Ilkka Hautala

FROM DATAFLOW MODELS
TO ENERGY EFFICIENT
APPLICATION SPECIFIC
PROCESSORS
ILKKA HAUTALA

FROM DATAFLOW MODELS TO ENERGY EFFICIENT APPLICATION SPECIFIC PROCESSORS

Academic dissertation to be presented with the assent of the Doctoral Training Committee of Information Technology and Electrical Engineering of the University of Oulu for public defence in the OP auditorium (L10), Linnanmaa, on 21 October 2019, at 12 noon

UNIVERSITY OF OULU, OULU 2019
Abstract

The development of wireless networks has provided the necessary conditions for several new applications. The emergence of the virtual and augmented reality and the Internet of things and during the era of social media and streaming services, various demands related to functionality and performance have been set for mobile and wearable devices. Meeting these demands is complicated due to minimal energy budgets, which are characteristic of embedded devices. Lately, the energy efficiency of devices has been addressed by increasing parallelism and the use of application-specific hardware resources. This has been hindered by hardware development as well as software development because the conventional development methods are based on the use of low-level abstractions and sequential programming paradigms. On the other hand, deployment of high-level design methods is slowed down because of final solutions that are too much compromised when energy efficiency and performance are considered.

This doctoral thesis introduces a model-driven framework for the development of signal processing systems that facilitates hardware and software co-design. The design flow exploits an easily customizable, re-programmable and energy-efficient processor template. The proposed design flow enables tailoring of multiple heterogeneous processing elements and the connections between them to the demands of an application. Application software is described by using high-level dataflow models, which enable the automatic synthesis of parallel applications for different multicore hardware platforms and speed up design space exploration. Suitability of the proposed design flow is demonstrated by using three different applications from different signal processing domains. The experiments showed that raising the level of abstraction has only a minor impact on performance.

Video processing algorithms are selected to be the main application area in this thesis. The thesis proposes tailored and reprogrammable energy-efficient processing elements for video coding algorithms. The solutions are based on the use of multiple processing elements by exploiting the pipeline parallelism of the application, which is characteristic of many signal processing algorithms. Performance, power and area metrics for the designed solutions have been obtained using post-layout simulation models. In terms of energy efficiency, the proposed programmable processors form a new compromise solution between fixed hardware accelerators and conventional embedded processors for video coding.

Keywords: application-specific processing, dataflow modelling, dataflow-based design framework, energy-efficient computing, video coding
Hautala, Ilkka, Datavuopohjainen suunnitteluun energiatehokkaille sovelluskohtaile prosessoreille. Oulun yliopiston tutkijakoulu; Oulun yliopisto, Tieto- ja sähkötekniikan tiedekunta; Infotech Oulu

*Acta Univ. Oul.* C 714, 2019
Oulun yliopisto, PL 8000, 90014 Oulun yliopisto

**Tiivistelmä**


Asiakirja esittelee datavuopohjaiseen suunnittelun perustuva työkaluketjun, joka on tarjottu energiatehokkaiden signaalikäsittelyjärjestelmien toteuttamiseen. Työssä esitetään suunnitteluvuoro mahdollistaa useiden heterogeeneiden prosessorityypien ja niiden välisen sisällön räättöömillä sovelluksellaan. Mahdollisesti uusia energiatehokkaita ratkaisuja tarjotaan uusille sovelluskohtaile prosessoreille.

**Asiakirjat:**
- datavuomallinnus, datavuopohjainen suunnittelu, energiatehokas laskenta, sovelluskohtainen laskenta, videonkoodaus
To Hilda, Valma and Ilmari
Preface

This thesis was carried out in the Center for Machine Vision and Signal Analysis (CMVS) research group of the University of Oulu, between 2013 and 2019.

First of all, I would like to express my deepest gratitude to my principal supervisor Associate Professor Jani Boutellier for invaluable help, ideas, guidance, support and encouragement during my doctoral studies. I would not even have started my Ph.D. studies if he hadn’t introduced me to the world of application-specific processors during a master’s degree course in the spring 2012.

I would also like to thank Professor Olli Silvén, who has given insightful feedback to me and helped me understand the big picture of the field. I appreciate that he as my supervisor has let me to fully concentrate on my Ph.D. research. His positive and encouraging attitude have been restorative for me.

I am grateful to Dr. Jari Hannuksela, who hired me to CMVS in 2012 and co-authored a couple of my first publications. I would also like to thank Dr. Teemu Nyländen, Dr. Timo Viitanen, and Dr. Jukka Lahti for sharing their professional expertise and knowledge and helping me to improve the content of the publications.

My warmest thanks to Prof. Karol Desnos and Prof. Guillermo Payá-Vayá, who have made a big effort by reviewing this study. I am grateful for Gordon Roberts for the language revision. Also I would like to thank all members of my follow-up group: Professor Janne Heikkilä, Dr. Esa Rahtu, and Dr. Janne Janhunen.

The financial support received from the Infotech Oulu Graduate School, University of Oulu Graduate School, Academy of Finland project ICONICAL, 6 Genesis Flagship project and Tauno Tönning Foundation is gratefully acknowledged.

Warmest thanks to my friends Dr. Juha Ylioinas, Dr. Markus Ylimäki, Dr. Sami Huttunen, and Mehdi Safarpour for various deep discussions during daily breaks and sporting activities. Furthermore I would like to thank all my friends, who have asked the question: “When is your thesis ready?” in the recent years for pushing me towards completing this thesis.

I want to express my deep appreciation for my parents Sinikka and Pertti for their sincere love and support throughout my life. They have always emphasized the significance of education and urged me to study ever since I started primary school. I want to thank all my siblings. Thanks to Janne for leading me to computers when I was
a kid. Thanks to Karoliina for being the best sister ever. Thanks to Markus for being my wrestling companion. Furthermore special thanks to my sister’s husband Janne for being my ”third brother”. Finally, I want to thank my dear wife Kaisa for her support and love and my dearest children Hilda, Valma and Ilmari for the joy that they bring everyday in my life.

Oulu, September 2019
Ilkka Hautala
**List of abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4G</td>
<td>4th Generation</td>
</tr>
<tr>
<td>5G</td>
<td>5th Generation</td>
</tr>
<tr>
<td>ADF</td>
<td>Architecture Definition File</td>
</tr>
<tr>
<td>ALF</td>
<td>Adaptive Loop Filter</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AR</td>
<td>Augmented Reality</td>
</tr>
<tr>
<td>ARM</td>
<td>Advanced RISC Machines</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASP</td>
<td>Application Specific Processor</td>
</tr>
<tr>
<td>AV1</td>
<td>Alliance for Openmedia Video 1</td>
</tr>
<tr>
<td>AVC</td>
<td>Advanced Video Coding</td>
</tr>
<tr>
<td>BDF</td>
<td>Boolean Dataflow</td>
</tr>
<tr>
<td>CABAC</td>
<td>Context Adaptive Binary Arithmetic Coding-algorithm</td>
</tr>
<tr>
<td>CAL</td>
<td>Cal Actor Language</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off-the-shelf</td>
</tr>
<tr>
<td>CSDF</td>
<td>Cyclo-Static Dataflow</td>
</tr>
<tr>
<td>CTU</td>
<td>Coding Tree Unit</td>
</tr>
<tr>
<td>CU</td>
<td>Coding Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DAL</td>
<td>Distributed Application Layer</td>
</tr>
<tr>
<td>DBF</td>
<td>De-Blocking Filter</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DD</td>
<td>Data-Driven / Demand-Driven</td>
</tr>
<tr>
<td>DFG</td>
<td>Dataflow Graph</td>
</tr>
<tr>
<td>DLP</td>
<td>Data-Level Parallelism</td>
</tr>
<tr>
<td>DPD</td>
<td>Digital Predistortion Filter</td>
</tr>
<tr>
<td>DPN</td>
<td>Dataflow Process Network</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>Acronym</td>
<td>Full Form</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DVB</td>
<td>Digital Video Broadcasting</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>eDRAM</td>
<td>Embedded DRAM</td>
</tr>
<tr>
<td>ELS</td>
<td>Electronic System Level</td>
</tr>
<tr>
<td>EPIC</td>
<td>Explicitly Parallel Instruction Computing</td>
</tr>
<tr>
<td>FD-SOI</td>
<td>Fully Depleted Silicon On Insulator</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>FinFET</td>
<td>Fin Field-Effect Transistor</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPS</td>
<td>Frame Per Second</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FU</td>
<td>Function Unit</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General-Purpose computing on Graphics Processing Units</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
</tr>
<tr>
<td>HDTV</td>
<td>High Definition Television</td>
</tr>
<tr>
<td>HDB</td>
<td>Hardware Database</td>
</tr>
<tr>
<td>HE</td>
<td>Horizontal Edges</td>
</tr>
<tr>
<td>HEVC</td>
<td>High Efficiency Video Coding</td>
</tr>
<tr>
<td>HM</td>
<td>HEVC Test Model</td>
</tr>
<tr>
<td>HMA</td>
<td>Hybrid Memory Architecture</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>ICG</td>
<td>Integrated Clock Gating</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction-Level Parallelism</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IPC</td>
<td>Instruction Per Cycle</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
</tr>
<tr>
<td>ITU-T</td>
<td>International Telecommunication Union - Telecommunication standardization sector</td>
</tr>
<tr>
<td>JCT-VC</td>
<td>Joint Collaborative Team on Video Coding</td>
</tr>
<tr>
<td>JVET</td>
<td>Joint Video Experts Team</td>
</tr>
<tr>
<td>KPN</td>
<td>Kahn Process Networks</td>
</tr>
<tr>
<td>L1</td>
<td>Layer-1</td>
</tr>
<tr>
<td>L2</td>
<td>Layer-3</td>
</tr>
<tr>
<td>L3</td>
<td>Layer-3</td>
</tr>
<tr>
<td>LD</td>
<td>Low Delay</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low Level Virtual Machine</td>
</tr>
<tr>
<td>LP</td>
<td>Low Power</td>
</tr>
<tr>
<td>LSU</td>
<td>Load Store Unit</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>MAPS</td>
<td>MPSoC Application Programming Studio</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Controller Unit</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple-Input and Multiple-Output</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multiprocessor System on Chip</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Experts Group</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>OpenCV</td>
<td>Open Source Computer Vision</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PDF</td>
<td>Parameterized Dataflow</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PiSDF</td>
<td>Parametrized and interfaced SDF</td>
</tr>
<tr>
<td>Prode</td>
<td>Processor Designer</td>
</tr>
<tr>
<td>PPA</td>
<td>Power, Performance and Area</td>
</tr>
<tr>
<td>PU</td>
<td>Prediction Unit</td>
</tr>
<tr>
<td>QP</td>
<td>Quantization Parameter</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>RA</td>
<td>Random Access</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RF</td>
<td>Register File</td>
</tr>
<tr>
<td>RGB</td>
<td>Red, Green and Blue color model</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RR</td>
<td>Round Robin</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RVC</td>
<td>Reconfigurable Video Coding</td>
</tr>
<tr>
<td>RW</td>
<td>Read and Write</td>
</tr>
<tr>
<td>SAD</td>
<td>Sum of Absolute differences</td>
</tr>
<tr>
<td>SAO</td>
<td>Sample Adaptive Offset filter</td>
</tr>
<tr>
<td>SDE</td>
<td>Stereo Depth Estimation</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Dataflow</td>
</tr>
<tr>
<td>SFU</td>
<td>Special Function Unit</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Threads</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multithreading</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static Random Access Memory</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>UMA</td>
<td>Uniform Memory Access</td>
</tr>
<tr>
<td>TCE</td>
<td>TTA-based Co-Design Environment</td>
</tr>
<tr>
<td>TLP</td>
<td>Task-Level Parallelism</td>
</tr>
<tr>
<td>TSMC</td>
<td>Taiwan Semiconductor Manufacturing Company Limited</td>
</tr>
<tr>
<td>TTA</td>
<td>Transport Triggered Architecture</td>
</tr>
<tr>
<td>TU</td>
<td>Transform Unit</td>
</tr>
<tr>
<td>UHDTV</td>
<td>Ultra High Definition Television</td>
</tr>
<tr>
<td>VCEG</td>
<td>Video Coding Experts Group</td>
</tr>
<tr>
<td>VE</td>
<td>Vertical Edges</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Reality</td>
</tr>
<tr>
<td>VVC</td>
<td>Versatile Video Coding</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
A  Set of actors
a  Actor
α  Activity factor
B  Block size
C  Set of FIFO interconnection channels
c  FIFO channel
C_l  Load capacitance
D  Directed graph
d  Disparity offset
E  Edge
D_{max}  Maximum disparity range
F_a  Set of Firing function of actor a
f  firing function
f_{clk}  Clock frequency
I  Image
I(x,y)  Image pixel at point (x,y)
I_Y  Luminance channel of image
I_R  Red channel of image
I_G  Green channel of image
I_B  Blue channel of image
I_D  Disparity image
I_{leakage}  Leakage current
I_{sc}  Short circuit current
n  Number of parallel resources
V  Voltage
V_{dd}  Supply voltage
P  Power
P_a^-  Output ports of actor A
P_a^+  Input ports of actor A
R_a  Set of firing rules of actor a
r_i  firing rule i
r_p  Ratio of parallel portion of task
r_s  Ratio of sequential portion of task
s_p  Speedup of portion of task
t  time
List of original publications

This thesis is based on the following articles, which are referred to in the text by their Roman numerals (I–V):


## Contents

Abstract

Tiivistelmä

Preface

List of abbreviations

List of original publications

Contents

1 Introduction

1.1 Background and motivation .............................................. 23
   1.1.1 Model-driven engineering ...................................... 25
   1.1.2 Y-chart design methodology ................................. 27

1.2 Scope of the thesis .................................................. 27

1.3 Contributions of the thesis ......................................... 30

1.4 Summary of original publications ................................. 31

1.5 Outline of the thesis ................................................ 31

2 Compute intensive stream applications ......................... 33

2.1 Video coding standards ............................................. 33

2.2 Overview of high efficiency video coding ....................... 36
   2.2.1 In-loop filters .................................................. 37

2.3 Wireless communication ............................................. 40

2.4 Digital image processing .......................................... 42

2.5 Conclusion .................................................................. 43

3 Describing applications using dataflow graphs ............. 45

3.1 Dataflow programming ............................................... 45
   3.1.1 Dataflow parallelism .......................................... 48
   3.1.2 Synchronization .................................................. 49
   3.1.3 Maintainability ................................................... 49
   3.1.4 Modelling granularity ......................................... 50

3.2 Dataflow model of computations ................................. 53

3.3 Parallel programming frameworks ............................... 58
   3.3.1 Dataflow modelling frameworks ............................. 59
   3.3.2 Low-level parallel programming APIs ..................... 63

19


1 Introduction

1.1 Background and motivation

In the era of the social media, the Internet of Things (IoT) and the virtual reality (VR), increasing complexity and heterogeneity of applications has introduced numerous challenges for both the hardware and software design of embedded mobile platforms. Future applications can have tight real-time requirements making high latencies on data processing and radio communication channels unacceptable [1]. Sometimes, complex tasks can be offloaded to be processed by cloud servers over a radio link, but this is not always possible due to issues related to energy [2], latency, reachability or security, and therefore data processing has to be performed in the mobile device.

Video streaming is an example of an application, where the decoding or encoding have to be performed in the device since the video has to be compressed to enable video transferring over a bandwidth-limited channel. The consumption of video material has increased greatly over the last few years and will continue to do so in the future. Ericsson's mobility report [3] from June 2018 estimates that the sum of all forms of video is over 73% of mobile data traffic in 2023 which means 107 exabytes per month. Based on this, it is evident that efficient video coding and wireless communication technologies are playing a critical role in handling growing amount of video data in mobile devices.

The high-end video coding standards such as High Efficiency Video Coding [4] (HEVC, H.265/MPEG-H) by the Joint Collaborative Team on Video Coding (JCT-VC) and its main competitors, the open and royalty-free VP9 [5] by Google and AOMedia Video 1 (A V1) [6] by Alliance for Open Media can reduce bandwidth requirements to half when compared to their predecessors. Consequently, the evolved compression methods require high computational performance and parallel data processing capabilities for high video resolutions [7].

The 4G long-term evolution (LTE) cellular network standard has been the key enabler for high definition mobile video applications [8]. Whereas, the upcoming 5G standard is going to offer higher data rates and lower latencies to enable applications such as Ultra High Definition video (UHD) and Virtual Reality (VR) space for mobile devices [1, 9]. 5G introduces Massive MIMO (multiple-input multiple-output) technology [10], where
tens or hundreds of parallel receivers and transmitting antennas are used to transfer data signals over the same radio channel [11].

Both video coding standards and wireless communication standards are examples of applications that comprise numerous complex digital signal processing algorithms [7]. Increasing complexity, heterogeneity, and parallelism of the applications combined with high design costs, the shrinking time-to-market window and tight energy requirements set new challenges for software and hardware development.

IC manufacturing costs are skyrocketing, when moving towards smaller technology nodes. Fig. 1 shows an advanced integrated circuit (IC) design cost when different manufacturing technologies are used. For commercial success, correct timing of product rollout is critical. Fig. 2 shows the schedule that was planned for 5G wireless communication development and deployment. It is noteworthy that the development window of the product is very short if the product development starts after the standard matures. By exploiting programmable platforms, product development can be started ahead of time before the final standard is approved. Solutions have to introduce more flexibility than before to be economically reasonable in the future.

Power consumption is one of the critical aspects when designing mobile devices. In the case of mobile phones or tablets, it could be satisfactory that the battery lasts one working day. Instead, some other applications can have a requirement that battery lifetime is months or years rather than days. However, the more critical issue is managing of the heat dissipation of future devices. High-end semiconductor manufacturing processes allow the packing millions of nanoscale transistors in one square millimeter. For example, Intel’s 10 nm FinFET technology can pack over 100 million transistors in
each square millimeter of a chip [13]. However, the threshold and operating voltage scaling of the chip have not kept up the same pace as the feature size scaling driven by Moore’s law. Due to that the power density and thermal hotspots of the chips have been increased, which is also known as the “dark silicon” problem. For example, in the case of 8 nm technology, there are predictions that only 50% of transistors of the chip can be powered on at the same time because of power density problems [14].

One way to mitigate the “dark silicon” problem is to exploit heterogeneous parallel architectures and application-specific hardware [17, 18]. The disadvantage of the approach is that programming of heterogeneous platforms is challenging [18, 19], since the designer must deal with varying programming models, software partitioning into multicore general purpose processors, and tailored accelerators with their diverse compute capabilities, their complex interconnects and memory architectures.

Widely used sequential software programming methodologies are ineffective for describing parallel applications, and porting them onto different hardware platforms is challenging. On the other hand, conventional hardware design methods are based on register-transfer level (RTL) languages, to describe functionality in low-level cycle-by-cycle. The low-level design methods are time-consuming, error-prone and also hamper design space exploration (DSE): “the task to explore the set of feasible implementations efficiently and finding not only one of these but many and also optimal ones” [20]. Automated design flows can be employed for efficient DSE, design verification and management of massive parallelism in order to reduce design cost. Lately, model-driven engineering has been exploited to foster automation of the design flows of signal processing systems [21, 22, 23].

1.1.1 Model-driven engineering

In general, a model describes a system without using the system itself, being an approximation of the system [24]. This thesis employs engineering models, which
are tapping into building and describing the human-made systems and processes. The models help to understand how the real systems are working by permitting exploration of the interesting properties of the systems without getting hands on the actual systems [24]. This thesis adopt definition of model from [25]:

“A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made.”

A model is an instance of a metamodel, which defines the language expressions that can be used for modelling a model [25].

In this thesis, system refers to an entity which consists of hardware and software components, that are needed to implement some desired functionality. The same system can be modelled at different levels of detail. A model A can be an abstraction of a model B if the model B describes the system with a lower level of detail than the model A. If model B is a refinement of model A then the model B adds details to the model A in a way that new properties are not in contradiction with the B’s. [24]

Abstraction is a powerful technique to raise productivity [26] and it has been also shown that optimization performed at a higher level often have more leverage than optimization at low levels of abstraction [27]. An example of exploiting abstraction is a compiler, which can translate a high-level description of a program (e.g., C-code) into a low-level machine language.

Actually, what the compiler does is synthesis. In synthesis, initial models are derived from a specification of a system, and then the models can be refined until the actual system is constructed. By using automation for refining, design efficiency is greatly improved and design errors can be avoided [28, 29].

Model-Driven Engineering (MDE) is an engineering technique to solve complex problems by generating implementations semi-automatically, or automatically from the models which are the primary artefacts of development processes [30]. In MDE, the system is typically specified using domain-specific modelling languages that often offer high-level abstractions for the desired domain [25]. Consequently, systems are less dependent on implementation technology and closer to the application domain [29]. This also permits non-platform specialists to implement systems, since specifying and maintaining models is simplified. Typically, models can be visualized, which helps designers and project managers to understand complex problems, and therefore improve communication about the problem.
Generators and transformers have a key role in MDE. They can analyse critical properties of models and use the models to synthesize source code, simulation test-bench, new models or other desired types of output. If the generators and transformers are functioning correctly, then refined models produced in a way from an initial model into a final implementation are functionally consistent with the initial model. Consequently, error-prone and tedious manual refining of models to implementation can be avoided by using the generators [29].

1.1.2 Y-chart design methodology

To satisfy tight design constraints, Hardware/Software (HW/SW) Co-design, the concurrent design of hardware and software components of the system, is employed [20]. Many HW/SW Co-design flows employ the widely used Y-chart system design methodology, which relies on the use of formal models and transformations [31, 32, 33]. Fig. 3 illustrates the Y-chart design methodology. In the Y-chart approach, the application is typically described using a formal model that captures the functionality of the application without reference to a possible hardware architecture which is run by the software. That is, the approach makes software and hardware architecture design independent from each other, which allows the use of high-level abstractions to describe the application without the need to considering timing, HW/SW partitioning or excluding any interesting implementation options in early design phases.

1.2 Scope of the thesis

This thesis contributes to different phases of the design flow of the signal processing systems which demand high performance, high energy efficiency, flexibility of software programmability, and low design cost.

The thesis aims to provide solutions for stream signal processing applications, where there is an infinite flow of data from the input which is orchestrated through the different processing steps to the system output. Typically, stream applications are compute intensive, parallel and data local. This thesis has mainly focused on HEVC video decoding algorithms (Paper I and Paper III), but also considers other algorithms from the wireless communication and the computer vision domain (Paper V).

