer-verlag, January 1992.

[214] D. Warren. DAI research report 141. Technical report, University of Edinburgh, 1980.

[215] P. Wauters, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-dynamic data flow. In *Proceedings of the 4th EuroMicro Workshop on Parallel and Distributed Processing*, pages 319–327, 1996.

[216] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[217] E. H. Weiss. Visualizing a procedure with nassi-shneiderman charts. *Journal of Technical Writing and Communication*, 20(3), 1990.

[218] C. Wong. *Design-time sub-task scheduling for embedded multimedia and telecom systems*. PhD thesis, Katholieke Universiteit Leuven, September 2003.

[219] W. Wu, I. Sander, and A. Jantsch. Tranformational system design based on a formal computational model and skeletons. In *proceedings of Forum on Design Languages (FDL 2000)*, Tübingen, Germany 4-8 September, 2000.

[220] Xilinx. http://www.xilinx.com.

[221] P. Yang. *Pareto-Optimization Based Run-Time Scheduling for Embedded Systems*. PhD thesis, Katholieke Universiteit Leuven, September 2004.

[222] P. Yang. *Pareto-optimization based run-time task scheduling for embedded systems*. PhD thesis, Katholieke Universiteit Leuven, August 2004.

[223] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. Managing dynamic concurrent tasks in embedded real-time multimedia systems. In *proceedings of the international symposium on system synthesis (ISSS02)*, pages 112–119, Kyoto, Japan, October 2002.

[224] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. *Multiprocessor Systems-on-Chips*, chapter Cost Efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. Morgan Kaufmann (Elsevier), 2004.

[225] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded multiprocessor SOCs. *IEEE Design&Test of Computers, special ITC issue*, 18(5):46–58, September-October 2001.

[226] C. Ykman-Couvreur, F. Catthoor, J. Vounckx, A. Folens, and F. Louagie. Energy-aware dynamic task scheduling applied to a real-time multimedia application on an xscale board. In *Submitted to SASIMI*, June 2004.

[227] C. Ykman-Couvreur, J. Lambrecht, D. Verkest, F. Catthoor, B. Svantesson, A. Hemani, and F. Wolf. Dynamic memory management methodology applied to embedded telecom network systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(5):650–667, October 2002.

[228] C. Ykman-Couvreur, P. Yang, P. David, S. Himpe, G. Keramidas, F. Catthoor, and J. Vounckx. Cost-efficient mapping of dynamic concurrent tasks in embedded real-time multimedia systems. Technical report, IMEC technical report, 2004.

[229] T. Z. Yu, E. H.-M. Sha, N. Passos, and R. D.-C. Ju. Algorithms and hardware support for branch anticipation. In *Great Lakes Symposium on VLSI*, 1997.

[230] S. Zamir. *Handbook of Object Technology*. CRC press, 1998.

[231] M. Zapf and A. Heinzl. Approaches to integrate petri nets and object-oriented concepts. *Wirtschaftsinformatik*, 42(1):36–46, 2000. (in German).

[232] M. Zhe, C. Wong, S. Himpe, E. Delfosse, F. Catthoor, and G. Deconinck. Task concurrency analysis and exploration of visual texture coder on a heterogeneous platform. In *proceedings of the 2003 IEEE workshop on signal processing systems (SIPS03)*, 2003.

[233] M. Zhe, P. Yang, S. Himpe, T. Van Achteren, P. David, C. Ykman-Couvreur, F. Catthoor, and J. Vounckx. Requirements for C annotations, tipsy usage, and XML specification of the gray box model and scenarios (version 0.2). Technical report, IMEC internal report, December 2003.

[234] M. Zhe, P. Yang, S. Himpe, T. Van Achteren, P. David, C. Ykman-Couvreur, F. Catthoor, and J. Vounckx. TCM demonstrators. Technical report, IMEC internal report, December 2003.

[235] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI: a system model for heterogeneously specified embedded systems. *IEEE Transactions of Very Large Scale Integration Systems*, 10(4):379–389, 2002.

