FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Virtual HW/SW Prototyping for Design and Runtime Prediction of Parallel Video Coding Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Florian Seitner
Matrikelnummer 9925654

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: ao. Univ.-Prof. Mag. Dipl.-Ing. Dr.techn. Margrit Gelautz

Diese Dissertation haben begutachtet:

(ao. Univ.-Prof. Mag. Dipl.-Ing. Dr.techn. Margrit Gelautz)	(Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard Rinner)

Wien, 10. Oktober 2013

(Florian Seitner)

Erklärung zur Verfassung der Arbeit

Florian Seitner
Hietzinger Hauptstraße 56/1/8, 1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

First of all, I want to thank my supervisor Margrit Gelautz for her constant support and encouragement throughout the years that I spent as a PhD student. She forged my skills in research and scientific writing and without her guidance, this work would not have been possible. I am grateful to Bernhard Rinner who kindly agreed to be the second reviewer of this thesis.

During my studies, I also had the pleasure to work with Michael Bleyer. His feedback and countless hours of joint work before various submission deadlines proved to be of enormous value for my research. I would like to thank Ralf Beuschel for introducing me to the field of video coding and showing me the importance of structured scientific working in an industrial environment. I would also like to acknowledge Tom Wilson for proofreading this thesis and for supporting me in finding time to finish this thesis.

Many thanks go to all of my colleagues in the Interactive Media Systems Group and at OnDemand Microelectronics who offered me the opportunity for many interesting discussions and a nice and productive working atmosphere.

I would like to thank the Austrian Federal Ministry of Transport, Innovation, and Technology (BMVIT) for its financial support of this thesis under the FIT-IT project VENDOR (Project no. 812429).

I would further like to express my thankfulness to my friends. (I do not put any names, since I am afraid to forget someone.) Their support and the ability to enjoy my life outside the pages of my thesis were always a great source of inspiration. I would also like to thank my family, especially my mother and father, who made it possible for me to study.

Finally, special thanks go to my beloved wife Julia for her support and patience during all these years. A thesis is like a marathon and she gave me support from the first to the last kilometer.

Abstract

The high computational demands of state-of-the-art video coding standards such as H.264 pose serious challenges on embedded processor architectures. A natural way to tackle this problem is the use of multi-processor systems. However, the efficient distribution of complex video coding algorithms among multiple processing units (PUs) is a non-trivial task. In order to use the available processing resources efficiently, an equally balanced distribution of the coding algorithm onto the hardware units must be found. The system designer has to consider data-dependency issues as well as inter-communication and synchronization between the PUs. Furthermore, efficient software design is necessary in order to satisfy the resource limitations in an embedded environment, such as low computational power, small-sized on-chip memories and low bus bandwidth. A parallel video coding implementation for an embedded system must be able to work under these resource restrictions.

Being able to predict the resource requirements of a parallel video coding application (VCA) is therefore essential during the design of a video coding system (VCS) considering these strict requirements on runtime performance and resource usage. This thesis contributes novel methods to support the complex design process of parallel VCS in an early phase of system design when highly critical decisions on hardware and software are made. The contributions of this thesis can be summarised as follows. (i) We propose the *Data-Driven Profiling* (DDP) method for analysing and visualizing the runtime complexity of a VCS. This method maps traditional runtime profilings onto the coding elements and functional blocks of a video coding algorithm. It enables the system designer to relate runtime complexity with the application levels where parallelisation takes place and introduces means for analysing the workload distribution. (ii) We demonstrate how to exploit DDPs for analysing complexity and deriving essential information for parallel system design. Assumptions about the performance of a VCA on a parallel architecture can be made, potential problems in work balancing identified and complexity variations in the functional blocks of a VCA's video coding elements analysed. (iii) We introduce the Partition Assessment Simulation (PAS) methodology for enabling the exploration of complex parallel VCS designs. This methodology exploits the structural and functional similarities of modern video coding algorithms for predicting a VCA's runtime on a "virtual" architecture. (iv) We implement a simulator for the PAS concept. By modelling and simulating an existing multi-processor platform, the PAS methodology is verified. We demonstrate the flexibility of the PAS to simulate complex parallel video coding platforms and to explore new parallel designs for functional as well as data-parallel H.264 decoder partitioning methods. We believe that the contributed techniques enable system designers to address the challenges of parallel VCS design in an intuitive and time-efficient way leading to application-tailored and cost-competitive VCS.