From a software perspective, the thesis focuses on the dataflow models which are used for the functional specification of an application. Dataflow-based application
modelling ensures modular, re-usable and concurrent description of the application that simplifies mapping of the application for different hardware architectures. Moreover, the rigorous formalism of dataflow-based design makes system analysis and verification easier. On the other hand, the use of high-level models can cause a performance penalty for various reasons. In that context, this thesis explores ways to shrink the performance gap between the traditional language-based implementations and dataflow-based implementation (Paper IV and Paper V).

From a hardware perspective, the thesis defines its own metamodel for describing hardware architecture at the system level. The hardware architecture model defines the hardware resources of a system that can be run by software. The model allows the presentation of processing elements (PEs), the interconnection between them and the memory components.

The thesis focuses on software programmable, customizable PEs that are based on the Transport Triggered Architecture (TTA) template, whose metamodel allows several customization points. In this context, the suitability of the TTA processor template is investigated for energy efficient and high performance embedded signal processing.
systems. The individual PEs are especially tailored with video coding algorithms in mind. Moreover, the advantages of employing multiple PEs are investigated by focusing on the task and pipeline parallelism of the applications. The TTA-based solutions are compared against general embedded GPPs, GPUs and ASICs. (Paper I, Paper II and Paper III)

The software and the hardware are bound together to form a system configuration. This phase comprises mapping and scheduling. Mapping assigns the different functional blocks of the software model onto the hardware resources. The goal of scheduling is defined as ordering relations between computational operations. Mapping and scheduling are considered to be the context of dataflow and are on the system level: mapping dataflow actors onto PEs and scheduling dataflow actors in PE. In Paper IV, mapping and scheduling are handled on platforms where applications are executed on top of the operating systems (OS), whereas, Paper V addresses dataflow mapping and scheduling in systems which consist of TTA PEs without an OS and GPP PEs with an OS.

The goal of the analysis is to provide information about the feasibility of the system configuration. The system configuration is feasible if it meets the requirements of the implementation constraints. The implementation constraints can be performance, power consumption, design area, temperature, etc. This thesis focuses on discovering performance, power, and area (PPA).

Based on information provided by analysis, it is possible to iteratively improve the system configuration to meet implementation constraints by modifying the system architecture model, the application model or the mapping. This process is called design space exploration (DSE) and is illustrated by feedback loops in Fig 3. Full automation of the DSE is beyond the scope of the thesis; rather, the thesis aims to provide rapid design flow to produce and analyse different system configurations. The analyses are mainly conducted by simulating the system at different levels of abstraction. For performance simulations, cycle-accurate C++ abstractions of hardware architecture are used (Paper I and Paper V), whereas power simulations were executed for the placed and routed implementation (Paper II) which holds realistic timing and physical information.

In the thesis, implementation constraints are set so that the final implementation should be energy efficient and cost efficient. Energy efficiency refers to the consumed energy per task, like joules consumed for filtering a single pixel. For example, in the context of the video coding, the thesis targets solutions which allows real-time (25 frames per second) decoding of Full HD (1920 × 1080) video frames with a suitability of
less than 1.5 W power consumption for mobile devices [34]. Cost efficiency refers to the design time in relation to performance or power, and also considers system flexibility. In this thesis, flexibility is provided by concentrating software programmable architectures whose functionality can be changed by using a high-level software code.

1.3 Contributions of the thesis

The thesis presents application specific programmable processors with an energy efficiency close to non-programmable fixed solutions. Moreover, the thesis proposes a high-level dataflow-based design-tool, which can be used for the automatized design of a multicore processing solution consisting of the aforementioned energy efficient processors and their software. The main contributions of the thesis are listed below.

– The dataflow-based design framework for the transport triggered architectures, which integrates ASP development and C-language based dataflow programming for the design of power-efficient and high-performance multicore ASPs and their software.
– A multicore runtime for dataflow process networks and extensive performance benchmarks using desktop, server and mobile multicore platforms with applications from different domains.
– Coding tree unit (CTU) based memory efficient algorithm implementations are presented for the de-blocking filter and SAO that are suitable for embedded devices.
– A memory efficient CTU-row based algorithm implementation for HEVC adaptive loop filter (ALF) is proposed.
– Efficient dataflow presentations of the HEVC In-loop filters and stereo depth estimation algorithms.
– Application-specific programmable multicore processor architecture for accelerating HEVC in-loop filters is proposed, namely de-blocking and sample adaptive offset filter.
– Extensive experiments that show the performance and power consumption figures of the proposed multicore processors on place and routed 28 nm standard cell technology.
– A programmable application specific processor for computationally expensive HEVC ALF is proposed. The architecture includes several special function units to accelerate ALF.
1.4 Summary of original publications

The contributions listed above were originally published in five separate articles. Reprints of the articles are included in the appendix of the thesis. Below, the content of the articles is summarized.

Paper I presents a programmable coprocessor architecture for HEVC in-loop-filtering. The implementation consists of three application-specific instruction set processor cores that are connected to a dataflow in a manner enabling pipelined processing. The paper proposes a coding tree block based in-loop-filtering algorithm that is suitable for memory-constrained embedded platforms and several special functional units for the processor architecture to accelerate the filtering.

Paper II extends the work of Paper I by presenting place and routed design using 28 nm standard cell technology. The paper also presents accurate performance and power consumption evaluations of the proposed implementation of HEVC in-loop filtering.

Paper III presents a programmable application specific processor for the adaptive loop filter (ALF), which was considered to be a part of the HEVC standard but was finally left out. Still, it is very probable that the ALF will be adopted in upcoming the Versatile Video Coding standard [35]. The paper proposes that the ALF algorithm processes coding tree blocks row by row in raster scan order. The paper also proposes several special function units to accelerate ALF.

Paper IV proposes a runtime for executing dataflow process networks on multicore platforms. The work investigates the effects of the CPU affinity in a dataflow context and proposes an operating system based load-balancing instead of limiting the thread migration between processing cores.

Paper V presents a HW/SW co-design flow starting from dataflow descriptions to the place and route. The paper experiments with the design flow by using the same application as Paper I, and two other applications from the field of machine vision and wireless communication.

1.5 Outline of the thesis

The thesis consists of an overview and an appendix, which contains the original publications. The overview part is organized according to the Y-chart flow. First, Chapter 2 outlines the demands and requirements of the stream based signal processing
applications and presents some recent applications in detail. Chapter 3 concentrates on the dataflow programming model by starting with a short historical review. After that, different types of dataflow models of computations and dataflow frameworks are presented. Chapter 4 deals with energy efficient computation platforms, and presents two application tailored solutions for the video coding algorithms. The mapping and scheduling of dataflow graphs into processing elements of the system is discussed. Chapter 5. Chapter 6 builds on the previous chapters, and presents a dataflow-based design framework for targeting multicore solutions consisting of heterogeneous and programmable energy efficient processors. Finally, Chapter 7 draws conclusions from the work. Contributions are presented in Chapters 3-6 and highlighted to the reader by using headings that start by the Contribution prefix (e.g. Contribution: CTU-based dataflow description in-loop filtering).
2 Compute intensive stream applications

Evolving video coding and wireless telecommunication techniques will have a significant impact on the emergence of new applications in the future. Already, these techniques have enabled applications that affect our everyday life. For example, we can watch a live broadcast of a football match, Youtube videos or movies by using a mobile device at the same time as we are traveling on a train or even on an airplane. The video content share of total mobile data traffic is estimated to be about 70% in 2023 with an annual growth of 45% [3]. To this end, it is clear that the complement of video coding techniques and communication networks have an essential role in empowering future applications.

Video coding and radio baseband processing are examples of areas where digital signal processing (DSP) is traditionally utilized. The applications in this field have tight real-time requirements. For example, when a user is watching a football match, the DSP system of a mobile device has to receive and decode video data-packets at least about 25 frames in every second. If there are errors in data transmission or the video decoding resources of the device are limited for some reason, the user could perceive lagging video.

This chapter presents applications that are implemented in this thesis. The aim of the chapter is to inform the reader about the requirements and characteristics of the applications. First, different video coding standards are briefly discussed and relevant video coding algorithms in the context of the thesis are presented. After that, a wireless communication application is presented. Finally, a depth estimation algorithm for machine vision or image processing applications is presented.

2.1 Video coding standards

The goal of video standards is to provide specifications for the video bitstream, syntax and decoder. Currently, there are several different video coding standards available, starting from 1984 when the first digital video standard H.120 by ITU-T was published [36]. Over the years different groups have developed the standards but the two main players in standardization efforts have been the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). The standards of the organizations contain patent covered technologies of several companies. As a countermeasure, companies like Google have lately invested in royalty-free video coding
techniques by acquiring the long-standing video coding company On2 Technologies and their VP8 video codec.

VP8 can be considered to be a rival for the MPEG4/AVC standard. The successors for these standards are Google’s VP9, published in late 2012 and HEVC (H.265 MPEG-H Part 2), published by MPEG and VCEG in early 2013. In 2015, tens of leading technology companies like Amazon, Intel, Microsoft, Mozilla, Nvidia, and Netflix allied with Google and founded the Alliance For Open Media (AOMedia) with a view to the development of royalty-free open video standards and competing with the MPEG standards [37]. The first project of AOMedia is AOMedia Video 1 (AV1), whose initial release was published in early 2018 and it is planned to be a royalty-free alternative for HEVC [37].

In late 2015, the Joint Video Experts Team (JVET) was formed by VGEC and MPEG to develop the successor to HEVC, which is called Versatile Video Coding (VVC) [35]. The first version of the VVC is planned to be ready in 2020 [38]. VVC is targeting applications like 8K UHDTV, virtual reality, multi-view, and 360° videos [38].

The lifetime of a video standard can be quite long. The MPEG-2 standard [39] published in 1994, is still widely used. That is because TV broadcasters have to be conservative in order to provide support for old terminal devices of their customers. Moreover, the low processing overhead of MPEG-2 enables very low latency encoding, making it suitable for live streaming. On top of everything else, MPEG-2 patents expired in 2018 [40], which actually can increase the popularity of MPEG-2. Another popular standard is H.264/AVC [41], the first version being published in 2003, and the 25th version of the standard was published in April 2017. Over the years, numerous extensions have been added to the H.264/AVC. H.264/AVC will not disappear soon, and support for it will continue to be needed [42].

Adoption of the HEVC standard has been slow. The Streaming Media Magazine’s [42] survey for service providers (2018) points out that one of the main reasons for its slow adoption is the licensing policy of HEVC, but low device support also hinders the adoption. Fig. 4 represents the development and adoption of the HEVC so far. The first implementations of HEVC were software based, but before the first version of the standard was ratified, fixed functioned non-commercial hardware implementation of the HEVC working draft four [43] was already proposed. About two years after that, Qualcomm introduced mobile SoC, with hardware encoding support [44] and later followed with Android OS support [45] to enable the use of hardware video accelerators. Intel packed the fixed HEVC hardware accelerators inside the Skylake desktop processor
Fig. 4. Evolution of the HEVC standard. Source material adopted from various sources.

[46] family in the year 2015, but operating system level support to enable the use of the accelerators was only deployed for Windows 10 and iOS two years later in 2017.

The basic technical principles behind video coding have remained similar over the years [36]. The current state of the art video coding standards are based on techniques like motion compensation, block based discrete cosine transform, scalar quantization and entropy coding, which were already present in H.261 published in 1988 [47]. Due to the long life of the standards, multiple standards are used and have to be supported alongside MPEG2 to HEVC. Fortunately, video coding standards are sharing similar types of computational elements which eases the implementation of systems with multi-standard support.

Increasing computing performance has rendered possible to add more coding tools for every generation of standards, and coding efficiency has improved significantly over the years. New generations of standards have halved the bitrate requirements in the same video quality when compared to their predecessors [38, 4, 41]. On the other hand, video decoder complexity has roughly doubled at maximum [48, 49] and video encoder complexity has increased an order of magnitude [50] in every generation. In addition to that, new applications have demanded higher video resolutions, which have significantly
increased computational requirements. Also, temporal resolution and colour resolution are increasing and adding more complexity.

Typically video coding is implemented in devices like set-top units, smartphones, smart televisions, tablets, cameras, and desktop computers. The growing segment of video playback-capable devices includes wearable devices like AR-glasses and smartwatches [51]. Depending on the device, the requirements on the performance, power budget and form factor can vary greatly. For example, in the case of desktop computers or televisions, high video resolutions like 4K and beyond are expected, but without tight requirements on form factor or power budget. A television can consume over 100 watts, whereas wearable devices can have a very strict, less than 100 mW power budget [52]. The wearable Google Glasses is reported to consume about 3 Watts for video recording, which is causing heating of the glasses to over 28 °C above its environment [53]. This is near the typical power consumption of smartphones, and imposes thermal issues for small form factor devices which have contact with human skin.

2.2 Overview of high efficiency video coding

The first approved version of the High Efficiency Video Coding (HEVC/H.265) standard [4] was released in April 2013 as a result of the work of the Joint Collaborative Team on Video Coding (JCT-VC), the partnership program of the International Telecommunication Union (ITU-T VCEG) and the international standardization organization (ISO IEC MPEG). HEVC is the successor of H.264/MPEG-4 AVC, and can offer over 50% better coding efficiency [7]. One of the main focuses of the standard was the enabling of 4K (resolution 3840 × 2160 pixels) video applications. High video resolutions combined with complex coding algorithms require parallel computation resources [54]. Fortunately, this was taken into consideration during the development of the standard, and HEVC introduces coding tools to alleviate parallelization.

A simplified block diagram of the HEVC decoder is shown in Fig. 5. An encoded bitstream is input to the decoder, whose first task is to perform entropy decoding. HEVC uses a context-adaptive binary arithmetic coding-algorithm (CABAC) for entropy coding. After entropy coding, an approximation of the original residual picture is formed. The transformation coefficients are derived using dequantization, and inverse transformation (inverse discrete cosine/sine transform) is applied to get the residual picture. After that, an intra or an inter picture is added to the residual picture. Finally, in-loop filters are
applied to the picture, and it is saved to the picture buffer. The picture buffer is needed for the motion compensation process when the future pictures are decoded.

**Coding structure**

The *Coding tree unit* (CTU) is a basic unit of HEVC coding, and it can be used to describe the whole coding process. A picture is divided into CTUs with a size that can be up to $64 \times 64$ (see Fig. 6 a)). The CTUs are further divided into coding units (CUs) with a size that can vary from $8 \times 8$ up to $64 \times 64$. Finally, CUs are divided into transformation units (TUs) and prediction units (PUs).

*Slices* are sequences that are composed of a variable number of consecutive CTUs in raster scan order (see Fig. 6 b)). *Tiles* are a feature that improves the support for parallel processing. A tile is a rectangular area which consists of multiple CTUs (see Fig. 6 c)). Typically, each tile contains an equal number of CTUs, except for those tiles which reside on picture boundaries and they can contain a differing number of CTUs.

### 2.2.1 In-loop filters

In this thesis, special focus is on the efficient implementation of the HEVC in-loop filters. Therefore, this section gives a brief introduction to in-loop filters.
It is common that some filtering is performed on a video signal to enhance the video quality or improve the coding efficiency. Filtering can be post-filtering or in-loop filtering. Post-filters are not defined in standards, and they can be implemented in several ways before the picture is displayed on a screen. In contrast, in-loop filters are part of the video encoding and decoding loop (see Fig. 5), and thus the filters should strictly obey the specifications of the standard. The HEVC standard defines two different in-loop filters called the de-blocking filter (DBF) and the sample adaptive offset filter (SAO). In addition to that, HEVC working drafts includes a third in-loop filter called the adaptive loop filter (ALF) [55]. The ALF was left out of the final HEVC standard because it is high complexity [56]. However, it has been proposed that HEVC’s successor, Versatile Video Coding (VVC) is going to adopt ALF as part of the standard [35]. Detailed descriptions of the filters are given in Paper I and Paper III, so only brief descriptions of the filters follow.

**The de-blocking filter**

Block-based prediction and transform coding causes discontinuities into block boundaries and that phenomenon is called blockiness [57]. Blockiness is typical for areas where coarse quantization is used. Coarse quantizations increase the prediction error. In the case of inter-prediction, the use of different prediction methods between adjacent blocks could also introduce discontinuities. Human vision is more sensitive to blockiness when texture on block boundaries is smooth, whereas, in the case of varying texture, blockiness is more difficult to notice.

The de-blocking filters exploit the aforementioned characteristic of human vision by determining block boundaries that are to be filtered and what is the appropriate filter strength for that particular boundary. Too strong a filter could smooth textures so that details are lost, whereas too weak a filter leaves blockiness visible. In the case of the
HEVC de-blocking filter [57], the filtering decision and strength of the filter are based on the pixel properties on the block boundary and the coding parameters which indicate what kind of blockiness could be present. For a detailed DBF algorithm description, the reader is kindly asked to refer Paper I or [57].

The sample adaptive offset filter

Transform coefficients of big transform blocks can have quantization errors that induce a ringing effect on a picture. In addition, the increased number of coefficients used in the HEVC interpolation filters amplifies the ringing effect. To address this problem the Sample Adaptive Offset Filter (SAO) has been adopted into the HEVC standard. [58]

SAO aims to reduce the mean sample distortion of a region. Samples of the region of interest are classified to multiple different categories using a predefined classifier. For each of the categories, the encoder calculates an offset value which is the mean difference between the original and reconstructed samples. The offset value of the category is then added to every sample which is part of that particular category. Two different types of SAO filter are adopted in HEVC; both of them are described in detail in Paper I.

The adaptive loop filter

The adaptive loop filter (ALF) is a Wiener-based filter [55], which improves objective and subjective picture quality. The purpose of using the ALF is to reduce the coding error caused by the preceding coding phases. ALF is based on minimizing the mean square error between the original and filtered samples, and the filter adapts depending
on the location of samples or the local properties of samples. Samples can be classified into different categories based on the adaptation mode. For every category, a single filter is selected. In the HM 7.0 version of ALF, samples can be classified into 16 different categories, and therefore 16 different filters can be used.

HEVC HM 7.0 uses a filter shape called the cross $9 \times 7$ square $9 \times 7$ [59]. The filter shape is presented in Fig. 7 a). The filter has 19 (9 bit) factors, but thanks to symmetry only ten different factor values. Fig. 7 b) illustrates the region adaptive adaptation mode, where samples are classified into 16 different categories depending on the sample’s spatial location.

Fig. 8 demonstrates the complexity of ALF by presenting the filtering flow of one sample. To filter one sample 19 different sample values are used. Every sample is multiplied by corresponding filter factor, and the resultants are added to together. In the end, the summed value is scaled and clipped to the desired bit depth. It is obvious that lots of parallel resources are needed for efficient filtering.

2.3 Wireless communication

Wireless telecommunication protocols are needed in devices which have high energy efficiency requirements, like mobile phones and IoT sensor nodes. The latest wireless
standards have strict real-time requirements, for example, one-millisecond latency in the case of 5G wireless protocol [60]. Consequently, an implementation of the protocol has to be computationally efficient, predictable and parallel. The protocols demand fast dynamic reconfigurations, like requirement switching between different modulation types within a microsecond time window when decoding samples [61].

Massive MIMO technology [10] exploits a vast number of antennas, and each of them has a dedicated digital baseband processing chain. This directly increases the complexity of the baseband processing and introduces challenges for the management of parallelism [62]. In some applications, a flexible solution is required to provide support for the different standards, algorithms, and antenna configurations and to adjust to the evolution of standards [62].

Paper IV and Paper V present a digital predistortion filter (DPD) which is used as an example of the dynamicity of existing wireless communication algorithms. DPD aims to suppress the most harmful spurious emissions at the mobile transmitter power amplifier output [63]. DPD comprises a set of complex-valued FIR-filters, which are runtime re-configurable. Depending on the dynamic configuration, the operation of FIR-filters can be changed. Fig. 9 illustrates the use principle of DPD in a wireless transmitter, and shows a dataflow realization of the DPD which is based on a parallel Hammerstein structure.
2.4 Digital image processing

Many DSP applications involve several pipeline processing steps for an input signal before the final output is produced. For example, a color image processing pipeline for signal captured from an imaging sensor of digital cameras can consist of several pipeline stages like preprocessing, white balancing, demosaicking, color transforms, post-processing and image compression [64]. Computations have to be performed fast with low power consumption to keep shot-to-shot delay short. Moreover, reconfigurability is needed for the ever-increasing number of features and options (different filters, color transformations, post-processing, etc.) which are demanded by users. As a result, high-end mobile phones and digital cameras can have multiple specialized processors for image processing.

Paper V presents a stereo depth estimation (SDE) application from the field of computer vision. A depth map can be derived from stereo camera images, and it can be utilized for a variety of applications. For example, a depth map can be used for applying the bokeh effect (out-of-focus blur) for a captured image [65]. The SDE implementation in Paper V is based on block matching where the sum of absolute differences (SAD) is calculated over a block of pixels and compared to each other to derive stereo disparity between images. SAD has been used in motion estimation for HEVC video coding [66]. Fig. 10 illustrates data flow of the SDE application implemented in Paper V.

Disparity image \( I_d \) can be derived by starting at with calculating the SAD between images \( I_L \) and \( I_R \) as follows:

\[
SAD(x, y, d) = \sum_{j=0}^{B-1} \sum_{i=0}^{B-1} |I_L(x + i, y + j) - I_R(x + i - d, y + j)|,
\]

(1)
where \( I(x, y) \) is the pixel value of the image at point \((x, y)\), \( d \) is a disparity offset between disparity range \([0, D_{\text{max}}]\) and \( B \) is the size of the block. In this thesis, input images are assumed to be rectified, and SAD is only needed to be performed for horizontal direction. The best matching disparity value can then be selected for disparity image \( I_D \) at point \((x, y)\) as follows

\[
I_D(x, y) = \arg\min_d SAD(x, y, d).
\] (2)

The block size \( B \) and a maximum range of disparity \( D_{\text{max}} \) search have a radical influence on the complexity of the SDE. Even rather small \( B \) and \( D_{\text{max}} \) SDE can be considered to be complex since the number of absolute difference calculations will be \( W_H B^2 \).