Index

action systems, 67, 71
ADDL, 65
addressing, 66
annotation
    attribute, 37, 38, 53, 57
    attributes, 53
    dynamism, 19, 32
    events, 31
    timing, 32
application
    event-driven, 178
    frame-based, 177
    type 1, 177
    type 2, 178
ARGOS, 26

barrier synchronization, 69, 82
binding
    dynamic, 26
bisimulation, 24
Boolean logic, 18

C, 30
C++, 30
CAMAD, 65
cdfg, 9, 20, 31, 53, 54, 67
    transformation, 62, 63
    transformations, 171
CFSM, 26
Church-Turing thesis, 18
cluster
    stop cluster, 46, 51
    thread cluster, 53
Co-Centric System Studio, 30
Combinational logic, 18
combinatorial explosion, 180
compile-time, 20, 28, 29
concurrency
    algebraic approach, 24
    concurrent FSM, 25
    declarative, 23
    discrete event, 25
    message passing, 24
    Petri net, 26
    shared state, 24
    synchronous data flow, 20
    synchronous/reactive, 25
constraint, 74
continuation passing style, 71
control port, 39
CORBA, 30
COSMOS, 65
cost, 74

data dependent, 9, 27, 48, 54, 63, 156–158, 175, 177–179
data port, 39
DDD, 66
deadlock, 18, 25, 26
decorated signal flow graph, 20
delay, 57, 67, 157
demonstrator, 13, 139
design methodology, 5
design pattern, 30, 159
design space, 73
design space point, 73
design-time, 62, 63, 76
details
    hidden, 10, 80, 156
    interesting, 9
        semaphore removal, 81
    visible, 74
DF*, 9
DTSE, 155
    exploration, 156
    pre-processing, 156
    script, 155
    task-level DTSE, 82, 177, 183
dynamic behavior, 3–5, 9, 27, 29, 32, 179

edge, 43
    implicit timing constraint, 45
    inhibitor arc, 52
    restrictions, 44
    token marking, 45
embedded system
    aspects of, 17
    future, 5
energy consumption, 4, 6, 9, 10
    maximum, 9

minimum, 9
Erlang, 30
ESTEREL, 26, 30
evaluation
eager, 160
lazy, 160
event
aperiodic, 56
jitter, 56
non-deterministic, 4
periodic, 56
events
non-deterministic, 27
execution rule, 43, 47
execution time, 92, 158
maximum, 9, 38
minimum, 9, 38
variation, 7, 54
exploration phase, 155

firing rule, 43
FORM, 66
FORMAT, 67
ForSyDe, 66
FSM, 18
CFSM, 30
FSMD, 30
hierarchical, 30
function pointer, 28
FUNSTATE, 31

garbage collection, 28
granularity, 7, 9, 22, 62, 160, 173
gray box
extraction, 62, 158
improving, 62

halt, 18, 26
hardware/software codesign, 18, 64
HardwareC, 65
high-level estimator, 73, 91
high-performance computing, 69, 75
HYPER, 67

inheritance, 28
instruction path, 9
intra task level, 62

JAVA, 30
JPEG, 12, 84

Kahn process networks, 30

level of abstraction, 21, 65
load balancing, 7
LOTOS
timed LOTOS, 67
LUSTRE, 26

model
gray box, 52
parallelism, 57
MTG*
definition, 37
two layers, 19
modeling
computation, 18
concurrency, 23, 26
data, 20
dynamic behavior, 32
heterogeneity, 29
hierarchy, 27
liveness, 27
non-determinism, 19, 24
performance, 26
reactivity, 31
modeling non-determinism, 26
Mozart-Oz3, 31
MPEG21, 12
MTG, 9
MTG*, 9
extraction, 158
intra task level, 52
token marking, 45
multi-processor platform, 4, 5

node
active, 43
and, 46, 48
behavioral, 41
control flow, 41
data, 43
deterministic or, 46
diamond, 46, 51
dynamic task, 45, 47
event, 46, 51
hierarchical thread node, 53
operational, 41
passive, 43
postset, 45
preset, 45
stop cluster, 46, 51
sync, 46, 49
synchro, 46, 50
task, 45
thread cluster, 53
thread frame, 45, 47
interleaving, 159, 182
overlapping, 182
thread node, 53
boundary, 54
non-determinism, 24