Typical preprocessing steps for RGB images before the SAD are greyscaling and Sobel filtering. The luminance image \( I_Y \) can be derived using a linear transform

\[
I_Y = \begin{bmatrix} I_R & I_G & I_B \end{bmatrix} \begin{bmatrix} 0.2126 \\ 0.7152 \\ 0.0727 \end{bmatrix}.
\] (3)

The Sobel filtering for image \( I \) is implemented as a 2-dimensional convolution using a vertical Sobel kernel as follows

\[
G_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I.
\] (4)

2.5 Conclusion

This chapter presented DSP applications from different domains, and their characteristics were outlined by keeping the main focus on video coding algorithms. It is common with DSP applications that they can be naturally described as a set of interdependent but concurrent processing steps that are applied on data. Consequently, data accesses are mainly local and the processing is very data intensive. In many cases, it is possible to distribute the processing to multiple processing pipelines to increase the parallelism.

Typically, these applications run on embedded small form factor mobile devices, wearable devices or sensor nodes which set the tight requirement on power consumption and the silicon area. Moreover, the ever-increasing complexity of applications requires
more performance from the computation platform. On the other hand, flexibility of the solutions is needed to respond to market challenges, like a short time-to-market window, increased chip manufacturing cost, and the broad variety of functionality to support. Many video coding and wireless communication systems have to be backward compatible with older standards or support other standards, and this also speaks in favour of programmable solutions.

Many new applications relies on parallel algorithms to allow implementations that meet power and performance requirements. This has set new challenges, especially for software design methods, that have a long tradition of describing algorithms using sequential paradigms. Hence, cost-efficient parallel software design methodologies are needed to tackle the design challenges of parallel systems.
3 Describing applications using dataflow graphs

3.1 Dataflow programming

In many real-world applications, there is a need to handle causal interaction between two actors, where the direction of interaction is important. One obvious example of this kind of interaction is data transfer, like transporting goods from a shopkeeper to a customer, or transferring data bits inside a processor from a register to an arithmetic logic unit. Such situations can be modelled using **directed graphs**, where interaction is described using a directed edge. In real-world situations, the capacity of the edge is limited - for example, the storage size of a transport truck or a bus width of the processor.

The directed graph $D$ is used in the dataflow programming paradigm to model applications. The dataflow application is modelled as a graph $D$, where data manipulations are described as a set of **vertices** $V$ (actors or nodes), and data paths as a set of **edges** $E$ (or arcs). Data packets, called **tokens** flow through edges which can be modelled as **unbounded first-in first-out** (FIFO) channels. In Fig. 11, a simple application is presented using a graphical and textual dataflow format.

The dataflow programming paradigm started to be formed in the 1960s. Sutherland described arithmetic computations in graphical format in his Ph.D. thesis "The On-line

![Fig. 11. Simple program presented by using the graphical dataflow graph $D$ and textual presentation.](image-url)
graphical specification of computer procedures” in 1966 [67]. Later on in the 1970s, dataflow was the subject of many research efforts, and the fundamental principles of the dataflow model were created. Dennis and Kosinki designed dataflow programming languages [68, 69] and also many dataflow processor architectures were presented [70, 71, 72].

The main motivation for developing the dataflow execution model was to avoid two parallelization bottlenecks which are present in the control flow based von Neumann execution model: the global shared memory and the program counter [73]. Backus [74] called the global memory problem the von Neumann bottleneck and he described the nature of the problem:

"Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself but where to find it."

In the von Neumann model, an instruction is scheduled when a program counter reaches it, whereas in the case of the dataflow model scheduling is based on data availability. In most dataflow systems, the basic firing rule [73] is that an actor has data available in its incoming edges, which means that the actor is self-scheduling. In general, the actor has a set of firing rules that have to be satisfied so that the actor becomes enabled and it can be scheduled (fired). During the firing, the actor consumes tokens from its input edges and produces tokens into the output edges.

In the dataflow model, only a partial order of tasks is specified that allows parallel and pipelined computations at the desired level of granularity. The advantage of this is demonstrated in Fig. 11 by a simple program that could be executed in two cycles using a dataflow execution model, whereas three cycles would be needed for the von Neumann execution model.

In fact, in the 1980s the dataflow concept seemed so promising that there were expectations that dataflow languages and architectures were going to supersede the von Neumann based languages and architectures, but these expectations were not realized [75]. According to Johnston et al. [75] reason for that was the hardware aspect of the dataflow. Early dataflow hardware architectures operated at instruction-level granularity.
In other words, each instruction execution was treated as a concurrent action, but before a new instruction could be issued to the deep dataflow pipeline, a predecessor instruction had to be executed [73]. Although this enables fine-grain parallelism, it leads to poor performance in the case of sequential algorithms.

Johnston et al. [75] described fine-grained dataflow architectures as machines that execute each instruction in its own thread, and von Neumann architectures as machines that execute a whole program in one thread. That is, these two architecture types are extremes on the spectrum of possible computer architecture. Based on this, it is intuitive to conclude that general purpose computer architectures would be a hybrid between these two architectures [76]. Obviously, this rise to questions about the suitable granularity of the threads, and led to many different studies of hybrid architectures starting in the 1980s [76, 73, 77].

The emphasis of dataflow research moved from fine-grain parallelism to the more coarse grain parallelism in the 1990s. One development direction with the most potential was coarse-grain dataflow. In coarse-grain DFGs, graphs contain macroactors [77] that are sections of sequential code (e.g. functions), and they can be programmed using an imperative language, such as C. Macroactors are compiled into sequential von Neumann processes, and scheduled and executed using dataflow principles in a multithreaded environment [75].

From the perspective of software engineering, coarse-grain dataflow offers many advantages which were noted and exploited in the field of signal processing already in the 1980s. In their paper, Lee et al. [78] propose coarse-grain dataflow-based techniques to program digital signal processors, and stated that the coarse-grain DFGs simplifies the programming task by enhancing the modularity of the code, and permitting algorithms to be described more naturally. It is important to notice that coarse-grained dataflow programming environments for digital signal processing should not be confused to dataflow hardware architectures [79]. From now on the thesis concentrates to dataflow only in the context of describing signal processing software.

In the 2000s, the shift from the single core era to the multi-core processor, and further towards era of the heterogeneous multiprocessor system on chip systems brought parallel computing almost everywhere. In this context, the dataflow programming paradigm has a considerable advantage over the imperative program paradigms by naturally expressing concurrency of an application at different levels of granularities. That is, edges directly show data dependencies between vertices, and therefore concurrent parts of the application can be easily detected and distributed over multiple processing
elements. As a consequence, the dataflow programming paradigm has been proposed as next-generation programming solution for future computing platforms that have a high degree of heterogeneity and parallelism [23, 80, 81, 61, 82].

### 3.1.1 Dataflow parallelism

Dataflow descriptions can explicitly present three different forms of parallelism:

1. **Pipeline parallelism**, which refers to the situation where actors have chained dependencies on each other, but they can be executed concurrently on a different set of data. If the complexity of the actors is similar, pipelining can yield a significant performance gain. In Fig. 12 a), c) and e) represent pipeline parallelism.

2. **Task parallelism** arises when two or more actors have no dependencies on each other. For example in Fig. 12 b), there are actors A and B which are independent of each other.

3. **Data parallelism** arises when multiple instances of the same actor, without any dependencies on each other, process data from different channels. For example, in Fig. 12 d) there are two instances of actor A, which are independent of each other and consume data from different channels. On the other hand, the schedule of Fig. 12 d), can also be derived from a graph which contains only one actor A, if A receives enough data for several firings. Data parallelism yields efficient load balancing if the behaviour of the actor is data independent.
3.1.2 Synchronization

When low-level parallel programming methods are used, the programmer has to ensure proper synchronization, which can be a very challenging task even for an experienced programmer. In practice, synchronization is needed in the creation of parallel tasks (fork and join), the coordination of parallel producer-consumer tasks and the prevention of simultaneous accesses for serially usable resources (mutual exclusion). In the case of a dataflow model, synchronization and parallelism are decoupled from the application, and dataflow runtime implementation ensures proper synchronization. This eases the development of parallel applications and forestalls programming errors related to synchronizations issues which can be hard to detect.

3.1.3 Maintainability

Since dataflow actors have strictly defined ports (input and output channels), the dataflow graph decomposes the program into encapsulated processing blocks which make the program description modular, and therefore improve the maintainability. The modular description enables several important properties like hierarchical representation, reusability, and reconfigurability.

The hierarchical representation means that an actor in a DFG can be described as the composition of the other actors that forms another DFG. The application is usually designed in a top-down manner, and the designer could define top-level actors first, and later proceed to refine the actors by using sub-graphs. Moreover, the hierarchical representation allows information hiding and enabling, showing only essential information which helps to understand the application behaviour. Hierarchical dataflow models and their advantages are discussed more in [83, 84].

Reusability means that when an actor is designed once it can be reused in the same or in other DFGs. Reusability is an important property in the context of video coding, since different video standards share similar functionalities.

In the context of dataflow, reconfigurability often refers to the possibility for dynamic reconfiguration of actors at runtime [85]. For example, the actor can have several operation modes, and it is possible to switch between them on the fly. In context of video coding, standards include several profiles and encoding options that have an impact on the decoder implementation. By switching operation modes, it would be possible to reconfigure the decoder, for example, to support a particular encoding option.
3.1.4 Modelling granularity

Dataflow applications can be modelled on different levels of granularity. At one extreme, a fine-grain dataflow actor can describe just a single operation such as some logical or arithmetic operation. Meanwhile, coarse-grained actors can describe a block of operations, whole functions or even larger unities. In other words, granularity defines how detailed the model is. Actually, granularity has an influence on the degree of explicit parallelism of the dataflow description.

From the perspective of the designer, granularity should be selected to be natural for the target application. For example, if an image processing pipeline that consists of different processing stages is considered, it is natural to divide the pipeline so that each actor performs one processing stage on one image at a time. As discussed above, hierarchical modelling can be used for information hiding; in that case, top-level models are refined by fine-grained actors and the connections between them.

Fine granularity improves the re-usability of the actors and helps to find more optimal schedules and minimize memory space [86]. It might enable a broader set of target devices since hardware synthesis for FPGA or ASIC also remain reasonable implementation options [87]. Moreover, fine-granularity raises an explicit parallelism of DFG. Unfortunately, a fine-grain dataflow application does not always perform as desired due to the limits of optimizations that dataflow code generation tools can provide [87]. In the case of programmable processors, too fine-grained descriptions can cause FIFO communication and scheduling overhead, which slows down the execution.

Problems related to overtly fine-grained descriptions can be partially addressed by using actor merging [87]. In actor merging, feasible subgraphs are formed and replaced by coarser composite actors [88] that implements the same behaviour. Boutellier et al. have briefly reviewed different actor merging methodologies in [87]. Janneck has a more theoretical and general perspective in his paper [88].

Contribution: CTU-based dataflow description in-loop filtering

In Paper I and Paper V, the inloop filter of the HEVC decoder is described in a dataflow manner, and it can be considered to be a rather coarse-grained description. The filter pipeline is partitioned into three actors, and each actor processes one coding tree unit (CTU) at the time. The CTU is the basic coding unit of the HEVC, and therefore it is the natural choice for the size of a token. Since the HEVC de-blocking filter can be
Fig. 13. The HEVC in-loop filters described as dataflow actors, operating at CTU-level. Revised from Paper I © IEEE 2015.
applied in two stages, two actors (DBF-VE and DBF-HE) are used to describe it in order to achieve a better load-balance across the pipeline. SAO is described using one actor. Fig. 13 illustrates the partitioning. If the video is encoded using multiple tiles, the actor pipeline can be replicated to increase data-level parallelism.

CTU-based filtering has an advantage over frame-based filtering, since it reduces the memory requirements significantly. However, the CTU-based approach complicates implementation, since all required data is not available to filter the whole CTU. Consequently, filtering of some parts of the CTU have to be delayed, and therefore some areas of the CTU have to be stored to memory. Finally, this leads a situation whereby there are parts from four different CTUs to be filtered in the SAO stage. The bottom part of Fig. 13 illustrates the situation. A more detailed description of the filtering flow is presented in Paper I.

Contribution: row-based dataflow description of stereo depth estimation

The stereo depth estimation application presented earlier in Section 2.4 is implemented using the dataflow actors in Paper V, and it is illustrated in Fig. 14. The implementation consists of five different types of actors that are Imageread, Grayscale, Sobel, SAD and Imagewrite. Dedicated actors for reading and reprocessing left and right images before SAD are instantiated. Actors process data row by row. However, Sobel and SAD actors require samples from multiple consecutive rows, and they have to fill their line buffers before they can start the processing. The Imageread actor can divide the image into multiple blocks, and fork multiple row streams to enable data parallel actor pipelines, which are finally joined by the Imagewrite actor.
Most of the existing software implementations of the SDE are frame by frame like [89], for example. However, pipeline parallel implementation of frame-based SDE requires almost $H$ times more memory than row-based implementation, if $H$ is image height.

### 3.2 Dataflow model of computations

A model of computations (MoC) is an abstract model which defines how computations can progress. Over the years, numerous different MoCs have been studied for different use cases. Since researchers developed the fundamental principles of dataflow in the early 1970s, several formal dataflow MoCs have been proposed. The use of dataflow MoCs has been common in DSP since they facilitate analysis of application and optimization using semantic restrictions [81].

Dataflow MoCs can be divided into two categories according to the actor independence or dependency of the data. In the case of a static MoC, the number of tokens consumed (consumption rate) and the number of tokens produced (production rate) by the actor is fixed, whereas in the case of a dynamic MoC token production and consumption rates can vary depending on the data. That is, production and consumption rates define the actor’s *dataflow behaviour*.

Stuijk [90] and Yviquel [91] presented a taxonomy to classify different dataflow MoCs by using different characteristics which are *expressiveness*, *practicality*, *efficiency*, and *analysability*.

Expression power describes the size of an application set that can be modelled by the MoC in question, but it does not consider, *practicality*, the ease to describing an application. The expression power of the static MoCs is more limited than dynamic MoCs, since the set of possible applications that can be modelled is smaller when compared to dynamic MoCs.

The main advantage of static dataflow MoCs is that their data independent dataflow behaviour increases analysability and enables compile-time scheduling decisions. When a dataflow graph is analysed, answers for the following *key questions* [92] is sought:

- Is there a sequence of actor firings that returns the graph to its original state (*cyclic schedule*)? The graph is said to be *inconsistent* if it does not have a cyclic schedule due to its token rates.
- Is there a *bounded cyclic schedule* for the graph?
Fig. 15. Simple synchronous dataflow graph, where numbers indicate consumption and production rates of actors and the black dot indicates delay.

- Can execution of the graph cause Deadlock (a situation where there are no fireable actors in the graph)?
- Is there a schedule for the graph where bounded memory can be used? Dataflow graphs are said to be well-behaved if there exists an infinite, fair firing sequence of the actors that requires only bounded memory on every edge [93].

Answers to the questions can enable (but not guarantee) the use of optimizations that increase the theoretical efficiency of an application (e.g., memory requirements, execution time). The questions are rather straightforward in the case of restrictive static MoCs (by solving balance equations [94]), but when dynamic formalisms are used, the questions can become more complicated or even undecidable. Often dataflow MoCs offer differently balanced trade-offs between expression power and analysability.

**Synchronous dataflow**

Synchronous dataflow (SDF) was proposed by Lee and Messerschmitt in 1987 for describing DSP applications for concurrent implementation on parallel hardware [94]. SDF is the most extensively studied dataflow MoC in the DSP context. In SDF graphs each actor consumes and produces a fixed number of tokens concerning their input and output edges. An SDF actor can have one or several firing rules, but all of them have consumed the same number of tokens from an input. Edges can contain a predefined number of initial tokens that are called delay tokens. The aforementioned static properties make SDF an attractive choice for embedded platform application design since it allows efficient implementation and validation of bounded memory and deadlock-free operation.

An homogeneous SDF graph is a special case of SDF, where all consumption and production rates are fixed to one. In the case of symmetric-rate dataflow, the SDF graph is restricted so that the production rate and consumption rate of FIFO are the same.
A simple SDF graph is presented in Fig 15. The numbers in the graph indicate the fixed consumption rates and production rates of the tokens. The black dot indicates delay tokens.

The in-loop filter application presented in Paper I and Paper V, and the stereo depth estimation application presented in Paper V can be considered to follow the SDF MoC.

**Parameterized dataflow**

*Parameterized dataflow* (PDF) is a meta-modelling technique which was proposed by Bhattacharya and Bhattacharyya [83] to increase the expressive power of dataflow models using runtime configurable parameters that allow dynamic control of both the functionality and dataflow behaviour of the actor. PDF allows descriptions of *quasi-static* dataflow models that can contain compile time schedulable static regions and dynamic regions scheduled at runtime. Therefore, quasi-static models offer trade-off between expressive power and analyzability. PDF can be used to generalize other exiting dataflow models to extend their expressive power, and therefore many parametric dataflow MoCs have been proposed such as *parameterized synchronous dataflow* (PSDF) [83], *schedulable parametric dataflow* (SPDF) and *transaction parameterized dataflow* (TPDF) [95] for example.

**Boolean dataflow**

*Boolean dataflow* (BDF) is a dynamic dataflow MoC [92]. Boolean dataflow extends the SDF model by exploiting dynamic *control actors* (SWITCH and SELECT) that can be used to implement if-then-else programming structures to control the flow of data. These two control actors have a Boolean (true, false) control input that chooses the active port from its two input/output ports. This property facilitates the scheduling of actors since only Boolean control edges need to be checked.

Fig. 16 shows a simple example where the conf actor produces tokens to the SWITCH and SELECT actors, and therefore controls whether FILTER1 or FILTER2 is used. Since the successor actors of the switch can have consumption or production rates that vary from each other, determining if the graph is *strongly consistent* [92] is more challenging than in the case of SDF.
Kahn process networks

Gilles Kahn described a simple language for parallel programming in his paper in 1974 [96]. Later on, his work has been called the Khan process network (KPN). The motivation of the work was to make system design languages more formal.

KPNs consist of concurrent processes that communicate through unidirectional FIFO channels, which have an unbounded capacity. The Kahn processes are stateful sequential programs that can write to outgoing channels and read tokens from incoming channels so that a single token can be written and read exactly once. Reading tokens from incoming channels are blocking whereas writing to an outgoing channel is unblocking. Blocking read means that the process has to wait as long as sufficient tokens appear in the channel. Writes can be unblocking because there is always free space in the outgoing channel due to the unbounded channel model.

Although the Kahn processes are concurrent and blocking reads can stall execution, KPNs can be executed in sequential environments. For this, some mechanics are needed to save a context of the process. Usually, this can be implemented using context switching. However, the overhead of context switching can be significant, and not all hardware architectures offer support for it. Paper V discusses the problem and shows how the use of context switching can be avoided.
KPN MoC ensures that processes are deterministic in the sense that the same history of input sequences of the process is always translated to the same history of output sequences. In general, the model that guarantees deterministic behaviour is a desired property, but the ability to describe nondeterministic behaviour is needed for some applications and KPNs are not suitable for that task.

**Dataflow process networks**

In 1995 Lee and Parks proposed dataflow process networks (DPN) [79]. In their paper Lee and Parks noted that nondeterminism could be added to the Kahn process networks by allowing any of the following methods:

- The process can test the emptiness of its inputs.
- The process can be internally non-determinate (e.g., allow access to random generators).
- Multiple processes can write to the same channel.
- Multiple processes can read from the same channel.
- Processes can share variables.

Dataflow processes allow the first method, test its inputs for emptiness, to make it possible for a programmer to express nondeterminism. A more formal presentation of DPN is given, and it follows the interpretation of [87].

DPN application can be described in the directed graph \( D = (A, C) \), where \( A \) is a set of actors (dataflow processes) and \( C \) is a set of FIFO interconnection channels. An actor \( a \in A \) can have a set of input ports \( P_{a}^{-} \) and output ports \( P_{a}^{+} \), that are connected a FIFO channel, \( c_k \in C \), where \( k \) is the index of the FIFO channel. In addition, the actor \( a \) has one or more firing functions \( f_i \in F_a \) and for each \( f_i \) there is a firing rule \( r_i \in R_a \). Firing rule \( r_i \in R_a \) defines the acceptable sequence of patterns, that enables firing of \( f_i \in F_a \), for each port \( P_{a}^{-} \). The firing rule \( r_i \in R_a \) also defines how many tokens are produced to output ports \( P_{a}^{+} \). Firing rule \( r_i \in R_a \) can be data dependent in the sense that tokens in input port \( P_{a}^{-} \) need to have specific value to satisfy the firing rule \( r_i \).

An actor \( a \) can be immediately fired, when one or more of its firing rules from \( R_a \) is satisfied. In a single firing, the satisfied firing rule \( r_i \in R_a \) is chosen and the corresponding firing function from \( f_i \in F_a \) is executed. Firing functions are also called actions that are indivisible atomic computation operations, and therefore also actor firing is the atomic operation. Actions can share the state of actor, which allows a
convenient way to describe feedback loops. Fig. 17 presents the encapsulation of the DPN actor $A$ with its state and actions $F_a$ with the firing rules $R_a$ that are associated to them.

DPN MoC provides Turing complete expression power and flexibility, but it is also hard to analyse, and all scheduling decisions are performed in runtime generally. However, not all DPN actors necessarily have dynamic behaviour. By using actor classification methods [97], it is possible to detect actors that can be modelled using a more restrictive MoC, which could allow efficient optimizations.

In Paper IV and Paper V, several DPN applications, including video processing and telecommunication domains, are examined. Moreover, Paper V propose the co-design framework which has been founded on DPN MoC.

3.3 Parallel programming frameworks

Parallelization is an effective way to improve performance and energy efficiency, but it also introduces overheads like synchronization, load imbalance, communication and thread management that cannot be avoided. Addressing these issues usually increases software design time.

Nowadays, there are lots of different programming frameworks for parallel computing [98]. They vary in terms of abstraction level, application domain or target platform. Most of the parallel programming frameworks are rather low-level in the sense that the programmer has to have in-depth knowledge of the underlying hardware and programming conventions for a particular programming environment to get high-quality results. Moreover, the programmer has to take care of proper synchronization, which is error-prone and can be tricky even for advanced programmers. Using a low-level
abstraction hinders software code portability, and as a consequence increases design cost and hampers software code re-use.

Low-level parallel APIs can be exploited as an intermediate layer of the refining process from high-level dataflow language down to machine language. Next, a non-exhaustive list of parallel programming frameworks based on dataflow MoCs is presented, and after that commonly used lower level parallel programming frameworks are presented.

### 3.3.1 Dataflow modelling frameworks

Several different dataflow modelling frameworks have been developed over the years. Some of the frameworks are intended to be used only for system modelling and simulating purposes, but an increasing number of frameworks are targeting the real implementation of the system.

**PREESM**

The Parallel and Real-time Embedded Executives Scheduling Method is a rapid prototyping and code generation tool that is used to simulate signal processing applications and code generation for multi-core DSPs, heterogeneous MPSoCs and many-core architectures [99]. PREESM is based on the Parameterized and Interfaced SDF (PiSDF) MoC [100], which is closely related to the PSDF MoC presented earlier in section 3.2. In PREESM application prototyping workflow, user inputs are algorithm presented in the PiSDF model, a system level architecture model and scenario that includes necessary information to link an algorithm and an architecture model. PREESM applies several transformations for the algorithm and architecture models and then it generates a static schedule and optimizes FIFO memory usage with a memory exclusion graph [99]. After that, PREESM generate static C code for each core that can be compiled to binary using the DSP vendor tools. The PREESM simulation produces a Gantt chart of the parallel code execution, a chart of expected execution speedup depending on the number of resources, and an evaluation of memory usage during the execution.

**SPiDER** [101] is a PiSDF-based real-time operating system for multicore heterogeneous processors. In SPiDER, the master GPP executes the global runtime that makes multicore scheduling decisions. Slave processing elements run a low memory footprint
local runtime that can process actors and send timing and configuration parameters back to the master.

The MPSoC application programming studio

The MPSoC Application Programming Studio [23] (MAPS) is a KPN based framework for mapping multiple concurrent applications onto heterogeneous MPSoCs. It provides algorithms for semi-automatic program partition into processing elements. Additionally, MAPS includes support for multiple concurrent applications and possible interactions between them.

Ptolemy

Ptolemy [102], a simulation and prototyping framework for heterogeneous systems, the pioneering software development framework for model-based design, simulation, and analysis. In addition to dynamic dataflow, Ptolemy also supports a variety of non-dataflow MoCs, and even allows the mixing of different MoCs.

PeaCE

The PeaCE is an open source hardware/software co-design environment that is built on top of the Ptolemy [103]. In the PeaCE system, behaviour is specified using an extended SDF model for computation and an extended FSM model for task controlling and interactions. PeaCE can automatically generate C and VHDL code from the model based input specifications for software and hardware implementations.

DAL

The distributed application layer (DAL) is a scenario-based design flow for mapping streaming applications onto heterogeneous many-core systems [104]. In DAL, applications are described to comply with KPN MoC, and the user can also describe a finite state machine (FSM) to coordinate the execution of applications. The states of the FSM represent different scenarios, and state transitions can be triggered by either a DAL runtime system or behavioural events of the currently running application. The scenario represents currently running or paused applications. In DAL, actor mapping to target
architecture follows a hybrid method where mapping optimizations are performed in both design-time and runtime by an evolutionary algorithm.

KPN networks, processing architecture, the scenario FSM and mappings are described using XML based notation. DAL actors are described using a C language. DAL also offers OpenCL backend for GPU devices, but it supports only SDF graphs [105].

**Orcc**

The open RVC-CAL compiler (Orcc) [106] is an Eclipse-based open-source tool-chain dedicated to dataflow programming. In Orcc workflow, dataflow programs are described using the RVC-CAL language, which is a standardized dataflow language based on DPN MoC and CAL actor language [107]. The Orcc tool-chain includes software and hardware backends that can generate C/C++ for programmable multicore processors and synthesizable register transfer level (RTL) code for FPGA, and ASIC design flows, respectively. Therefore dataflow applications developed using Orcc workflow can be ported for over a large range of different platforms.

Orcc is built on top of the Eclipse, and it can offer a modern integrated development environment for dataflow application development. Orcc provides a graphical dataflow
graph editor and actor source code editor that includes syntax highlighting, code completion and code validation. In Fig. 18, a screen capture of Orcc graphical graph editing tool is presented.

**PRUNE**

PRUNE [81] is an open source programming environment that is based on dynamic dataflow MoC which has been developed especially for heterogeneous platforms that include graphical processing units (GPUs). PRUNE was the first framework that enabled execution of dynamic data rate dataflow applications on OpenCL devices [108]. PRUNE MoC is based on symmetric-rate dataflow and also resembles Boolean dataflow with its Boolean controllable token rates that allow dynamic dataflow behaviour.

An application described using PRUNE MoC can have three different types of actors. The **static processing actor** is the actor that has only static ports which have fixed token rates. The **dynamic processing actor** is the actor that has at least one dynamic port, and any number of static ports. The token rate of a dynamic port is controlled by a dedicated Boolean control port that selects the token rate of the dynamic port to be either zero or some other predefined positive token rate. A **configuration actor** has at least one static regular output port that is connected to a control port of a dynamic processing actor.

The PRUNE framework includes the PRUNE compiler and analyser that takes three inputs: an application graph, a platform graph, and application to platform mapping information. The analyser checks that the application graph follows the design rules of PRUNE MoC. The compiler generates the main program file in C format, which can be compiled for target machine binary using a C compiler. The user has to provide the source code of the actors in C or OpenCL format, depending on whether the actor is mapped on the CPU or the GPU target.

**TensorFlow**

TensorFlow [109] by Google, is a dataflow-based framework targeting machine learning. However, it is suitable also for other application domains too. TensorFlow has its dedicated model of computation, which allows operations (actors) to have a mutable state, for example. The goal has been to unify computation and state management in a single programming model, and to enable different parallelization schemes. In TensorFlow, data which is flowing through the edges is called tensors. Tensors are
n-dimensional arrays, which contain primitive types of elements like integers, floats or strings. TensorFlow supports dynamic control flow by using classic switch and select control operations [109]. The dataflow model ensures that TensorFlow graphs can be mapped into computer clusters, and further down to multicore CPUs, GPUs and tailored ASICs [110] called Tensor Processing Units.

### 3.3.2 Low-level parallel programming APIs

**POSIX Threads**

The threaded execution model is a well-known abstraction of concurrent execution. In the model, one thread can be seen as an independent block of instructions, which can be scheduled concurrently with other possible threads. Threads are widely used to implement parallel and asynchronous programs for shared memory multicore environments. One popular implementation of the threads is POSIX Threads (pthreads) which is the standardized (IEEE POSIX 1003.1c) application programming interface (API). The API provides procedures for thread management, synchronization, and communication.

POSIX Threads can be considered as a low-level abstraction of parallel programming since the designer has to manage threads explicitly (create, destroy), distribute workload and take care of proper synchronization. The designer has to protect critical sections of data to avoid deadlocks and data races using mutual exclusion or semaphores. The designer needs a profound understanding of low-level concepts of parallel programming.

Many dataflow tools generate lower level code which utilizes the pthreads to spread the execution of the actors for the multiple cores [106, 81, 104, 105]. Also, in the Paper IV, the authors proposed a dataflow runtime that is based on pthreads.

**Message passing interface**

The message passing interface (MPI) is a specification of the message-passing library interface for addressing the message passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations in each process [111]. Initially, MPI was designed for programming of distributed systems, like server environments where multiple processors with their own memory address space are connected to the same network.
Currently, MPI standard has support for shared memory multi-core processors, but the programming model follows a distributed memory model [111]. MPI can be considered to be a low-level parallel programming model since the programmer is responsible for identifying parallelism explicitly and implement it properly using MPI constructs. MAPS MPSoC dataflow frameworks exploit MPI constructs for generating code for Network on Chip (NoC) architectures [112].

**Intel Cilk Plus**

Cilk [113] was initially developed at the Massachusetts Institute of Technology (MIT), and later commercialized and, finally, acquired by Intel, and being currently known as Intel Cilk Plus. Intel Cilk Plus provides simple constructs that can be used to express potential parallelism in a program. Cilk can be considered to be a C/C++ language extension for task and data parallelism, and it requires compiler support and specific runtime implementation to execute Cilk programs.

The main focus of the Cilk constructs is to exploit task level parallelism and data level parallelism. Parallel loop construction is one of the examples of the structures that utilize task level parallelism. For data-level parallelism, Cilk provides the special notation for array operations and special compiler directives that permit the compiler to perform loop vectorization.

Although Cilk can be seen as higher level abstraction of parallel programming than pthreads for example, still, the programmer has to expose and identify possible parallel portions of the program and take care of synchronization. However, Cilk takes care of the runtime scheduling of the workloads. The runtime scheduler of Cilk is based on a dynamic scheduling policy called *work-stealing* [113]. In work-stealing scheduling, an idle processor can steal work items from other processors work-queues to balance work-load dynamically.

**Open Multi-Processing**

Open Multi-Processing (OpenMP) [114] is an API for multi-core shared memory architectures, and it is based on compiler directives which are supported by many C, C++, and Fortran compilers. As in Cilk, OpenMP provides constructs to express data-level parallelism and task-level parallelism. Since loops are the common source parallelism, the parallel loop directives are an integral part of OpenMP API.
The OpenMP runtime system handles thread management to simplify the job of the programmer. For the scheduling and distributing of the workloads to computing resources, OpenMP uses the fork-join model. In the fork-join model, a master thread forks off a defined number of worker threads that are assigned to different processors by the system scheduler. When the work items that are assigned to the threads are computed, a master thread joins the worker threads.

OpenMP offers support for heterogeneous systems by introducing a set of device constructs, which are also referred to as the Accelerator Model [115]. The accelerator model is a host-centric model, which consists of a host device and multiple target devices (accelerators) of the same type [116]. Using target constructs, the programmer can specify regions of the code that have to be offloaded to the accelerator device.

The OpenMP API is a widely used industry standard, and it has support for many embedded platforms, including DSPs and MPSoCs. Consequently, OpenMP has been exploited to implement dataflow model based runtimes for embedded devices [117, 118].

Open computing language

Open Computing Language (OpenCL) [119] is an open standard, developed by the Khronos Group for general-purpose parallel programming targeting heterogeneous processing platforms. OpenCL is mainly used in general purpose computing for exploiting the computing power of GPU devices. However, OpenCL is not only targeted at a GPP with GPU accelerator combination, but also DSPs, FPGAs and other embedded accelerator devices that implement OpenCL support.

The OpenCL programming model is host-centric, in which a host device offloads compute kernels to accelerator devices. Kernels are written using OpenCL C kernel language, which is based on the C99 standard with language extensions that facilitate management of parallelism.

From the perspective of the programmer, OpenCL provides a rather low-level of abstraction for parallel programming, and because of that, the designer still has to manage concerns related to data transfer between devices, synchronization, and the scheduling of tasks. Several dataflow-based solutions have been proposed to address these issues by raising the level of abstraction. [105, 120, 121, 122] have presented different OpenCL based methodologies to execute SDF graphs on heterogeneous platforms. In [81, 108], Boutellier et al. relaxed the constraints induced by SDF and
introduced the PRUNE framework, which allows the execution of dynamic dataflow applications on heterogeneous platforms.

Compute unified device architecture

Compute Unified Device Architecture (CUDA) is the proprietary parallel computing framework for GPGPU computing developed by NVIDIA. CUDA is very similar to OpenCL, but CUDA supports only NVIDIA’s own GPU devices. As in OpenCL, CUDA provides abstractions for threading, memory management, and synchronizations, which are exposed through a set of language extensions. Additionally, the CUDA programming interface includes a runtime library. The runtime library provides functions for memory allocation and deallocation from a target device and data transfers between host and target, for example.

CUDA is also exploited when generating lower level code from dataflow descriptions. Huynh et al. [123] proposed an automated GPU mapping framework for SDF based StreamIT applications. They have developed a backend for the StreamIT compiler, which generates NVIDIA GPU compiler compliant CUDA C code. There are several other dataflow programming model based products that exploit CUDA for offloading computing to GPU, such as Tensorflow [109], PTask [124], Xcelerit [121] and Alea reactive dataflow [125].

Halide

Halide [126] is a domain specific programming language for image processing targeting high-performance implementations. Halide is special in the sense that it decouples the algorithm from the schedule which refers to the memory placement and the order computation. In Halide, the idea is to optimize the schedule by exploring different design choices of tiling, fusion, re-computation vs. storage, vectorization, and parallelism without changing the algorithm description, which makes the algorithm description portable. It is possible to generate the schedules automatically [127] or manually by a user. Halide offers several backends for different platforms and can exploit SIMD (SSE, NEON) units, multiple cores, GPUs (Cuda or OpenCL code generation), and complex memory hierarchies.

In [128], PRUNE uses the description language of Halide to extract the internal data parallelism of actors and increase portability by avoiding writing separate actor
implementations for actors that are mapped into GPUs. The authors show that Halide can provide significant throughput improvement over the plain C, OpenCL or Cuda actors.

3.4 Conclusions

This chapter presented dataflow graphs and their benefits for describing signal processing applications. The main motivation to use dataflow descriptions is their ability to express the concurrency of an application. This is a very important characteristic because most of the computing platforms nowadays have a lot of parallel resources.

Different Model of Computations (MoCs) for dataflow graphs were presented in this chapter. MoCs can be divided into static models and dynamic models. The demand for dynamic MoCs are increasing in the future since the complexity of applications is growing. On the other hand, the use of static MoC can allow efficient offline optimizations applied for an application.

Lastly, different parallel programming frameworks were presented. Traditional parallel programming APIs use a low abstraction level which slows down software development, makes programming error-prone and hinders code portability for different computation platforms. Dataflow based programming environments can avoid these pitfalls by decoupling synchronization and parallelism from the application description. The popularity of dataflow programming frameworks can be predicted to increase in the future. For example, Google has already launched the dataflow based software development environment Tensorflow for machine learning.
4 Energy efficient computing platforms

This chapter presents the implementation options from different abstraction levels that can be used for designing energy-efficient embedded systems. At the end of the chapter, an energy efficient application specific processor architecture template is presented and evaluated in the context of video coding algorithms.

In embedded systems, power management is important for several reasons:

- Embedded mobile systems are battery operated and are usually constrained by physical size. Heat-dissipation becomes more difficult when the size of the device gets smaller. By reducing power consumption, heat production is reduced, and the battery size can be smaller. Heat dissipation problems increases chip temperature and can induce reliability issues and further increases the leakage power of device [129].

- Increased performance demands of embedded applications are making embedded architectures more complex, and desktop processor features like out-of-order executing [130], branch prediction [131], multiple cores [132], and hierarchical multi-level cache memories [133] are adapted to embedded platforms. These techniques increase the power requirements of the embedded system considerably [134].

- Complex applications can have various use case scenarios, that can have widely varying requirements. An embedded system has to be designed to meet the worst case requirements (performance, environmental and manufacturing process variations, etc.). Due to this, the system can be overprovisioned for some of the use case scenarios, which can be imply to undesirable power wastage if adequate power saving techniques are employed.

- Current high-end CMOS technologies can pack a considerable amount of transistors into a small area. However, the scaling of threshold and supply voltages of transistors has not kept pace with feature scaling [135]. The trend has led to a proportional increase of power-per-transistor, and further increased power density and thermal hotspots in chips [17]. Thus, energy consumption is a crucial factor that limits the performance of embedded processors and power saving techniques have become enablers for performance scaling.

Next, power saving techniques that are related to the original publications are presented. For further information about different register-level, architectural-level and
application-level power management techniques in embedded systems and the research challenges, [136, 17, 137, 138] are recommended.

\section{4.1 Parallel architectures for energy savings}

In parallel processing, different tasks are computed simultaneously by exploiting different hardware resources. The primary motivation to use parallel processing is to increase system throughput and energy efficiency.

When parallel computing resources are employed, Amdahl’s law \cite{139} is often used to calculate the theoretical speedup for a task at fixed workload as follows

\begin{equation}
\text{speedup} = \frac{1}{r_s + \frac{r_p}{n}},
\end{equation}

where \( r_s + r_p = 1 \), \( r_p \) is the ratio of the parallel portion of the task, \( r_s \) is the ratio of the sequential portion of the task and \( n \) is the number of parallel resources. The law says that if there is a sequential partition of the task, time reduction is limited even the resources are increased to infinity.

Gustafson’s law \cite{140} introduces another aspect by setting the execution time of a task to be fixed as follows

\begin{equation}
\text{speedup} = r_s + s_p r_p,
\end{equation}

where \( s_p \) denotes the speedup of the portion of the task that can be improved by adding more resources. Thus, Gustafson’s law suggests that in constant time, larger problems can be solved by adding more resources. Therefore, the parallelization limitations of the sequential portions of the task might be possible to address by increasing the problem size of the task. In another hand, the law says that scalable tasks are suitable for parallelization efforts. For example, in the context of video coding, it is possible to increase the problem size by increasing spatial, temporal or colour resolution, for example.

The average power consumption of the CMOS circuit \cite{141} can be defined as

\begin{equation}
P_{\text{avg}} = P_{\text{switching}} + P_{\text{short–circuit}} + P_{\text{leakage}} = \alpha C_L V_{dd}^2 f_{\text{clk}} + I_{sc} V_{dd} + I_{\text{leakage}} V_{dd}.
\end{equation}

The first and the second term of the equation is related to dynamic power, where \( \alpha \) is activity factor, \( C_L \) is the load capacitance, \( V_{dd} \) is the supply voltage and \( f_{\text{clk}} \) is the clock frequency. \( I_{sc} \) is a short circuit current. The last term is related to the static power consumption, where \( I_{\text{leakage}} \) is leakage current. The equation shows that dynamic power
is linearly dependent on the clock frequency and quadratically dependent on the supply voltage.

Let there be a constraint so that task $P$ has to be completed in time $t$, and there is machine $M_1$ clocked at frequency $f$ so that it can execute task $P$ in the given time $t$. Also there is an other machine $M_N$, which consist of $N$ parallel instances of $M_1$, and can also execute task $P$ in the given time $t$, but using a clock frequency of $\frac{f}{N}$. The machine $M_N$ has a total load capacitance $N \times C_L$ so lowering the frequency to $\frac{f}{N}$ gives the same dynamic power in case the machine $M_1$. However, the lengthened clock period means that there is more time to charge the capacitors and the supply voltage of the machine $M_N$ can be reduced. *Because dynamic power reduces quadratically and static power consumption reduces linearly when the supply voltage is reduced, the supply voltage has a significant impact on the total power consumption.*

For example, in Paper II, energy efficiency for the TTA based multiprocessor system is measured by using two different operating voltages and frequencies. The results showed that the TTA cores operating at 0.8 V and 530 MHz yield about 1.6 times better energy efficiency than the cores that are operating at 1.0 V and 1.2 GHz.

### 4.1.1 Types of parallelism

In section 3.1.1, different types of parallelism that can be extracted from dataflow graphs were presented. Next, the parallel hardware constructs that are typically found in contemporary computation platforms are reviewed. All the following types of parallelisms could be exposed from a description of a dataflow actor or dataflow network by a compiler for exploiting parallel hardware capabilities during actor execution.

**Bit parallelism**

A few decades ago, it was not unusual that computer architectures have 1-bit serial processors [142, 143] with 1-bit datapath and registers, but evolving chip manufacturing technologies have allowed an increase in the processor word size, and nowadays the typical word sizes of processors are 32-bit or 64-bit. The advantage achieved by the increasing processor word size is that the number of instructions can be reduced in those cases where a greater bit-width than the processor word size is needed. For example, to perform a 32-bit operation in a 16-bit processor, operands need two registers for high and low bits and two instructions for the operation. In Paper I and Paper III bit-level
parallelism is exploited by using 32-bit memory fetches to get four 8-bit pixels from memory simultaneously.

**Instruction parallelism**

Instruction-level parallelism (ILP) is a form of parallel processing whereby multiple instructions are executed simultaneously. There are several micro-architectural techniques which are designed to take advantage of ILP. Most of the current processors exploit *instruction pipelining* which enables partial overlapping execution of multiple instructions by dividing the instruction execution into multiple pipelined steps. In the best case, this enables the execution of one instruction per clock cycle (IPC).

The instructions that can be executed in parallel can be extracted on runtime with hardware support or on compile time with software support. Thereafter, parallel instructions can be dispatched to parallel execution units. Examples of architectures that support this kind of execution are *superscalar* [144], *Very long instruction word* (VLIW) [145] and *Explicitly parallel instruction computing* (EPIC) [146] processors, since they can execute more than one instruction per clock cycle.

Superscalar processors like Intel x86 architectures implement dynamic extraction of the ILP. However, the dynamic extraction of ILP requires complex hardware and therefore some processor architectures, like VLIW-architectures, exploit static extraction of the ILP [147]. ILP is an effective way to expose dataflow actor’s implicit parallelism, that is not inherent from the dataflow model. In this thesis, the focus is on Transport triggered architectures (TTA), which can heavily exploit ILP. For example, Paper III presents a TTA which can execute fifteen operations simultaneously.

**Data parallelism**

Data-level parallelism (DLP) refers to the possibility to perform the same operation for multiple data items simultaneously. Many CPU architectures provide hardware support for data parallelism through SIMD-instructions (single-instruction, multiple-data). SIMD-units usually perform the same operation for all elements of the vector simultaneously. Since the workload of processing is equal for different data elements, the load-balancing across homogeneous processing elements are optimal. Modern compilers have support for automatic vectorization and they can exploit the SIMD-extensions [148] [149] provided by CPUs. However automatic vectorization techniques have still
some pitfalls, and therefore designers might have to use explicit intrinsics to exploit SIMD extensions [150].

GPUs are an example of parallel processor architecture that is designed to exploit massive data parallelism. GPUs could have hundreds of homogeneous scalar processors that could execute thousands of threads concurrently. GPUs are based on an SIMT (single instruction, multiple threads) architecture model [151]. The SIMT model is more flexible and expressive than SIMD, since it allows multiple execution flow paths, random memory access, and portable intrinsics free code. However, as usual, flexibility comes with some loss of efficiency.

One of the main advantages of both the SIMD and SIMT models is that they increase the workload of a single instruction, and therefore reduces the instruction fetch overhead by sharing the same fetch and decode hardware between multiple execution units.

**Task parallelism**

In the case of task-level parallelism (TLP), a set of different tasks can be executed concurrently using the same or different data. In multicore environments, independent sequences of execution (threads or processes) can be distributed across several cores and execute simultaneously. While DLP favours homogeneous hardware, TLP justifies the use of heterogeneous hardware due to the dissimilarity between tasks.

Single-core can execute multiple threads simultaneously using the technique called simultaneous multithreading (SMT), where several independent threads can issue instructions to multiple functional units in a single cycle [152].

Execution of multiple threads can also be interleaved across CPU cycles, which usually requires hardware support to perform switching between threads efficiently. Multithreading can improve single-core utilization since if the thread in execution has to wait, another thread can be switched for execution.

Efficient implementation of context-switch support for some processor architectures can be difficult. In such cases, lightweight context-switch free methods [153] can be exploited (with some restrictions) to implement software multithreading for a single core, as is done in Paper V.