non-manifest loop, 9
non-manifest loop bound, 54, 178

OCAPI, 30
OCCAM, 67
Olympus, 65
operational semantics, 22
    data nodes, 47
    operational nodes, 43
    task level, 45

parallelism, 63
Pareto curve, 155, 158
    3D, 165
    multiple constraints, 162
    multiple costs, 162
    quality, 160
Pareto point, 73
partial recursive function, 19
Petri net, 19
    arc, 19
    concurrency, 19
    firing, 19
    inhibitor arc, 19
    marking, 19
    non-determinism, 19
    place, 19
    token, 19
    token game, 19
    transition, 19
    variants, 26
PhD team, 4, 6
plug-in, 28, 182
POLIS, 30
polymorphism, 28
POOSL, 67
port
    cardinality restriction, 39
    control, 39
    data, 39
pre-processing phase, 155
priority
    task, 178
profiling, 91, 158, 180
Ptolemy-II, 30
push-down automata, 19

quality of service, 4, 12, 32, 74, 75

race condition, 21, 23, 24, 79
random access machine, 19
recursion, 95
    argument transformation, 98
    argument transformation function, 102, 114
    assumptions, 97
    base case, 97
    base case calculation, 97
    call trace, 108
    call tree, 98
    call tree padding, 120
    comparison iterative recursive, 107
    future work, 99, 126
    impact of transformation, 126
    in-place optimization, 123
    intermediate function calls, 109
    memoization, 123
    multiple base cases, 117, 119
    parallelization, 99
    performance, 98
    profiling, 130
    return value, 97, 124
    return value combination function, 97
    side effect, 96, 123
    side-effect, 120
    stack frame, 123
    systematic trade-off, 123
    tail recursion, 95
    Towers of Hanoi, 114
    upper bound on iterations, 108
    visual texture coding, 130
recursion removal
    call tree branch, 101
    function composition, 100, 106
    Hanoi, xxix, 114, 118
    iteration index, 105
    overhead, 106
    parallelization, 117
    path evaluation equation, 104, 105
    propagation equation, 101, 103
    recursion depth, 105
    symbolic simplification, 116, 117
recursive case, 97
regular expression, 18
reverse engineering, 71
run-time, 4, 9
    overhead, 90

safety, 26
scenario
    intra thread frame, 180
    intra thread frame subscenario, 180
    thread cluster scenario, 181
scheduling
    design-time, 7, 76, 138, 180
    run-time, 8, 53, 61, 63, 138, 179, 180, 182
    two-phase, 7, 181
SDL, 26, 30, 65
semaphore
    lock removal, 69

sequentialization, 80
semaphore removal
graph transformations, 80
lock coarsening, 69
manual restructuring, 69
synchronization map, 80
SHE, 67
side-effect, 64, 71, 72
lack of, 72
SIGNAL, 26
SIL, 67
SILAGE, 66
simulation, 4, 29, 70
SPARK, 66
SpecCharts, 26
SPI, 30
starvation, 25
state
explicit, 64
FSM, 18
Petri net, 19
waiting, 63
state space explosion, 19
streaming computation, 27, 87, 160
STRICT, 66
STRIDE, 66
SyncCharts, 26
synchronization, 26
system-on-chip, 4
system-wide trade-off, 155
SystemC, 30

task concurrency management, 6
task level, 62
task level transformation, 79
TCM
exploration, 159
overview, 6
pre-processing, 157
TCM script, 157
timing constraint, 9, 54
execution rate, 55
latency, 55
timing guarantee, 56
Tipsy, 31
trade-off, 62
synchronization, 90
TRADES, 67
transformation
abstraction, 156, 157, 159
bottleneck identification/removal, 156, 158
bottleneck identification/trade-off, 157, 160
classification, 168
complexity reduction, 156, 159, 160
freedom creation, 156–159
impact estimation, 67, 74
loop transformation, 66, 67, 69, 157
pre-synthesis, 66
source to source, 66
TCM exploration, 76, 171
TCM pre-processing, 76, 171
Turing complete, 26
type system, 28

UML, 67

Verilog, 30
VHDL, 30, 65, 67
voltage scaling, 5, 9, 139

workload, 7

YAPI, 30