In Paper I, TLP is exploited by distributing different pipeline phases of inloop filtering for the pipelined multicore platform, where each processor executes a single thread. Paper V presents an automated flow to distribute different tasks for different
cores and moreover allows of mapping multiple lightweight threads (dataflow actors) for a single TTA-core.

4.2 Dynamic voltage and frequency scaling

Dynamic Voltage and Frequency Scaling (DVFS) is a widely used power saving technique in embedded devices. The goal of DVFS is dynamically controlling clock frequency to match the actual requirement of the current task, and at the same time permitting the reduction of the supply voltage which has a significant impact on power consumption. However, employing DFVS for a processor is not trivial, since it requires a thorough analysis of processor behaviour at multiple frequencies and voltage levels. Moreover, the DVFS system introduces different kinds of overheads, like a slow voltage transition. DFVS can be extended using techniques like Razor, which dynamically detects and corrects timing-related errors, and controls supply voltage to get rid of the worst case voltage save margins.

4.2.1 Clock gating

Clock gating is a register transfer level technique for power saving. In clock gating, a clock signal of a register is gated, so that the clock signal reaches the register only if a dedicated signal enables it. This reduces dynamic power consumption, since the register timed with the clock can change its output only on the clock edge.

If the standard cell library provides the integrated clock gating (ICG) cells, the synthesis software can employ them for automatic insertion of the clock gating cells. The benefit of clock gating is software dependent, and dynamic power consumption of infrequently used resources are greatly reduced.

In this thesis, clock gating is employed for the re-implemented version the ALF processor of Paper III. The revisited version of the processor, presented later on in this chapter, employs 28 nm standard cell technology, instead of the 90 nm technology used in Paper III.

4.3 Tailored architectures

By providing customized processing resources for different applications, it is possible to increase the performance of the system on a given power budget. Currently, high
performance embedded mobile platforms are typically Multi-Processor Systems on Chips (MPSoC) that integrate a wide variety of functions on the same chip [157]. Such platforms can include multiple programmable general purpose processors, fixed application specific accelerations, application specific processors, and reconfigurable hardware [44]. Fig. 19 illustrates the number of different processing units on a certain MPSoC mobile platform. The figure shows that there is a dedicated processing unit for many functions, like imaging, navigation, wireless communication, gaming and encrypting, for example. These kinds of application tailored solutions have had an important role in enabling performance scaling [158].

Different implementation options like GPP, GPU, ASIC, ASP, and FPGA can be exploited as a component of the MPSoC device. All these options have different characteristics regarding design complexity, power consumption, and performance. Table 1 presents roughly the tradeoffs between different implementation options.

**General purpose processors**

General purpose processors (GPP) like RISC-based ARM processors are widely used in many embedded devices since they are the main processor in many commercial off-the-
Table 1. Trade-offs between different implementation options.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Performance</th>
<th>Power</th>
<th>Programmability</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPP</td>
<td>--</td>
<td>--</td>
<td>++</td>
</tr>
<tr>
<td>GPU</td>
<td>++</td>
<td>--</td>
<td>+</td>
</tr>
<tr>
<td>ASP</td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>FPGA</td>
<td>++</td>
<td>--</td>
<td>--/++</td>
</tr>
<tr>
<td>ASIC</td>
<td>++</td>
<td>+++</td>
<td>--</td>
</tr>
</tbody>
</table>

shelf (COTS) SoC platforms. GPPs enable software-based application development flows and support for a wide range of applications. GPPs are typically optimized for making execution sequential, control-oriented code as fast as possible. The instruction sets of the GPPs are balanced to support commonly used operation. The energy efficiency of GPPs is typically poor, since optimizing for single threaded execution leads to the use of high clock frequencies (fast and leaky transistors, deep pipelining) and extra hardware for runtime scheduling, branch prediction, pipeline registers, cache management and so on. In the case of GPP, as well as all other types of software programmable processors, one of the main sources of energy consumption is instruction fetching and decoding, combined with operand loading from memory and storing to memory [159].

The energy efficiency of the GPPs can be improved by using multiple homogeneous processing cores combined with DVFS and different instruction set extensions like ARM NEON SIMD extensions [160]. Due to dark silicon problems some of high-end MPSoC have employed heterogeneous multicore GPP configurations, which have typically consisted of a set of low-power cores and set of high-performance cores. Since all cores share the same instructions set, the workloads can be swapped between cores on-the-fly. The recent high-end Qualcomm Kryo 485 GPP comes with three different processor configurations, each with their own clock domains: 1 × 2.84 GHz Cortex A75 Prime core, 3 × 2.42 GHz Cortex A76 and 4 × 1.8 GHz Cortex-A55. The Prime core is tailored for maximum single thread performance, with a larger cache and higher maximum clock frequency.

The main advantage of the GPPs is low development cost, thanks to widely available COTS processors based on ARM cores and the availability of the variety of software development tools. In addition to ARM cores, there is also free IP-blocks based on open-source architectures like Open-RISC [161] and RISC-V [162], that can be used as general purpose processors.
**Graphics processing units**

Graphics Processing Units (GPU) are specialized processors for accelerating graphics processing. Nowadays, desktop GPU-architectures are massively parallel, incorporating hundreds of programmable rather simple computing units that are clocked with relatively slow frequency. GPUs have been shown to be very efficient for data parallel high throughput problems, not only in the graphics domain, but also other domains, like machine learning.

General Purpose Computing on Graphics Processing Units (GPGPU) is made possible for programmers through APIs, like CUDA and OpenCL, which provide access to the GPU hardware. Programmability and performance, combined with wide the availability of GPUs, have increased the popularity of GPGPU computing, and GPUs are employed in various application domains today. However, GPUs are not suitable for all kinds of applications. For example, applications which have low latency requirements.

**Application-specific circuits**

When an application demands maximum performance with ultra-low power consumption, implementation is often based on application-specific integrated circuits (ASIC). Typically, ASICs have been tailored for only one application, or a very restricted set of applications. Since the hardware is matched with application parallelism and control flow, energy efficiency (throughput per watt) of the ASICs are unbeatable when compared to other implementation options. However, ASICs have some drawbacks, that increase the design cost and the risks of ASIC projects:

- **Flexibility**: Since ASICs are fixed hardware for a specific task, their programmability is very restricted or non-existent. Updating functionality or fixing design errors is therefore difficult after the chip has been manufactured, which increases risks, verification time and reduces life cycle of the product.
- **Design-time**: The design of an ASIC is a hardware design task which includes several design phases and design iterations. Through these phases, hardware designers have to resolve complex timing problems, and perform exhaustive verification to ensure a fully functional chip. The problems are increasing with design complexity, and an increased amount of engineer-years will be needed to implement future applications.
Application-specific processors

Application-specific processors (ASP) are processors that are tailored for a particular set of applications. ASPs are typically designed to fill the energy-efficiency gap [163] between programmable general purpose processors and fixed hardware accelerators. ASP designs are aimed to achieve defined performance while minimizing power consumption, silicon area, and design cost by offering programmability. The goal is pursued by architectural specialization and exploiting different types of parallelism. Architectural specialization can include implementing application specific instructions and data types. Moreover, ASP architecture can be matched to the application’s data flow by specialized interconnection and memory design. Typically, ASIP processor designs have taken advantage of the various type of parallelism, like ILP (VLIW-architectures), DLP (SIMD-units) and TLP (multi-core).

The flexibility of ASPs can provide many desired advantages over ASICs:

- Extended time-to-market window – product development can be started before all details of specifications or standards are matured.
- Reduced design time – fast software-based algorithm mapping flow to silicon.
- Reduced risks and extended life cycle – bug fixes can be easily provided by software updates, and new generations of the application can be deployed without hardware re-spin.
- Multipurpose and area efficiency – a single processor can be customized for multiple different workloads.

When compared to GPP solutions, the drawback of the ASPs is increased hardware and software design cost. Efficient design flows are therefore needed to harness the full potential of ASPs. Recently evolved ASP design flows [164, 165], especially advances in compiler technology have to make ASPs an economically reasonable design option. Advanced ASP design tool-chains are highly automatized and include tools like a re-targetable compiler, a re-targetable simulator, and automatic processor RTL-generator. Starting with initial processor templates, the designer can design and test several processor architectures targeting a specific application in a short time.

Examples of commercial ASP toolchains have been CoWare’s Lisatek [28] and Tensilica’s Xtensa [166], later acquired by Synopsys (ASIP Designer [164]) and Cadence (Xtensa Processor Generator [165]), respectively. The ASIP designer allows a high degree of freedom to design different types ASPs, like CISC, RISC and VLIW-style
architectures for example. The Cadence Xtensa Processor Generator is based on a RISC-style configurable processor template, Xtensa, whose ISA can be flavoured by customized instructions.

Lately, RISC-V, open-source, and extensible instruction set architecture (ISA) is gaining popularity [167, 162]. RISC-V have been developed to provide programmable processor base for custom accelerators [167]. Open-source Rocket Chip Generator tool [168] can be used for generating different variants of RISC-V architecture with possible customized hardware extension to accelerate an target application.

In this thesis, an open-source ASP design tool, TTA-based Co-Design Environment (TCE) [169] is employed. TCE builds on a fundamentally different processor template, with one instruction set architecture, transport triggered architecture (TTA) which can be considered to be a coprocessor rather than a general purpose main processor.

Field-programmable gate array

Field-programmable gate arrays (FPGA) are programmable logic which are based on numerous RAM-based lookup tables. Therefore, an FPGA can be configured to implement the same logical functionality as in the aforementioned GPPs, GPUs, ASICs, and ASPs, which makes FPGA the most flexible design option. Using the softcores [170], FPGAs enables rapid software-based application development flows. However, due to the overhead of field programmability, these implementations cannot compete against their hardwired counterparts in terms of performance or energy consumption. For example, the clock frequencies of an FPGA implementation can be more than a magnitude lower than a hardwired implementation [171]. To make the most of FPGAs, an application has to be highly parallel, and the best results are usually achieved by hand optimizing the register transfer level descriptions. Even though high-level synthesis tools (HLS) [172] have evolved over the years and reduced the design overhead [173, 174], they still have limitations that make them unfeasible for system-level synthesis [175].

4.3.1 System interconnection network and memory architecture

In general, the multiprocessor system consists of processing elements (PEs) and memory components that are attached to the interconnect network. The interconnection between components can be implemented in several ways. For example, PEs can be connected to the same shared memory, or PEs can have a direct connection between each other or a
shared bus along with the multiple attached PEs. Different interconnection network
topologies and memory architectures can have a significant impact on programmability,
performance, energy consumption and scalability. Complex interconnection networks
can consume a considerable amount of energy to drive high capacitive loads [159].
Reducing interconnection and tailoring memory architecture [134] by the actual needs
of the application, notable energy savings can be achieved [176].

Using a dedicated memory for a different purpose can yield performance improve-
ments or reduced energy consumption [177]. For example, processors can have separated
physical memory address spaces for private data memory, locally shared data memory
and globally shared, which can be varying in size and implemented by using different
technologies.

Multi-port memories components enables simultaneous reading and writing from
memory. They are used in the implementation of register files, and video memory for
example. Multi-port memories allows increased memory bandwidth and improve the
parallel scalability in the case of the shared memory architectures. The disadvantage of
multi-port memories is increased energy consumption and silicon cost.

DRAM memory performance scaling has been a problem (memory wall problem)
for a long time [178]. Whereas SRAM memory can be accessed in a few cycles, DRAM
memory access latency can be hundreds of cycles. The memory wall problem is often
mitigated by exploiting hierarchical memory organizations, like multilevel on-chip
SRAM cache memories, which can be over ten times more energy efficient when
compared to DRAMs [177]. Caches can consume a significant amount of the total power
of the chip, and many embedded systems employ scratchpad memories [179, 134].
Recently, Embedded DRAMs (eDRAM) have been viewed of as being an appealing
implementation option for on-chip memories due to their higher density and lower
leakage power when compared for SRAMs [180]. The disadvantage of eDRAM is that
memory cells need periodic refreshing which reduces the performance and increases
energy consumption.

The design flow proposed in Paper V enables data path architectural customization
points for PEs like functional units (FU), register files (RF), operation set (custom
operations) and interconnection network between the RF and FUs. On the system
level, the designer can optimize an on-chip interconnection between PEs and memory
components to match the application dataflow. Moreover, the designer can easily
experiment performance difference between a multiport memory and an arbiter based
single port memory.
**Uniform memory access**

The Uniform Memory Access (UMA) is a typical memory architecture for multicore processors. In the UMA model, the main memory is shared between each core, and access times for memory are uniform. ARM Cortex A15, which has been used in the experiments of Paper IV and Paper V, is an example of a UMA model processor. In the processor, each core has a dedicated L1-cache memory, but a coherent L2-cache shared among all cores. Another example of a UMA processor is the Intel Haswell i7 desktop processor, which has been used in the experiments of Paper IV. The Intel i7 has shared L3-cache and core dedicated L1 and L2 caches. Because of the cache memories, the aforementioned processors require techniques to maintain coherency of shared caches and main memories. Although the hierarchical memory architecture alleviates the communication bottleneck of shared memory, if tens of the cores are connected to the shared memory, coherency can be complex to manage. Despite the scalability issues of the UMA, it offers an easy programming model since all cores have a uniform memory view.

**Non-uniform memory access**

*Non-Uniform Memory Access* (NUMA) [181] is memory design which is typically used in server systems where several multicore processors are connected to together. A NUMA system (see Fig. 20) can consist of several *nodes*. Each node comprises typically a symmetrical multicore processor and a local main memory. It is common that processor cores share an L3 cache memory for fast inter-node communication. Nodes can communicate with each other via interconnect. Typically, currently used NUMA systems are cache coherent, which means that all memories in the system are visible for every core, and the cores can access memory data that resides on remote nodes. That is a clear advantage from the perspective of a programmer. However, memory access times are non-uniform since access to remote memories has longer latency when compared with access to local memories. This can lead performance pitfalls if the programmer or a compiler is unaware of the NUMA architecture. In Paper IV, this problem is discussed in the context of the RVC-CAL dataflow programs, and the problem is discussed later in section 5.3.
**Hybrid memory architecture**

Yviquel et al. [182] described a memory architecture that they called *hybrid memory architecture* (HMA). In the architecture, each PE has its private data memory and instruction memory. Inter-PE communication is performed through shared memories that form a communication network between cores. This kind of architecture is natural for dataflow programs since the local data of actors can be stored in the private memory, and incoming and outgoing tokens can reside in shared memory. HMA divides memory into small subcomponents, which reduces memory pressure, provides simultaneous R/W access and reduces power consumption when compared to the global shared memory architecture used in many GPPs [182]. The thesis employs HMA in work of Paper I and Paper V.

### 4.4 Transport triggered architecture

Transport Triggered Architecture (TTA) processors resemble VLIW processors in terms of fetching, decoding and executing multiple instructions during each clock cycle, offering *instruction level parallelism* [183]. However, TTA datapath have fundamental
Fig. 21. Comparison between datapaths of TTA and VLIW processor architecture. © 2019 IEEE.

difference compared to VLIW: Each FU and RF of TTA is connected to bypass network whereas FUs of VLIW have direct connect to RF. That is, in TTA every data movement is under control of programmer and VLIW is programmed by operation level granularity. Fig. 21 illustrates differences in the processor datapaths.

In TTA, operations take place as side effects of data transfers, controlled by move instructions that are the only instruction of TTA processors. Using move, data is transferred between function units (FU) and register files (RF) via transport buses. FUs are logic blocks that implement different operations, such as additions or multiplications. Depending on the set of operations included in FUs, the FU has one or more input ports and registered output ports. In every single FU, one input port is a triggering port and data moved to it triggers an operation in the FU in question. If an operation has multiple operands, it is assumed that all the other operands are transferred to FU input ports before or at the same time as the operand which is going to be written to the trigger port. After triggering an operation, the operation result can be moved from an FU output port to the input ports of one or more FUs/RFs, which makes TTAs exposed datapath processors, enabling many compiler optimizations.

For example, in the case of software bypassing optimization [184], intermediate results are not necessarily transferred through registers between uses, which reduces
register pressure. Since register operations can consume a significant share of the total processor power, software bypassing can reduce power consumption significantly [185].

In the case of TTA, the compiler has the responsibility of optimizing and scheduling data moves. Therefore the performance of a TTA relies on the capabilities of the compiler. On the other hand, when scheduling decisions can be made at compile time, the runtime hardware is more straightforward, reducing the power consumption and the silicon area of the system [183].

Fig. 22 presents a simple TTA processor, which has three transport buses (black horizontal lines), a load store unit (LSU), an adder, a multiplier, and one RF. Small squares are FU ports, and a cross inside the square indicates the triggering port. The instruction fetches and the decode unit is responsible for loading one instruction per bus from the instruction memory, and executing them. Three transport buses enable executing three instructions in parallel during each clock cycle.

For example, in one clock cycle, we can simultaneously move a result from the LSU to the input port of the adder unit using bus 0, move the \textit{mul} result to the adder’s triggering port using bus 2 and move the result of the previous add operation to the RF using bus 3. The previous example can executed in a single clock cycle and presented by one TTA assembly instruction word

\[
\text{LSU.out1->ADD.in2, MUL.out1->ADD.in1.t.add, ADD.out1->RF.1;}.\]

FUs and RFs are connected to transport buses via sockets (rectangular vertical blocks), and the arrows above the sockets indicate the input/output direction of a socket. The black dots on the sockets indicate where FUs/RFs are connected to the transport buses. If all the sockets are connected to all the buses, a processor is said to be fully-connected and a compiler has complete freedom in optimizing data moves. However, the full connectivity may lead to high routing congestion and low maximum clock frequency in the place and route stage of the processor design flow.

Fig. 22. A simple TTA processor. © 2019 IEEE.
Similar to DSP-processors, also TTAs use the Harvard architecture where program and data memories are separated. Additionally, TTAs can have multiple data memories, and each of them appears as a different address space to the programmer.

Implementing full context switch support to TTAs would be unfeasible due to TTA’s many registers within (pipelined) FUs. For that reason, interrupts and pre-emptive multitasking are commonly unsupported, and TTAs are used as slave co-processors of a master GPP that handles control-intensive software, such as an operating system [186].

One of the main challenges of TTA-based solutions is the need for a considerable amount of instruction memory due to wide instruction width, which is typical for TTA architectures. Moreover, serial parts of application code efficiency is poor since most of move slots are no operations (NOPs). These problems are well-known in the TTA community [187] and several techniques to address the problem are proposed. In [187], Jääskeläinen et al. noted that careful interconnection design can lead to smaller instructions. They also propose use of multi-level instruction memory hierarchy to mitigate the problem. There is also several works that propose use of different kind of code compression methods to get instruction memory footprint smaller [188, 189].

4.4.1 TTA design flow

TTA processors can be designed using the open-source TTA-based Co-design Environment (TCE) [169]. TCE supports the design and simulation of single-core [190] and GPU-style data parallel symmetric multicore [191]. Moreover, Paper V presents a TCE extension that allows task parallel heterogeneous multicore system design. The TCE design environment includes the graphical processor designer tool (Prode), which can be used to produce Architecture Definition Files (ADF). An ADF defines the type and amount of resources (e.g., FUs, buses, sockets, and RFs) and their interconnection [192].

Using the TCE compiler, tcecc, high-level programming languages (C/C++, OpenCL) can be compiled to TTA machine code. Tcecc is a retargetable compiler, adapting itself to the designed architecture, and automatically taking advantage of added processing resources. TCE offers a cycle-accurate simulator for analysis of program execution on a target TTA processor. The re-targetable compiler and processor simulator enables fast design space exploration.

A synthesizable RTL description of the given processor design can be created using the TCE processor generator tool. TCE provides a hardware database (HDB) which incorporates implementations for an extensive set of FUs and RFs. Designers can define
Fig. 23. Simplified TTA-design flow using TTA-based Co-Design Environment.
their own special operations by testing functional simulation models of operations (C/C++) for the TCE operation set simulator and the hardware description (VHDL or Verilog) of the operation for the HDB.

A typical TTA design flow using TCE is illustrated in Fig. 23. The TTA-processor design follows the HW/SW Co-Design principles where hardware design is tightly coupled with software design, and the design flow is an iterative process where hardware and software are optimized towards the design requirements based on the feedback of the software code profiling, hardware synthesis, and power simulations. TCE offers efficient profiling tools so that the designer can easily recognize the hotspots of the code and processor architecture and customize the processor operation set for the application code. Moreover, TCE offers a set of design space explorers that can be used to automatically optimize the processor architecture for the given application. Further details of the TTA design flow are given in [169, 191, 190, 2].

4.5 Contribution: TTA processor architectures for HEVC in-loop filters

Now that an overview of efficient computing technologies has been presented in general, an ASP case study for HEVC in-loop filtering is presented.

Video coding is a heterogeneous application consisting of various types of algorithms, and it is an example of a stream application, where continuous data stream flows through different chained processing stages from input to output and each processing stage includes different possibilities for parallelization and acceleration.

In Paper I and Paper III two different TTA cores for HEVC video coding are presented. Paper I proposes a homogeneous multicore TTA architecture for DBF and SAO (referred to Inloop-TTA) whereas Paper III proposes a single TTA-core for ALF (referred to ALF-TTA). Next, a summary of the relevant parts Paper I, Paper II and Paper III is given and readers are asked to refer the original papers for further details.

4.5.1 Architecture of TTA-cores

Both of the TTA cores are targeted for filtering $1920 \times 1080$ at 30 fps video, which is about 62 million luma samples per second. This means that a processor clocked at 300 MHz can consume maximum of five cycles for one sample. Fig. 8 shows that this is a quite challenging task, since the processing flow of a single sample requires...
eight stages for ALF with optimal parallelism. The DBF and SAO algorithms are less
demanding, but they include more control code than ALF.

The performance requirements are achieved by customizing the operation set for the
targeted algorithms, providing data parallel resources, and exploiting ILP. However,
the opportunity for accelerating other signal processing algorithms is maintained by
avoiding hardware optimizations that are too specific and predominantly relying on
operations that the TCE compiler can automatically utilize.

The Inloop-TTA (see Fig. 10 in Paper I) has five transport buses, which practically
means that it can execute approximately two operations in one clock cycle (if we assume
operations that require two input operands). Since ALF is a more demanding algorithm
than DBF or SAO, ALF-TTA is equipped 15 transport buses. This enables a high
degree of parallelism like, for example, ALF-TTA can have seven perform multiplying
operations (see Fig. 8 stage 2) and four additions (see Fig. 8 stage 1) in the same cycle,
if assumed that filter coefficients are already moved to multiplier outputs. The TCE
compiler can extract the static ILP of the filter algorithms for the proposed architectures
quite well, and the average utilization rate of transport buses is over 90% in the main
loop of algorithms.

The interconnection networks have been optimized to achieve higher clock frequen-
cies and energy efficiency. The IC of the Inloop-TTA is hand optimized using mainly the
following strategy:

- If an FU has multiple input ports, the ports will not be connected to the same bus.
- FU output ports are connected to all buses.
- RF output ports are not connected to the same bus as each other.

In the case of ALF-TTA, the interconnection network is first reduced using the TCE
automatic design space explorer tool, connection sweeper, which has its main focus
on reducing RF number connections. After that, the aforementioned hand-optimizing
strategy is utilized.

Special operations

Special operations usually improve performance by reducing clock cycles that is
needed to perform desired function. Moreover, special operation can reduce needed
instruction memory size, which enables use of smaller memories or more aggressive
code optimization to further improving performance. Attractive targets for special
Fig. 24. TTA architecture for HEVC HM 7.0 Adaptive Loop Filter.
instructions are functionalities, which can be easily implemented in hardware, but need multiple clock cycles when performed by software using basic instruction set. For example, some bit manipulation tasks needs only wiring when implemented using hardware, whereas multiple operations can be needed if basic instruction set is used.

In the case of the in-loop filter algorithms, sub-word parallelism of 32-bit memory operations is used to fetch pixel data from memory. 8-bit pixels are packed into 32-bit words, and therefore four pixels can be fetched and stored simultaneously. For that, both proposed architectures have special load-store operations that can separate the 32-bit memory word into four 8-bit pixels, and join four 8-bit pixels into the 32-bit memory word.

A clipping operation, which saturates the sample value to the desired range limits, is widely used in image and video processing algorithms. Both of the architectures have a special operation for clipping.

Since DBF and SAO include some control code for the filter decision, Inloop-TTA incorporates a selector-operation for speed-up branch predication [193], which is used to eliminate conditional branches. For SAO, an Inloop-TTA operation set is extended by a rather special pipelined operation for edge offset-filtering.
4.5.2 Multicore TTA for pipeline-parallel in-loop filtering

To increase the throughput of DBF and SAO, the multicore system (see Fig. 25 a)) comprising three instances of the Inloop-TTA core, as proposed in Paper I, to exploit pipeline parallelism. The in-loop filtering is divided into three tasks, which are de-blocking of vertical edges (DBF-VE), de-blocking horizontal edges (DBF-HE) and SAO filtering and each task is mapped to a dedicated but identical Inloop-TTA core (see Fig. 25).

The interconnection of the system forms the pipelined chain of cores and ping-pong memories. That is, each core can communicate with two other PEs (other Inloop-TTA cores, Source PE or Sink PE) via ping-pong memory (one with read-only access and one with write-only access). Each ping-pong memory is implemented by two random-access memory units and a memory swap unit, and it allows double buffering (both PEs connected to ping-pong can have simultaneous access to memory) without a need for expensive dual-port memories. The control unit schedules the memory swaps to change accessors of the memory components. The functionality of the control unit it is discussed more in the next Chapter 5, which deals with scheduling.

In some dataflow-based multicore systems like in [194], intercommunication is implemented using hardware FIFO memories instead of random-access memories. In some cases, like in an audio processing application where a signal is 1-dimensional and can be processed in-order, FIFO-memories are efficient. However, in the case of image processing applications, FIFO-memory based implementation can require complex data rearrangements so that PE can efficiently process data. In the worst case, a big chunk of data has to be copied to the PEs private data memory before processing. By using random-access memory, complex data rearrangements and data copying can be avoided. However, the advantage of FIFO-memories is the reduced read latency. Nevertheless, static instruction scheduling can often schedule random-access loads earlier to compensate for longer latency.

4.5.3 Experiments

This subsection summarizes the relevant experiments which were conducted for Paper I, Paper II, Paper III and Paper V, but includes also some unpublished results.

Table 2, Fig. 26 and Fig. 27, present some key figures of the proposed TTA cores and the task parallel multicore TTA-architecture. Both cores are placed and routed
Table 2. Hardware resource of Inloop-TTA and ALF-TTA cores.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Inloop-TTA</th>
<th>ALF-TTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwidth</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>ALUs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adders</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Relational ops fu</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Logic ops fu</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Bitwise ops fu</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Multipliers</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>LSUs</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Int RFs</td>
<td>5 × 16</td>
<td>6 × 16</td>
</tr>
<tr>
<td>Bool RFs (1 bit)</td>
<td>1 × 2</td>
<td>1 × 2</td>
</tr>
<tr>
<td>Special Ops</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Buses</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Partial</td>
<td>Partial</td>
</tr>
<tr>
<td>Instruction Width (bits)</td>
<td>131</td>
<td>314</td>
</tr>
<tr>
<td>Synthesis config</td>
<td>28 nm</td>
<td>28 nm / 28 nm ICG</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>530 MHz @ 0.8V</td>
<td>313 MHz @ 0.8V</td>
</tr>
<tr>
<td>Gate count (kGE)</td>
<td>106</td>
<td>158 / 131</td>
</tr>
<tr>
<td>Core power</td>
<td>10.2 mW</td>
<td>29.1 mW / 10.4 mW</td>
</tr>
<tr>
<td>Performance (FHD fps)</td>
<td>26 (DBF+SAO)</td>
<td>30 (ALF)</td>
</tr>
</tbody>
</table>

using 28 nm standard cell technology to get real power and area estimates. Since there is no access to 28 nm memory compilers, those physical characteristics and timing models of memories, which satisfy the access time requirements are extracted using CACTI-P [195], and then used for creating the required input files for the synthesis, and the place and route tools. The performance numbers are based on cycle-accurate simulations considering filtering on a luminance-channel only unless otherwise mentioned. Pessimistic estimation for a performance of implementation that also includes chroma-channels (4:2:0 sampling) can be calculated by adding a 1.5 × overhead for the reported results. For easier comparisons with other works, the chrominance overhead is added to the reported energy efficiency results.

A single Inloop-TTA core can perform DBF and SAO filtering in real-time for 1920 × 1080 All-intra video sequences are on a 530 MHz clock, whereas the pipeline-parallel multicore architecture can perform real-time filtering of 4K RA/LD videos.
using a 1.2 GHz clock. An ALF-TTA core can perform adaptive loop filtering in 30 FHD frames per second using 313 MHz clock frequency.

The power consumption of a single Inloop-TTA core at 530 MHz (0.8 V) with 8 kB data and 64 kB instruction on-chip SRAM memory is about 19 mW. The fetching instruction of a wide instruction in every cycle makes instruction memory the significant contributor to total power consumption by consuming about 8 mW. When considering the power consumption distribution of the core (Fig. 26), the register files (4 mW) and the interconnect network (1.7 mW) account for over half of the total power consumption of the core. Fig. 27 presents the power consumption of the multicore system running at 530 MHz and 1.2 GHz clocks, and core voltages of 0.8 V and 1.0 V, respectively. At 1.2 GHz the total power consumption was 207 mW and in the case of a 530 MHz clock, the total power consumption was reduced to 66.6 mW. In terms of energy efficiency (including also the chroma channels), the single Inloop-TTA core consumes about
Table 3. CHStone benchmark (µs) comparison on 100 MHz clock rate.

<table>
<thead>
<tr>
<th>Application</th>
<th>Inloop TTA</th>
<th>ALF TTA</th>
<th>SIMD TTA</th>
<th>NIOS II [190]</th>
<th>mBlaze [190]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA</td>
<td>2685</td>
<td>3108</td>
<td>3298</td>
<td>6040</td>
<td>14933</td>
</tr>
<tr>
<td>BLOWFISH</td>
<td>4993</td>
<td>5674</td>
<td>4885</td>
<td>10854</td>
<td>16714</td>
</tr>
<tr>
<td>AES</td>
<td>117</td>
<td>659</td>
<td>270</td>
<td>591</td>
<td>734</td>
</tr>
<tr>
<td>MOTION</td>
<td>11</td>
<td>67</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPEG</td>
<td>19235</td>
<td>26507</td>
<td>25806</td>
<td>180017</td>
<td>231131</td>
</tr>
<tr>
<td>GSM</td>
<td>81</td>
<td>120</td>
<td>129</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>ADPCM</td>
<td>237</td>
<td>489</td>
<td>463</td>
<td>1030</td>
<td>1562</td>
</tr>
</tbody>
</table>

0.56 nJ/px whereas the multicore Inloop-TTA systems consume about 0.74 nJ/px (530 MHz, 0.8 V) and 0.99 nJ/px (1200 MHz, 1.0 V). However, if the clock frequency of a single Inloop-TTA core (1295 MHz, 1.0 V) is raised so that performance is the same as multicore Inloop-TTA at 530 MHz, the estimated energy consumption is about 0.81 nJ/px.

The measured power consumption for the ALF-TTA core is 29 mW without memories, but when the dynamic power optimization technique called clock gating is employed, power consumption is reduced by 64% to 10 mW (0.23 nJ/px). A similar power consumption reduction in TTA processors when employing clock gating is also reported in [196].

The CHStone benchmark suite [197] was used to evaluate the general purpose of the proposed TTA-cores. CHstone provides C-implementations of different algorithms from various domains from security to wireless communications. The benchmark results are presented in Table 3. For comparison, the results of Nios II and mBlaze soft-cores and an SIMD TTA tailored for computer vision are also included in the table. The results show that TTA processors can easily beat softcore processors in all benchmarked applications. Especially, Inloop-TTA (106 kGE) shows excellent performance-area efficiency, beating the ALF-TTA (122 kGE) and SIMD-TTA (163 kGE) in most of the test cases.

4.5.4 Related work

Paper I, Paper II and Paper III present related research work and gives comparisons between the proposed implementations and related work.
GPP-based software implementations of a video decoder have been presented in many works [198, 199, 200]. For example, Raffin et al. [200] propose a highly energy optimized 21 nJ/px software HEVC video decoder for an ARM cortex based multicore platform. If the Full HD resolution at 30 frames per second is considered, the energy consumption of 21 nJ/px means about 1.3 W power consumption.

GPUs have been used for dequantization and an inverse transform [201], motion compensation [202], intra-picture prediction [203] and in-loop filters [204, 205]. Most of GPU-based implementations of video coding algorithms, found from the literature are targeted at discrete desktop GPUs. Only [205] proposes a mobile GPU for video coding, but it uses a high-end mobile GPU (GK20A Kepler-based GPU with 192 CUDA cores) which can have over 8 W power consumption. Based on [205], the estimated consumed energy is about 100 nJ/px in the case of HEVC video decoding.

There are several commercial [206, 207] and academic [43, 52, 208] ASIC hardware video decoders supporting the HEVC standard. Verisilicon’s Hantro G2 is a multi-format decoder which supports both HEVC and VP9 standards [206]. Verisilicon reports that power consumption of the Hantro G2 is 233 mW (0.44 nJ/px) when decoding 4K video at 60 fps and implemented using the TSMC 28 nm HP manufacturing process. Tikekar et al. [52] report that their decoder, which is targeting wearable devices and implemented with a TSMC 40 nm LP manufacturing process, consumes energy from 0.35 nJ/px up to 0.77 nJ/px depending on the coding options.

Abeydeera et al. [209] proposed an FPGA-based implementation of an HEVC decoder, which can process 30 4K frames per seconds. They reported that the implementation on a Xilinx Zynq 7045 platform consumes 126k lookup tables, 58k registers, and 335 18 kb block RAMs. As in the case of other comparable FPGA-based implementations [210], they do not provide power consumption figures for the solution.

**ASP for video coding**

Motion estimation has been a typical target for ASPs [211, 212] since it is one of the most computationally complex parts of video coding [7], and there are several variants of motion estimation algorithms with a similar type of computation [213], that make a customized instruction set reasonable.

One of the commercial ASP examples are TriMedia’s TM3260 and TM3270 [214], that are targeting H.264 and MPEG2 video standards. The processor is a 5 issue slot 32-bit VLIW-processor, which provides SIMD operations and special operations for
CABAC entropy coding. In [215], H.264 CABAC is accelerated using TTA architecture, which has nine transport buses, three ALUs, multiplier and one specialized FU for CABAC. The architecture was able to decode 3 Mb/s.

An HEVC interpolation filter and motion compensation implementation based on the configurable processor architecture, Xtensa, is proposed in [216]. The implementation exploits 8-way 16 bit SIMD operations and special instructions to tackle the overhead of misaligned memory access. The clock frequency of the customized processor is 630 MHz (40 nm process) and its estimated power consumption is 0.6 mW/MHz. The authors report that they can decode HEVC video 416×240 at 30 fps on the ISA extended processor at 500 MHz. This means about 100 nJ/px, but it is worth mentioning, that they have not optimized the other parts of the codec.

Janhunen et al. [217] proposed three different variants of a TTA-based ASP for a de-blocking filter with multi-standard support (H.264 and VP9). The architectures have up to 23 data transport buses taking advantage of instruction level parallelism, and 4-way 32-bit SIMD operations are supported and heavily used when exploiting available data parallelism of de-blocking algorithms. The authors reported that their best architecture can perform de-blocking 1920×1080 FHD video at 56 fps, whereas the corresponding multi-standard fixed hardware solution [218] is only 1.15 times faster.

Yviquel et al. [182] implemented the dataflow-based HEVC decoder using 12 symmetric TTA-cores. Their core includes 18 transport buses, three ALUs, a multiplier and three RFs with 14 slots. The implementation can decode five 720p frames in a second when it was estimated that the system could have a 1 GHz clock frequency. They report that 22% of the total time was consumed in in-loop filtering.

Alizadeh et al. [219] proposed ISA extended RISC-V architecture for an HEVC de-blocking filter. Their extended ISA includes very similar custom scalar instructions to those presented in Paper I: absolute value, clip and fused operation (a-2b+c). The authors claim that they can filter 1920×1080 at 30 frames per second by using the 900 MHz processor clock frequency.

4.6 Conclusion

Mobile GPPs are not ideal for long-term high-resolution video playback in the case of small form factor devices, like smartphones and wearable devices, due to their power consumption and heat dissipation. However, GPPs can be considered to be a suitable solution for devices where the heat dissipation and power consumption is not a problem.
Current video coding algorithms have better support for data-parallel computing, making GPU-based acceleration reasonable. However, most of GPU-based video coding implementations are targeted at discrete desktop GPUs, and currently, there are no energy efficient GPU implementations. Still, it is very likely that energy optimized embedded GPU implementation can get beyond 20 nJ/px in energy efficiency. The lack of the low-power GPU implementations can be explained by the fact that low-end GPU offers poor support for GPGPU APIs, which hinders software development [2].

ASICs are traditionally facilitated for video processing, due to the tight power budgets of embedded mobile devices. The downside of dedicated hardware video coders is that they are typically restricted in terms of the video coding standard, profiles, and configurations. That also set limits on content providers, since they have to stick to a restricted set of coding options, like specific standards, encoding tools, resolution and frame rates. As already discussed in Chapter 2, several video coding standards are actively used and offering fixed HW acceleration for all of them can be economically unsustainable.

Several works have proposed ASPs for accelerating the different algorithms of video coding. The main motivations to use ASP can be considered to be an energy and cost efficient way of offering support for multiple standards, various coding options and also other applications with similar computational requirements. Both TTA cores proposed in this thesis are capable of FHD real-time filtering with a very low, less than 100 mW, power consumption, making them suitable for embedded solutions where programmability is needed. According to the author’s best knowledge, the architecture proposed in Paper I offers the best solution for an HEVC de-blocking filter if energy efficiency and software programmability are considered. Paper III was the first and only application specific processor which has been designed for a very compute intensive HEVC adaptive Loop Filter.

Based on the experiments, it can be estimated that the whole HEVC decoder implemented using multiple heterogeneous tailored TTA-cores with power optimizations [196] could reach FHD real-time HEVC video decoding with less than 2 nJ/px energy consumption (based on the assumption that the in-loop filter share is about 20% of the total decoder complexity). This would be about one order of magnitude more than customized ASIC solution [52], but an order of magnitude less than the embedded multiprocessor GPP solution [200].
5 Mapping and scheduling of dataflow graphs

Current embedded high-end SoC platforms have a huge amount of computation power, however harnessing capabilities of highly parallel and heterogeneous architectures efficiently has been challenging [158]. High-level abstractions like dataflow models have proposed to be potential solution for more efficient exploitation of the resources of parallel platforms. Before a dataflow graph can be executed on a platform, the following issues have to be considered:

– **Mapping**: How are PEs allocated for actors in the dataflow graph? Which mapping is optimal concerning communication cost, load-balancing, energy efficiency or some other desired property?

– **Scheduling**: How are PEs temporally allocated between actors that are assigned to a particular resources? How are the actor firings ordered and how do we determine if the firing rules are met? Which schedule is it that optimizes some a particular property, like latency or the required memory space?

It is crucial to bear in mind the targeted computation platform when an efficient solution is being explored for the problems mentioned above. In the ideal case, there is a single portable model of an application, and a dataflow compiler tool that can be used to synthesis (software or hardware) for different target platforms so that the implementation optimally adopts all the resources of the target platforms. In practice, this is a very challenging task even in a restricted set of possible target platforms. Like already discussed, possible target platforms can have a varying number of heterogeneous processing cores: GPPs, FPGAs, DSPs, ASP and hardwired accelerators. In addition, memory hierarchy and intercommunication networks can differentiate and introduce different speeds to communication channels. The dataflow compiler should be aware of the characteristics of those structures in order to achieve high quality implementations.

A selected dataflow MoC also influences the execution of the dataflow graph. For instance, the execution of static dataflow models can be differentiated from the execution of dynamic models due to better predictably of the former. When using a single-core processor to execute static dataflow graphs, a lock-free schedule (if it exists) can be determined at compile time [94], and execution of the graph is straightforward. In cases of dynamic dataflow applications, runtime scheduling is needed due to behaviour being dependent on data.
5.1 Actor mapping

In a single core compilation process, mapping refers to code selection and register allocation, that is operations and variables of code are assigned to the instructions and the registers of a machine. However, in this thesis, mapping refers to the context of multiple PEs and dataflow application, where blocks of code (actors) and logical communication links (FIFO buffers) are assigned to a PE and a physical communication medium (e.g., shared memory component). Fig. 28 illustrates a dataflow application (a) which is mapped to a platform (b).

Dataflow actor mapping defines which PEs are responsible for executing each of the actors. The goal of mapping is to find the optimal resources to execute the workload of the actors. Optimization criteria can be performance [220], power consumption [221] or thermal dissipation [222], for example. Multi-objective optimization methods are also proposed [223], where the goal is find pareto optimal mappings to offer trade-off between multiple optimization criterion.

The quality of mapping is emphasized in the case of heterogeneous platforms because PEs can have custom accelerators that are suitable for the workload of the particular actor. The finding of optimal mapping can be modelled as a graph partition problem [220, 224], where set of actors are divided to the number of subsets which equals the number of the PEs, and the number of possible mapping combinations can be
calculated as follows

\[ \text{numCombinations} = \text{numCores}^{\text{numActors}}. \]  

For example, if the application is partitioned to ten actors and the target platform has four PEs, then there are over one million possible mapping configurations. Therefore, exhaustive search to find optimal mapping is not acceptable when the actor count is high.

Several dataflow actor mapping algorithms are proposed in the literature and the work of [225] and [226] gives an extensive overview of the different approaches. Referring [226], the mapping algorithms can be divided into static and dynamic mapping methods.

### 5.1.1 Static mapping

Static mapping methods rely on parameters and metrics that are known at the time of design, and therefore they are unable to handle dynamic behaviour. Static mapping [227] is well suited in those cases where the workload of the cores and the performance of the cores remain somewhat stable since it does not introduce any runtime overhead. Static mapping can be defined manually or automatically in compile time. However, manual mapping of hundreds of actors on the platform that is constituted of many processing cores can be difficult, particularly in cases where the designer is unfamiliar with the application behaviour and the underlying hardware.

Profiling can be exploited to find static mapping automatically. The execution time of the dataflow graph with different mappings is measured, and the best mapping is selected. Since, in most cases it is not possible to profile all possible mappings, some heuristic [224] has to be used to reduce exploration space. For example, Chavarrias et al. present an automatic tool [228] for the static mapping of the actors into cores where they form groups of actors to reduce the problem size. The actor groups are formed based on hierarchy levels of a dataflow model.

### 5.1.2 Dynamic mapping

In dynamic applications, processing workloads can have such a vast temporal variety that runtime re-mapping is needed to balance workloads in order to prevent bottlenecks. Thermal issues can cause such unpredictability that dynamic mapping has to be considered. For example, the temperature of the core might run over the thermal budget.
of the system, and the clock frequency of the core has to be lowered, making migration of actors to other cores beneficial.

Dynamic mapping methodologies can be further divided to on-the-fly, hybrid and hybrid with runtime remapping. The on-the-fly method uses runtime analysis to make mapping decision, whereas hybrid methods use both design-time and run-time analysis. The run-time remapping algorithms enable actor migration from one resource to another during the execution. [226]

In [220] Yviquel et al. present a mapping method that can be classified to the hybrid category and targeted at homogeneous platforms. They define three metrics that are exploited to find efficient actor mappings: resources, communication and performance. Resources are constraints that define the number of available processing cores. Communication refers to the connectivity of the actors, and one important goal is to minimize the communication between different processing cores by mapping actors that have a large amount of communication between each other to the same processing core. This causes overloading of some processing cores and underloading some other cores, and the objective is a balance of the workloads of the cores and avoiding performance bottlenecks. The workload of a core mainly consists of the workloads of the actors that are mapped to that core. Using these constraints, they solve graph partitioning using the greedy Kerninghan-Lin algorithm [229].

Ngo et al. [226] proposed actor mapping methodology with runtime remapping support, targeted at heterogeneous platforms. The method considers overheads caused by actor migration and varying communication latencies caused by data transferred through different communication media (e.g., Network on Chip or Bus). The graph partition algorithm used in the runtime remapping in [226] is based on the Fiduccia and Mattheyses algorithm [230] which is widely used in VLSI design tools.

Paper IV proposes a methodology where actors are divided into actor groups in design time, and the runtime remapping of the actor groups into PEs is left for the scheduler of the operating system.

5.2 Dataflow actor scheduling

Scheduling can refer to the temporal ordering of instructions to be executed. In the case of embedded processors like RISC, VLIW or TTA processors, the compiler statically determines the execution order of program instructions at compile time. General purpose
superscalar processors on the other hand have specialized runtime hardware which dynamically determines [231] the order of the instructions issued to execution units.

When considering the scheduling of a large grain dataflow graph to a platform that incorporates multiple PEs, a scheduler determines the order of actor firings (executing of a block of code) in each PE. According to Sriram and Bhattacharyya [232], application graph scheduling incorporates three activities, which are the processor assignment step, actor ordering step and timing of actor firing step. In this thesis, mapping refers to the processor assignment step, and scheduling refers to the local scheduling that happens in each PE, and includes the ordering step and the timing step.

The different dataflow actor scheduling methods can be roughly classified into three categories static, dynamic and hybrid methods, where the latter has characteristics from both static and dynamic. Static methods are efficient since their run-time computation overhead is small. On the other hand, their flexibility is restricted and in data depended cases schedules have to select worst case option. Dynamic methods offer flexibility but with a cost of run-time computation overhead. The requirements of an application and target platform are guiding the selection a proper scheduling strategy. Usually, in time-critical embedded applications static strategies are used. However, increasing dynamism of applications have increased demand for hybrid and dynamic scheduling strategies on embedded platforms [182, 233].

### 5.2.1 Static scheduling methods

Extreme use of static scheduling methods is fully static scheduling [234]. In that method, execution order and exact timing for actor firing are statically defined in design-time. This reflects the design of synchronous circuits where data availability in a particular time window is ensured using static timing analysis. The method is efficient, since there is no synchronization overhead. The disadvantage is that actor execution time can vary and determining the worst-case execution time could be hard or impossible. For example, the actors of HEVC In-loop filter (Fig.13) have high variations in execution times because of different filtering modes and the fact that filtering can be switched off on a CTU basis. In this case, the worst-case execution time can be calculated, but it would be so pessimistic that it would harm performance.

Self-timed scheduling [234] relaxes the requirement for guaranteed actor executions times of the fully static method. In self-timed scheduling, the exact timing information is omitted, and the actor ordering step is performed in design-time. The timing of actor
firing is decided at run-time by checking if the firing rules of the actor are met. This scheduling method introduces a synchronization overhead and hardware cost in order to enable mutually exclusive access to shared memory resources [232]. However, the advantage is that a compiler does not have to be able to consider the timing and other PEs in the case of a multicore platform. Therefore, self-timed scheduling is widely used, especially in SDF applications. In Paper I, the self-timed scheduling approach is used for orchestrating dataflow through the multicore pipeline. The runtime timing is implemented using the hardware control unit, which is discussed in more detail later in section 5.2.4.

5.2.2 Hybrid scheduling methods

Hybrid scheduling refers to methods which have combined static and dynamic characteristics in the ordering step and could utilize design-time and runtime information. The goal of hybrid scheduling methods is to offer a trade-off between flexibility and efficiency.

Quasi-static scheduling can be used when the data-dependent control paths of DFG can be somehow predicted (e.g., statistically). Self-timed schedules can be defined for different data-flow paths at design-time and the sequencing schedules in runtime (e.g., based on some conditions). Quasi-static scheduling is explained in detail and discussed in [83, 232]. In [86], flow-shop scheduling algorithms are exploited for low overhead run-time sequencing quasi-static schedules in a multiprocessor context. Boutellier et al. [235] presented a methodology for automatic discovery of quasi-static schedules of dynamic RVC-CAL dataflow programs.

A scenario-aware dataflow [236] model can also capture the dynamism of applications by determining different scenarios (e.g. operation modes). Static schedules for different scenarios can be determined in design-time and then switch between them in runtime. Other scenario-based approaches have been presented in [237, 104], for example.

5.2.3 Dynamic scheduling

Fully dynamic scheduling performs all scheduling decisions in runtime. It offers maximal flexibility but with the a cost of complexity. In runtime, the time window of
Dynamic dataflow MoCs like DPN requires dynamic scheduling methods, due to data-dependent behaviour. The two main methodologies to implement a scheduler for DPN are the round-robin scheduler and the combined data-driven / demand-driven scheduler [238].

In round-robin (RR) scheduling (illustrated in Fig. 29), the actor ordering step can be performed compile time, since DPN MoC allows non-blocking actor execution. Therefore, the scheduler goes through actors in a predetermined order and checks the firing rules. If the firing rules are met, the corresponding action is fired and if there is no action in the actor that can be fired, the scheduler proceeds to the next actor. In the multicore case, scheduling is distributed among local schedulers, so that each core has a dedicated local RR-scheduler as illustrated in Fig. 29.

In [239], the authors have investigated different scheduling policies that can be considered as derivatives of RR-scheduling. One of the derivatives they called non-preemptive policy. The a non-preemptive policy is like RR, but the same actor is fired consecutively until the firing conditions are not met. The non-preemptive policy is facilitated in Orcc TTA-backend [182], for example. Paper IV proposed a multicore C-backend for Orcc which facilitates dedicated local RR-schedulers. Moreover, in the case TTADF design flow (Paper V), each PE executes the dedicated local RR-scheduler, which handles the firing of the actors of the PE. The dedicated local RR-schedulers can be considered suitable especially for TTA processors since the order of the actor’s firing rule tests are already known, and a compiler can optimize the code of the dedicated scheduler better.
When many actors are mapped to a single core, the RR-scheduler can introduce an overhead when checking invalid firing conditions. A data-driven / demand-driven (DD) - scheduler tries to address this by making a runtime decision about the next schedulable actor. The decisions are based on information on the state of FIFO buffers. An actor could block if some of its *output buffers are full*, or if some of its *input buffers are empty*. Therefore, if an output buffer is full, the data-driven policy schedules the consumer actors of a particular FIFO buffer. In the case of where input buffer is empty, the demand-driven policy schedules the producer actors of a particular FIFO buffer.

A Demand-Driven scheduling policy is proposed by Kahn and MacQueen [240] for KPNs. Combined Data-driven / Demand-Driven scheduling was first proposed by Jagannathan and Ashcroft [241] and Parks discuss the dynamic schedulers for KPNs in his thesis [242].

Yviquel et al. propose a multicore DD-scheduler implementation for DPNs in [238]. Implementation of a multicore DD-scheduler is rather complicated since it requires intercommunication between the local schedulers. In addition, in many cases, careful analysis to enhance actor merging, mapping, and ordering could make DD-scheduling redundant.

**Scheduling of actions**

The firing of Khan processes can be implemented efficiently by an assign thread for every process. The blocking reads of KPNs ensure that a thread can be suspended in the case of a blocking channel and activated again when new data arrive in the channel. When a thread has been activated, the execution continues checking the blocking rule and rechecking it. If there is success, execution proceeds to check the next rule, or to fire the process. That is, there is no need to recheck all firing rules since they are already checked.

In the case of a DPN actor, the first step toward actor execution is the action selection phase (*action scheduling*), which could require testing of many firing rules to determine the action that can be fired. In a straightforward implementation like [243, 244], the firing rules are checked in round-robin order for every action by respecting action priorities (partial order to check firing rules of actions) and FSM state (acceptable order to fire actions). This kind of implementation can be inefficient, since multiple redundant firing rules have to be checked, and even multiple times, if any action is not fireable.
In [245], Cedersjö and Janneck propose translation of DPN actors to *actor machines* [246, 247] to improve execution efficiency. The main idea of the proposed method is that the controller of the actor machine is a state machine, and its memories test the results of firing rules and change state, based on the firing rules tests. Thanks to different states, there is no need to re-check rules that are known to remain unchanged. The downside of the actor machine approach is that a relatively simple actor can have several actor machine states, which increases the program code size.

Minimizing the number of firing rule tests is especially important in the case of computing architectures that have no branch prediction support, which is typical for TTA-based architectures. However, the code efficiency of TTAs is poor for control dominated code. Therefore, the optimization of action scheduling has to consider in a case-specific manner.

In paper V, the protothreads [153] based approach is employed to present actor machine controller-like action schedulers efficiently maintaining code readability. Listings 5.1 and 5.2 show action scheduling C-codes for a SPLIT-actor in a case of two similar implementations from [247] and Paper V. SPLIT reads the token from the port and depending on token value, fires action_0 or action_1.

### 5.2.4 Contribution: hardware scheduler for the in-loop filters

In Paper I, a coprocessor architecture for an HEVC in-loop is proposed. The architecture comprises three PEs, and for each PE, one task of the in-loop filtering is assigned as is shown in Fig. 25. Since execution times of the tasks of in-loop filtering are highly data-depended, runtime scheduling of the tasks is needed to minimize the idle times of the PE. The figure shows the control unit, which is responsible for orchestrating dataflow through the processing pipeline. All PEs are connected to the control unit, and they signal the control unit when they have completed their task. The control unit on the other hand signals PEs when they can start the processing or halt the processing. The control unit also swaps memory connections between the PEs so that simultaneous write and read access is possible by using a pair of single port memories (ping-pong-memory). To minimize the waiting times of the PEs, the following scheduling rules for the control unit can be determined:

- Swap Memory if both of the PEs which are connected to the ping-pong memory are ready.
int Schedule_Split()
{
    int v_ac0;
    int v_ac1;

    switch (next_state)
    {
        case 0:
            goto Split_State0;
            break;
        case 7:
            goto Split_State7;
            break;
    }

    Split_State0:
    if (TestInputPort(&A, 1))
    {
        v_ac0 = ReadToken(&A, 1);
        if ((v_ac0 >= 0)) {
            // body of action_0
            goto Split_State0;
        } else {
            v_ac1 = ReadToken(&A, 1);
            if ((v_ac1 < 0)) {
                // body of action_1
                goto Split_State0;
            } else {
                next_state = 7;
                wait();
            }
        }
    } else {
        next_state = 0;
        wait();
    }

    Split_State7:
    next_state = 7;
    wait();
}

Listing 5.1. Generated SPLIT-actor's action scheduling C-code in [247].

FIRE split(spliit_STATE + state)
{
    PORT_VAR("portA","v","int");
    PORT_READ("portA","v");
    if (v>=0)
        /* body of action_0 */
    else if (v<0)
        /* body of action_1 */
}

Listing 5.2. SPLIT-actor's action scheduling C-code in TTADF (Paper V).

108
Send start signal to a PE immediately after both ping-pong memories which are connected to the PE have been swapped.

- Halt PE immediately after the PE's task is finished.

Table 4 shows the percentage of active processing cycles (utilization) for different PEs when using four different quantization parameter values (QP), three main configurations (all-intra, random-access, low-delay) and two different video sequences. Variations in PE utilizations are significant between different coding configurations, and this reveals the data-dependent nature of the In-loop filters. For example, utilization of the PE that is assigned for SAO varies from between 50% to 93%. The results suggest that load-balancing across the PEs would be beneficial.

**Power management**

The proposed system offers power management handled by the control unit, which halts the PEs immediately when they are ready and wakes them immediately when data can be processed. The advantage of this approach is that PEs do not need to access shared memories occasionally to resolve if firing conditions are met. The halted PEs dynamic power consumption is therefore zero.

However, more advanced power management could also be used. For example, Melot et al. [248] have proposed a scheduler which makes use of dynamic voltage and frequency scaling (DVFS) and can generate energy efficient schedules for homogeneous manycore processors. In the case of in-loop filters, PE-wise DVFS based on the QP-value could be an option to reduce power consumption further. Table 4 shows that when the QP-value is increased, utilization of the SAO PE is reduced and utilization of the DBF PEs is increased.

It is possible to roughly estimate how much benefit the DVFS method would offer over the proposed method by considering that the optimal load balance between the PEs can be achieved somehow. The results of Paper II show that PEs can be clocked at 0.53 GHz with a voltage of 0.8 V and 1.2 GHz at the voltage of 1.0 V. Based on [249], linear line fitting

\[ V = 0.2985f_{clk} + 0.6418 \]  \hspace{1cm} (9)

is used to define the relationship between voltage \( V \) and clock frequency \( f_{clk} \). Assuming that voltage and frequency can be optimally adjusted, the reduced dynamic power savings can be calculated for DVFS using Eq. 7 and Eq. 9. Based on the data in Table 4,
Table 4. Utilization rate of PEs (DBF VE - DBF HE - SAO) in HEVC in-loop filtering. © 2015 IEEE.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>QP</th>
<th>All intra</th>
<th>Random access</th>
<th>Low delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasketballDrive</td>
<td>22</td>
<td>81 - 78 - 77</td>
<td>67 - 71 - 91</td>
<td>70 - 73 - 89</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>73 - 74 - 85</td>
<td>76 - 83 - 68</td>
<td>73 - 79 - 74</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>73 - 74 - 81</td>
<td>81 - 89 - 57</td>
<td>80 - 87 - 57</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>76 - 77 - 70</td>
<td>83 - 92 - 54</td>
<td>83 - 92 - 50</td>
</tr>
<tr>
<td>ParkScene</td>
<td>22</td>
<td>81 - 82 - 84</td>
<td>78 - 83 - 73</td>
<td>75 - 79 - 78</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>68 - 70 - 93</td>
<td>80 - 87 - 67</td>
<td>75 - 81 - 70</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>68 - 70 - 90</td>
<td>80 - 88 - 64</td>
<td>77 - 85 - 63</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>74 - 76 - 73</td>
<td>81 - 92 - 60</td>
<td>83 - 94 - 50</td>
</tr>
</tbody>
</table>

the average power saving would be about 15% and 18% in the best case if DVFS is exploited. However, perfect load balancing might be complicated to achieve, and the DVFS system introduces different kinds of overheads [154] (e.g., voltage transition latency) that should also be considered.

5.3 Contribution: OS-based runtime actor mapping and scheduling

Heterogeneous computing nodes and hierarchical cache memories combined with operating system software can make the system very unpredictable. Therefore, it can be challenging to define an efficient model for scheduling and mapping of actors. Avoiding naivety, the model should consider the behaviour of cache memories, varying PEs, the cost of different communication channels, the thermal effects, the operating system, etc..

Paper IV proposes an operating system based dynamic run-time remapping methodology for DPN based RVC-CAL actors. The motivation for the work was the observation that the state-of-art RVC-CAL framework Orcc could yield very slow speed-ups in Linux multicore systems, especially on a NUMA-based multicore server platform. Fig. 30 illustrates the problem in a case where a video motion detection application is scaled to the multicore Intel Xeon server.

The proposed method addresses the issue by forming a fixed number of static actor groups which are assigned to different threads. For each thread, there is a dedicated actor scheduler which is responsible for checking the firing rules in a round-robin fashion and the fire actors that are part of the particular group of actors. The POSIX Thread API has
been employed for the thread management, and the decision about remapping the actor groups into cores is left for the scheduler of the operating system.

In Paper IV, the proposed methodology is tested using five different applications and three different platforms, including a desktop multicore processor Intel i7, a server platform Intel Xeon with two NUMA nodes and an embedded heterogeneous mobile processor Samsung Exynos 5422. The proposed remapping methodology improves performance when compared to [220], which is adopted into Orcc, especially in the case of the server and the mobile platform. Fig. 31 and Fig. 32 summarize the advantage of allowing the OS scheduler to perform runtime-remapping.

The reason for the weak performance of the existing Orcc method in the NUMA system is that it pins actors to a particular physical core trying to minimize remote memory access. However, this approach considers only data locality, but not the congestion of the interconnect, nor the memory-access traffic is spread across the system. Moreover, the with existing method Orcc does not consider any information about the NUMA platform, for example how the nodes are connected.

The method proposed in Paper IV relies on AutoNUMA [250]. The Linux kernel has support for automatic NUMA balancing, which combines two basic strategies to
optimize the performance. The scheduler moves the threads (thread placement) closer to the memory they are accessing (CPU follows memory) and the data closer to the threads (memory placement) that reference it (memory follows CPU). The performance results of the NUMA system can be further improved by compiling the Linux kernel with Carrefour [251]. Carrefour is an advanced memory placement algorithm on traffic management the goal of which is to minimize congestion on interconnections or memory controllers.

In case of the heterogeneous ARM big-LITTLE architecture, the problem of the existing Orcc C-backend is that it naively assumes a symmetric multicore platform, but does not consider the a big-LITTLE case where cores are not symmetric. The method proposed in Paper IV relies on the Linaro Linux kernel scheduler, which is aware of the Arm’s big-LITTLE architecture.
The benchmark results for the Intel i7 platform show that the proposed implementation can exploit simultaneous multithreading (Hyper-Threading) better than [220], but in the most of the test cases, increasing the number of threads over the number of physical cores (4) hurts performance. This can be explained by the similarity of actor workloads, which can cause a bottleneck because of the limited number of similar hardware resources. SMT would be more beneficial if the workload of the threads, which are mapped to the same physical core, would have variation so that different hardware resources of the core could be used simultaneously. Because [220] uses CPU affinity, the linux scheduler is not able to migrate actors to other cores, which prevents load balancing and could also harm SMT performance.

5.3.1 Summary

This chapter presented different methods for dataflow actor mapping and scheduling. A self-timed hardware scheduler for orchestrating the flow of CTU-data packets through the processing pipeline is proposed. Moreover, the operating system based runtime actor mapping and scheduling are proposed for Orcc to take computation architecture better into account.

Especially advanced dataflow actor mapping methodologies are essential, since platforms are becoming more and more complex by introducing heterogeneity into the processing units, memory architecture and interconnect. Essential for efficient mapping is knowledge about the system architecture and the application behaviour. However, systems driven by operating systems and having hardware cache management are very unpredictable. Consequently, runtime mapping methodologies are needed to mitigate unpredictable and time-dependent application behaviour. Moreover, thermal-aware mapping methodologies are needed to tackle the increasing power density problems on hot chips.
6 From dataflow descriptions to application-specific TTA processors

In Chapter 3, dataflow-based application modelling was discussed, and different dataflow models of computations were presented. Furthermore, the advantages of using dataflow models for describing signal processing applications was outlined. Platforms that can be used to compute complex signal processing applications efficiently were examined in Chapter 4. Finally, in Chapter 5, methods scheduling and mapping the dataflow programs to different computation platforms were explored. The goal of this chapter is to tie the previous chapters together by presenting the dataflow-based multicore HW/SW co-design flow for signal processing applications where flexibility and energy efficiency are required. The chapter is directly related to Paper V, which generalizes and automatizes the design steps needed to design systems such as those presented in Paper I, and raises the abstraction level of the multicore SW/HW co-design with negligible impact on performance.

6.1 Contribution: TTADF design flow

Paper V presents the dataflow-based design framework for TTA architectures that respects the widely used Y-chart system design methodology [31, 32, 33] (see Fig. 3). The framework is called TTADF. TTADF integrates application specific processor development and dataflow programming by targeting power-efficient embedded solutions. TTADF builds on TCE by enabling design and simulation of task-parallel streaming applications that run on heterogeneous multicore architectures. A designer can rapidly evaluate and simulate different architectures that can comprise TTA cores for data-intensive processing and GPP cores for control-intensive processing. The TTADF design flow can produce a synthesizable register level description of the designed architectures and compatible software program binaries for them. This allows the possibility to adopt the hardware synthesis flow as part of the design loop for more reliable analysis results. Fig. 33 illustrates the TTADF design flow.
### 6.1.1 Applications

The primary goal of the Y-chart design methodology is to make application behaviour and system architecture independent from each other, and further, to separate communication and computation. This allows the use of high-level abstractions to describe the application without excluding any interesting implementation options in the early design phases.

The first task of the designer is to design an **actor network**, which is the top-level description of the dataflow application, consisting of a list of actors and the connections between them. TTADF employs dynamic dataflow MoC for application descriptions. To describe actor networks, TTADF employs dedicated XML-based syntax. When the structure of the application is sketched, the designer can proceed to refine the application model by describing the functional behaviour of new actors.

To describe the behaviour of the actor, C-language can be used. However, each actor description has to meet the predefined structure and specialized TTADF API calls have to be used for implementing framework-specific functions such as a communication I/O. C-language allows the use of legacy code and libraries, and optimized C compiler tools for a wide variety of platforms. Most importantly, the custom operations of TTA
processors can be exploited using automatically generated intrinsic functions that are vital for energy efficient implementation.

6.1.2 Platform architecture

A description of the platform architecture is needed to guide the code transformation from the high-level descriptions to efficient low-level software or hardware code. The architecture model gives details about hardware resources that can be utilized for computations. The architecture model includes information on available PEs and the memory organization that is used to communicate. PEs are further modelled using a lower level architectural model that gives more detailed information about a particular PE. For example, in Paper V, a specific meta-model is followed to describe the architecture model in XML-format, and individual TTA cores are further described using ADF [192], which is a cycle accurate model of the processor.

Once the initial application description is ready, the designer can proceed to design the architecture model. Although TTAs can be optimized to be more suitable for control code [252], they are primarily targeted to at data-intensive code. In Paper V platform architecture is therefore divided into the host architecture and the TTA Co-processing architecture. The host architecture could be GPP or MCU with a real-time operating system that has memory mapped access to the Co-processing architecture. The TTA Co-processing architecture includes TTA-based PEs and memory components, which are connected to them. The architecture model does not define any specific communication protocols, but it only defines the connection between PE components and memory components. Multiple PEs can be connected to the same memory (multiport or arbitrate) for intercommunication between PEs in order to form processing networks that match the data flow of the application.

6.1.3 Mapping

In mapping, the tasks of an application are assigned to system resources over time. On the system-level, actors are assigned into cores and data structures (like FIFOs) into shared memories. On a core-level, operation of instruction is assigned to a functional unit, and private data is mapped to local memories. In the case of complex interconnect networks, the mapping should also determine communication routes between processing units. The system compiler can be used to generate lower level code from higher level
Fig. 34. a) Actor network of a dynamic predistortion filter. b) architecture model of pipelined connected triple core In-loop TTA. c) Mapping of the application network of a) to the platform architecture presented in b). Revised from Paper V © 2019 IEEE.
dataflow abstractions. The system compiler can automatically perform actor mapping using similar methodologies to those discussed in Chapter 5, or lean on actor mapping defined by the designer. Based on the mapping information, the system compiler can generate refined code for individual cores. The low-level code (C, LLVM) can be compiled to machine code using the core’s compiler, which is responsible for scheduling the instructions for the functions units of the core.

The last step before the application can be compiled for simulation is to define the actor mapping file, which includes rules for distributing an actor-network over the architecture model. In TTADF, the actor mapping is static and developer defined, meaning that runtime changes for mapping are not possible. Based on the actor mapping, the TTADF compiler automatically maps FIFOs to memory components (private or shared). If there is no direct connection between two PEs, but an indirect connection exists, the designer can define data repeater actors to route data tokens to the desired PE through connecting PEs. Fig. 34 illustrates how an actor-network of the application (a) is mapped (c) for the triple TTA-core platform (b). Since there are no direct connections between TTA core 0 and TTA core 2, two data repeater actors (DR IQ1 and DR IQ1) are mapped to TTA core 1.

6.1.4 Analysis of the system configuration

In the analysis, the primary goal is to investigate how the current system configuration (application, architecture, and mapping) satisfies the implementation constraints (functionality, performance, power, area, etc). TTADF has three simulation models with different speed and accuracy for investigating the characteristics of the system configuration. The fastest C++ simulation model can be used to determine the correct functionality and approximate performance of the system. Cycle-accurate performance results can be achieved using the SystemC simulation model, which employs a more realistic memory model. The most accurate model is the RTL-level model, which can be refined by using hardware design flows in physical models to attain accurate estimates for area and energy. Since the simulation time increases with the accuracy of the simulation model, it is not suitable to keep hardware synthesis inside the design loop all the time. The designer should first therefore find promising system configurations that satisfy the performance constraints and after that proceed to investigate area and power.
6.1.5 Hardware synthesis and place and route

The RTL model can be refined towards physical implementation using the ASIC or FPGA design flow offered by commercial EDA tools. Hardware synthesis from RTL into a netlist of logic ports, and further to placing and routing cells and black box components onto die is essential to get an accurate performance, power, and area (PPA) estimate. This is emphasized even more in the case of high-end CMOS technologies, due to the increased significance of wire delays and wire loads to PPA [253]. The importance of the layout therefore becomes more pronounced. This is a challenge for automating DSE since the simulation of the physical models of wires and gates is time-consuming.

In Paper II, the pipelined triple core TTA design was placed and routed using 28 nm standard cell technology, and Fig. 35 shows a physical view of the chip. The figure illustrates the flow of data through a processing pipeline and underlines how memory components can be placed near to PEs in order to minimize delays.

6.2 Related work

Several different system design frameworks which facilitate dataflow-based application design are proposed in the literature and work of Park et al. [254] and Teich [20] give a comprehensive review of them. In the context of TTA-based multicore design, Jääskeläinen et al. [255] have proposed design flow symmetric TTA processors which are programmed using OpenCL for targeting task parallel workloads. In this section, the main focus is on the work that supports the dataflow process networks and targets solve task and pipeline parallel workloads using TTA-processors.
In [194] and [220], the authors present an automatic synthesis of TTA processor networks from dynamic dataflow programs using the Orcc framework. They propose a design flow where the RVC-CAL dataflow language is used to describe an actor-network. Orcc [106] compiles the actor-network into LLVM (Low Level Virtual Machine) [256] assembly code in the case of [220], or into C code in the case of [194]. In both works, TCE is then used to generate RTL descriptions of processor cores and the machine code for each core. Orcc is responsible for generating the top-level RTL description of a TTA processor network. In both works, a dedicated TTA processor is generated for each actor, and inter-processor communication is implemented via hardware FIFOs between TTA cores. Each input/output port of each actor is realized as an input/output stream FU to the processor which executes the actor. In [220], the authors define three TTA processor configurations: standard, custom and huge with different numbers of function units and transport buses.

Similarly, Yviquel et al. [182] refine the basic ideas presented in [220] by introducing a hybrid memory architecture designed for dataflow programs. Instead of using hardware FIFOs for inter-processor communication the authors exploit shared memories, which enable more flexible and copy-free communications. The shared memory based interconnection network between TTA processors is based on the actor-network of an application. As in [194], RVC-CAL is used as the input language, which is first transformed into an LLVM intermediate representation and then into binary code, which is suitable for the target TTA processor. In [182], Yviquel et al. demonstrate their work by implementing HEVC and MPEG-4 video decoders on top of custom, fast and huge TTA processors. Currently, their work is a part of Orcc [106], and it is called the Orcc TTA backend.

The design flow presented in Paper V and the Orcc TTA backend frameworks can mostly be used for the same purpose, but their design flows have a substantial difference. In the framework of Paper V, the designer separately specifies the system architecture (TTA core definition, core connections, and host connections), after which the dataflow application is mapped to the architecture. In contrast, in the Orcc TTA backend, the system architecture (TTA core interconnections) is derived from the dataflow application. From this viewpoint, the work of Paper V can be considered to be more generic. On the other hand, the Orcc TTA backend provides more automation due to the automatic interconnect generation and actor mapping features. Orcc also offers high-level dataflow analysis features which are currently not available in the framework of Paper V.
Another significant difference between Paper V and the Orcc TTA backend is the dataflow modelling language. Orcc’s TTA flow uses RVC-CAL language, and Paper V uses C-language combined with XML-meta modelling language.

In Orcc, FIFO sizes are forced to be powers of two, which enables optimized ring buffer implementation. On the other hand, this can cause a significant increase in memory requirements in some borderline cases. In the design flow of Paper V, the designer can select the token size and capacity of FIFOs freely for an optimal fit with the application requirements. This feature is essential when memory and energy constrained embedded systems are being considered.

The Orcc TTA backend simulation model is totally based on TCE. TCE has the assumption on the ideal memory which is always accessible at fixed latency. Therefore, Orcc generates multiport shared memories for simultaneous memory access of TTAs and does not offer support for memory arbiters. In the work of Paper V, TCE’s ideal memory model is overridden by automatically generated SystemC simulations, which allows cycle-accurate simulation of architectures that contain shared memories that are behind the memory arbiter.

The framework of Paper V enables map actors to the host processor, and in simulations, these actors are executed on the host system of the framework. This feature can be exploited in many ways. For example, the input and output data of the TTA co-processing system can be written and read directly from the memory of TTAs using actors mapped to the host, and there is no need for simulation-specific TTA special function units, which generate input and write output data, as is the case with the Orcc TTA backend. Testing of an individual actor on a particular TTA core is easy and fast since test data for the actor is created by other actors on the application at runtime.

### 6.3 Experiments

The design flow of Paper V was tested using three different applications. The HEVC In-loop filter was translated from standard C implementation (Paper I) to TTADF dataflow format. From the computer vision domain, the Stereo Depth Estimation (SDE) algorithm was ported from OpenCV to TTADF dataflow format. The wireless communication application Dynamic Predistortion Filter (DPD) was ported to TTADF format from RVC-CAL descriptions by exploiting Orcc generated C-code.

Table 5 compares the different programmable implementations of the three applications. The main findings of the experiments were:
Table 5. Comparison of different programmable implementations of the test case applications. © 2019 IEEE.

<table>
<thead>
<tr>
<th>Appl.</th>
<th>Architecture</th>
<th>Tool</th>
<th>Tech. (nm)</th>
<th>Ckf (MHz)</th>
<th>Perf.</th>
<th>Power (mW)</th>
<th>Energy Eff. (mJ/Perf.unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inloop</td>
<td>Inloop TTA × 3 (Paper II)</td>
<td>None</td>
<td>28</td>
<td>1200</td>
<td>153 fps</td>
<td>207</td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td>Inloop3 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>148 fps</td>
<td>211 ²</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>Shared3 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1000</td>
<td>24.0 fps</td>
<td>154</td>
<td>6.42</td>
</tr>
<tr>
<td></td>
<td>[182] Fast TTA × 12</td>
<td>Orcc</td>
<td>40</td>
<td>1000</td>
<td>14.7 fps ¹</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Shared12 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>93.1 fps</td>
<td>617</td>
<td>6.63</td>
</tr>
<tr>
<td></td>
<td>Inloop12 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>536 fps</td>
<td>843 ²</td>
<td>1.57</td>
</tr>
<tr>
<td>SDE</td>
<td>[89] OpenCL SIMD TTA</td>
<td>OpenCL</td>
<td>28</td>
<td>800</td>
<td>117 Mde/s ³</td>
<td>33</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>[89] Intel Core i5-480M</td>
<td>OpenCL</td>
<td>32</td>
<td>2600</td>
<td>30.3 Mde/s ³</td>
<td>35000</td>
<td>1155</td>
</tr>
<tr>
<td></td>
<td>[89] Qualcomm Adreno 330</td>
<td>OpenCL</td>
<td>28</td>
<td>578</td>
<td>99.1 Mde/s ³</td>
<td>1800</td>
<td>18.16</td>
</tr>
<tr>
<td></td>
<td>Inloop1 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>14.9 Mde/s</td>
<td>69 ²</td>
<td>4.63</td>
</tr>
<tr>
<td></td>
<td>Inloop3 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>44.8 Mde/s</td>
<td>211 ²</td>
<td>4.71</td>
</tr>
<tr>
<td></td>
<td>Inloop12 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>152 Mde/s</td>
<td>843 ²</td>
<td>5.55</td>
</tr>
<tr>
<td></td>
<td>Odroid-XU3 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>2000</td>
<td>82.1 Mde/s</td>
<td>5861</td>
<td>71.4</td>
</tr>
<tr>
<td>DPD</td>
<td>Shared3</td>
<td>Orcc</td>
<td>28</td>
<td>1000</td>
<td>1.75 Ms/s</td>
<td>154</td>
<td>88.0</td>
</tr>
<tr>
<td></td>
<td>Shared6</td>
<td>Orcc</td>
<td>28</td>
<td>1000</td>
<td>2.38 Ms/s</td>
<td>309</td>
<td>135.5</td>
</tr>
<tr>
<td></td>
<td>Shared12</td>
<td>Orcc</td>
<td>28</td>
<td>1000</td>
<td>3.31 Ms/s</td>
<td>617</td>
<td>248.8</td>
</tr>
<tr>
<td></td>
<td>Shared6 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1000</td>
<td>5.2 Ms/s</td>
<td>309</td>
<td>59.42</td>
</tr>
<tr>
<td></td>
<td>Inloop6 (Paper V)</td>
<td>TTADF</td>
<td>28</td>
<td>1200</td>
<td>7.46 Ms/s</td>
<td>422 ²</td>
<td>56.57</td>
</tr>
</tbody>
</table>

¹Estimated assuming that the share of inloop filtering is 22% [182] of total HEVC decoding workload
²Estimate based on Paper II, ³16-bit floating point, no sobel filtering and uniqueness thresholding
– The TTADF implementation was on average 2× faster than the Orcc TTA backend implementation when both of the implementations were based on the same Shared-TTA core and the code is generated from the same RVC-CAL description.

– Exploitation of special function units gives a substantial competitive advantage to TTADF. The In-loop-core tailored from HEVC In-loop filtering is about 6× faster than the Shared-TTA core in a case of both implementation uses of TTADF flow. When compared to RVC-CAL HEVC In-loop filter design using the 12 Fast – TTA core, the data and task parallel based TTADF implementation, Inloop12 is about 36× faster.

– Using the TTADF framework for design In-loop-filtering solutions introduces only a 3% overhead (Inloop3) when compared to the fully manual C-only design presented in Paper I.

– Exploiting the pipeline parallelism of the in-loop filter gives a 2× speed up when compared to single core and three core pipelined architecture.

– The positive results regarding the parallel scalability of hybrid memory architecture presented in [182] are confirmed. Paper I shows that in single core cases, where all actors are mapped to the same core, actor scheduling overhead and conservative loop unrolling (due to program memory limits) decrease throughput with a consequence that speedups can become superlinear.

– The energy efficiency of TTA-cores is over a magnitude better than ARM-based GPP cores, in those cases where there is no architecture-specific code optimization used (Stereo Depth Estimation application).

Energy efficiency

Energy efficiency is always highly dependent on various design choices like manufacturing process technology, operating voltage, clock frequency and architectural design (e.g. design size, number of logic gates, memories, etc). The TTA as a fundamental design block has a significant contribution in achieving energy-efficiency in the design flow. However, when energy efficiency is also considered, software design and the multicore interconnect have an essential role.

It is claimed that software and hardware design should be tightly linked in order to achieve energy efficient solutions. To this end, justifications are presented (in addition to the TTA architecture itself), as to why the proposed design flow is suitable for designing energy efficient solutions:
In TTADF, the designer can directly exploit the custom operations (in special function units, SFUs) offered by the underlying TTA architecture, with calls from software code. Custom instructions can considerably contribute to energy efficient design (e.g., by allowing reduction of clock frequency and even operating voltage, while still meeting performance requirements). The benefits of this can be seen in the HEVC In-loop Filtering application where the processor cores are equipped with SFUs that are heavily utilized by the software code.

TTADF introduces SFUs for FIFO-specific communication operations, and the compiler detects automatically if these SFUs are present and uses them if available. By using hardware accelerated FIFO operations, accessing the FIFO data structure is faster, but even more importantly, it can significantly reduce the required program memory size, which has a direct impact on the power consumption. The SFU for FIFO operations is utilized in all applications, but in the case of the DPD application, a notable decrease in program memory size can be observed. This is because the DPD source code contains many FIFO operation calls inside loops that are unrolled by the compiler.

In TTADF, FIFO operations are designed so that the designer can arbitrarily choose the size of the FIFO buffers. Thus, the designer can optimize the size of the shared and data memories and, again, reduce memory power consumption. With all three applications presented in Paper V, the software has been designed so that on-chip SRAM memories suffice for the whole application, instead of requiring power-hungry off-chip DDR memories.

In TTADF, a hybrid memory architecture is used, where each TTA core has a separate private data memory and private instruction memory. Inter-core communication is performed through shared memories that form a communication network (possible respecting application data flow) between cores. The memory organization divides memory components into small subcomponents, which reduces memory pressure, provides simultaneous R/W access (program, private, shared) and reduces power consumption when compared to the large global shared memories used in many other platforms.

Cost efficiency

It is difficult to give any exact figures on how much the design flow presented in Paper V reduces the design time of multicore systems since these things are always highly
relative to the designer itself. However, if compared to the time consumed in designing the system presented in Paper I, the time consumed in designing three applications for eight different platform configurations in Paper V, it can be said that the design flow reduces design time significantly and enables fast design space exploration.

The design of TTA cores using TCE tools is fast, and the learning curve of tools is low even for newcomers to processor design. This judgment is based on the fact that TCE tools have been taught in undergraduate processor design courses at several different universities.

Using the TTAs as core components of the design flow gives flexibility that reduces the design risks and product lifetime. Paper I, Paper II and Paper V show that TTAs can be designed to be suitable for different types of signal processing applications. As a consequence, TTAs can be designed to have better area efficiency than ASICs which implement the same functionality. Moreover, it is worth mentioning that the exposed datapath of TTAs enables software fixes for timing related hardware bugs which are noticed after the chip manufacturing process. This is a highly desirable feature due to the high design cost of current manufacturing technologies.

6.4 Conclusions and future work

The design flow of Paper V offers a way to raise the abstraction level of multicore co-design with negligible impact on performance. The flow addresses the problem of task and pipeline parallel workloads on TTA-based processing elements. The experimental results suggest that the proposed design flow can outperform the current state of the art, the Orcc TTA backend, by a clear margin regarding performance. Especially the possibility of exploiting special function units gives a substantial competitive advantage for the work of Paper V over the Orcc TTA backend. Allowing this kind of direct code optimization should be one of the primary development targets for Orcc because the current capabilities of the compilers are not sufficient to efficiently facilitate special instructions. This can be seen to be a serious issue that slows down the shift from low-level to higher level descriptions. If the high level abstractions are used it is important that a framework not limit the designer possibilities to optimize application.

In the future, it is worthwhile to investigate transforming the RVC-CAL actor networks to TTADF format. In addition, automatic mapping features should be adopted for the proposed framework. This would include mapping of the actors to cores, an automatic communication route explorer and the creation of corresponding repeater
actors. For power savings, approaches similar to the notifying memories [257] could be adopted into the proposed framework. Adopting RISC-V toolchains the part of TTADF, would offer possibility to design open-source and extensible a host processors for TTA-based co-processing systems.
Summary and conclusions

This thesis has made contributions throughout the design flow starting from a dataflow description of the application and ending on the implementation of application-specific processors.

The frequently changing functionality of systems and increasing chip manufacturing costs have created a demand for highly reusable platforms which are software programmable, but still without compromising too much area and energy efficiency. This has made application specific processors an appealing candidate for the building blocks of future increasingly heterogeneous SoCs, since their high performance, low power consumption, and automated SW/HW co-design flows can provide a compromise in design costs between software-based GPP solutions and fixed functioning HW solutions.

In this thesis, two customized, but still multipurpose and fully programmable, transport triggered architecture cores were proposed for HEVC In-loop filters. The cores were designed using a rapid application specific processor design tool-chain, TTA-based Co-Design Environment (TCE). The experiments provided by the thesis showed that the proposed solutions form a new design option for HEVC implementations alongside conventional embedded GPPs and ASIC in terms of energy efficiency and programmability. Energy efficiency of implementations makes them suitable design options for small form factor embedded devices with a power budget under 1.5 W.

Interconnection networks and memory architectures have a significant role when considering performance, power consumption, and parallel scalability. One way to improve energy efficiency and scalability is to distribute the application into several rather small customized PEs, with their own small and fast on-chip memories. The PEs are connected respecting the data flow of the applications for enabling parallel scalability by reducing memory congestion and access latencies. The disadvantage of the approach is that software design becomes more complicated since the designer has to orchestrate rather small chunks of data through parallel and heterogeneous PEs with various memory address spaces. However, the problem can be mitigated by utilizing automated high-level model based design tools.

The thesis proposed a design flow which integrates ASP development and dataflow-based programming for enabling rapid experimentation and prototyping of high performance and energy-efficient heterogeneous multi-core systems. The design flow allows
fast adaption of new dynamic dataflow applications for tailored multicore systems. The critical enabler for this is the use of formal models which allow automated software synthesis from a single application model for different multicore platforms.

To conclude, the solutions which meet performance, energy, and design cost requirements of future applications are likely to have the following properties.

**Heterogeneous and software-defined SoCs**

Currently, multiprocessor System-on-Chips (MPSoC) are mainstream in mobile devices [157]. MPSoCs can integrate tens or hundreds of IP blocks inside the same chip, including different processing elements such as GPU, FPGA, ASIC or ASP, in addition to the traditional general purpose cores. It is expected that heterogeneity will only increase in the future [18]. Due to inflexibility and the skyrocketing design costs of ASICs and the compromised performance of standard processors, ASPs have become an attractive design option for SoCs, which have to answer the demands of feature-rich products. Usually, an instruction set of an ASP is tailored for the desired application domain to achieve better energy efficiency. That is, critical parts of an application can be accelerated using specialized hardware, and extra functionality can be excluded to simplify the architecture. The programmability of the ASPs offers a time-to-market advantage over the ASICs, since software development is faster than hardware development. Also, the flexibility enables the start of product development during an in early phase, before the algorithms (e.g., video standard or wireless radio standards) are frozen.

**Multi-level parallel processing**

A high degree of parallel processing at different granularities is crucial if high energy efficiency is needed. Fine grain parallelism such as instruction level parallelism can take advantage of a single thread by issuing multiple operations simultaneously. Advanced compiler techniques have been enabled exploiting of ILP, also in small accelerators since complex runtime multi-issue instruction schedulers can be replaced by extracting the parallel instructions at compile time [187]. Exploiting Data level parallelism using SIMD vector processing resources is widely used, and is a very effective way of improving processor performance. Task-level parallelism refers to coarse grain granularity method multiple task or processes that are executed concurrently in different processing cores.
Frequency scaling of processors slowed down after the turn of the 2000s, and the exploiting of the TLP were seen as methods to address the issue by increasing the number of parallel cores to cut down voltages and frequency. This resulted in the growth of the number of cores on the chips, and the end of the trend is not yet in sight.

**Model-driven system development tools**

Developing complex, high-performance, embedded applications for heterogeneous and parallel computer architectures is becoming increasingly demanding and is a time consuming task without suitable tool-chains. Many of current mainstream tool-chains are tedious and difficult to use, since they are operating at too low-level. Model-driven system design methods are likely to be needed to manage the system design process of the next generation complex and massively parallel applications, and at the same time to reduce the design costs of the application. Already the growth of heterogeneity and parallelism has induced a *software productivity gap* [258] due to traditional programming paradigms have failed to respond to the demands of the new applications. Consequently, the pressure for a paradigm shift to a higher level of abstraction has increased. Raised abstraction levels allows optimization at higher level which often enables better gains when compared to low level optimization.
References


133


135


141


Available: https://doi.org/10.1007/s11554-015-0519-1

L. Sousa, “GPU Parallelization of HEVC In-Loop Filters,” International Journal of
https://doi.org/10.1007/s10766-017-0488-z

in embedded systems,” in 2015 International Conference on Embedded Computer Systems:

http://www.verisilicon.com/IPPortfolio_14_98_2_HantroG2.html, 2018, “[Online;
accessed 17/1/2019]”.


8K Ultra HD applications,” in 2016 IEEE International Solid-State Circuits Conference
(ISCC), Jan 2016, pp. 266–268.

hevc decoder on an fpga,” IEEE Transactions on Circuits and Systems for Video Technology,

hd real-time hevc main profile decoder,” IEEE Transactions on Consumer Electronics,

estimation,” in 2007 Ph.D Research in Microelectronics and Electronics Conference, July

specific instruction-set processor,” IEEE Transactions on Circuits and Systems for Video Technology,

https://doi.org/10.2478/s11772-013-0071-0

Itegem, D. Amirtharaj, K. Kalra, P. Rodriguez, and H. van Antwerpen, “The tm3270 media-
processor,” in 38th Annual IEEE/ACM International Symposium on Microarchitecture

binary arithmetic decoding on transport triggered architecture,” vol. 6821, 2008. [Online].
Available: https://doi.org/10.1117/12.772019

[216] V. Magoulianitis and I. Katsavounidis, “Hevc decoder optimization in low power config-
urable architecture for wireless devices,” in 2015 IEEE 16th International Symposium on A
World of Wireless, Mobile and Multimedia Networks (WoWMoM), June 2015, pp. 1–6.


Original publications


Reprinted with permission from IEEE (I, II, III, V) and Springer (IV).

Original publications are not included in the electronic version of the dissertation.
699. Liu, Xin (2019) Human motion detection and gesture recognition using computer vision methods
708. Shahabuddin, Shahrir (2019) MIMO detection and precoding architectures

Book orders:
Granum: Virtual book store
http://granum.uta.fi/granum/
Ilkka Hautala

FROM DATAFLOW MODELS TO ENERGY EFFICIENT APPLICATION SPECIFIC PROCESSORS