Kurzfassung

Die hohen Anforderungen, die moderne Videokodierstandards an die Rechenleistung stellen, können auf vielen Embedded-Architekturen nicht oder nur eingeschränkt gelöst werden. Der Einsatz von Multi-Prozessorsystemen und die Aufteilung der Kodierung auf mehrere Prozessoren bieten hier eine elegante Lösung. Der Entwurf von parallelen Videokodiersystemen stellt jedoch bei komplexen Algorithmen wie H.264 eine herausfordernde Aufgabe dar. Es gilt hier, eine gleichmäßige Aufteilung der Rechenschritte auf die verfügbaren Prozessoren zu finden und dabei bei der Partitionierung die hohe Anzahl an algorithmischen Abhängigkeiten zwischen den einzelnen Schritten zu berücksichtigen. Des Weiteren müssen architekturbedingte Ressourcenlimits wie z. B. die Speichergröße berücksichtigt werden.

Diese Arbeit widmet sich der Performanceanalyse und -vorhersage von parallelen Videokodiersystemen. Der wissenschaftliche Beitrag dieser Arbeit umfasst zwei Methoden, um die Laufzeit von Videokodieralgorithmen effizient zu analysieren und bereits in früheren Phasen des Designprozesses Annahmen über die Eigenschaft des Gesamtsystems treffen zu können. Die erste Methode, das Data-Driven Profiling (DDP), ermöglicht es, die Laufzeit eines Videokodiersystems im Zusammenhang mit den zu verarbeitenden Daten zu analysieren. Dabei werden traditionelle Laufzeitprofile automatisch auf die Kodierelemente und -schritte des Kodieralgorithmus abgebildet. DDP gibt Aufschluss über die Laufzeit, die für die Kodierung einzelner Kodierelemente und funktionaler Kodierblöcke aufgewendet wird und wie diese das Laufzeitverhalten von parallelen Videokodiersystemen beeinflussen. Die zweite Methode, die Partition Assessment Simulation (PAS), macht sich strukturelle und funktionale Charakteristika hybrider Videokodieralgorithmen zunutze, um Laufzeitabschätzungen für virtuelle Architekturen zur Videokodierung zu treffen. Diese Methode baut auf DDP sowie Konzepten der simulationsbasierten Laufzeitvorhersage auf und ermöglicht bereits in einer frühen Phase der Systementwicklung das Ausprobieren unterschiedlicher Designvarianten und das schnelle Adaptieren von parallelen Videokodiersystemen an Designvorgaben. Diese Arbeit beschreibt eine konkrete Implementierung für das PAS Konzept und liefert mit Hilfe einer bestehenden Multiprozessorarchitektur eine Verifikation und Genauigkeitsanalyse. Die Flexibilität, neue Designmöglichkeiten zu erschließen, wird anhand konkreter Beispiele demonstriert.

Die vorgestellten Techniken ermöglichen es, beim Design von parallelen Videokodiersystemen gezielt und anwendungsspezifisch auf Komplexität und benötigte Hardwareresourcen einzugehen. Bereits in einer frühen Phase des Designprozesses können Abschätzungen über das Laufzeitverhalten des Designs gemacht und dadurch das Entwicklungsrisiko signifikant gesenkt werden.

Contents

List of Figures **xiii**

List of Tables **xv**

List of Abbreviations **xviii**

1 Introduction **1**
1.1 Design of parallel video coding architectures 1
1.2 Motivation and objectives 2
1.3 Contributions 3
1.4 Resulting publications 4
1.5 Organization 5

2 Prior work on complexity and runtime estimation **7**
2.1 Analytic runtime prediction 7
2.2 Runtime prediction based on dynamic profiling 8
2.2.1 Statistical profiling 10
2.2.2 Instrumented profiling 10
2.3 Simulation-based runtime prediction 13
2.3.1 Hardware simulation techniques 13
2.3.2 Instruction set simulation 13
2.3.3 HW/SW-codesign 14
2.4 High-level design exploration 15
2.5 Partition Assessment Simulation in context of prior work 15
2.6 Summary 16

3 Characteristics of modern video coding algorithms **17**
3.1 Historical development of digital video coding 17
3.2 Concept of hybrid video coding 19
3.3 Hierarchical structuring of video coding elements 21
3.4 Coding tools 22
3.4.1 Spatial prediction 22
3.4.2 Motion-compensated prediction 23
3.4.3 Transformation and quantisation of residual data 27

3.4.4 Deblocking filter . . . 27
3.4.5 Entropy coding . . . 28
3.5 Parallel video decoding . . . 28
3.5.1 Dependencies between macroblocks . . . 29
3.5.2 Functional partitioning . . . 30
3.5.3 Data-parallel partitioning . . . 31
3.6 Summary . . . 36

4 Data-driven runtime analysis **39**
4.1 Data-driven profiling . . . 39
4.2 Automatic generation of data-driven profiles . . . 41
4.2.1 Finite State Machines and Pushdown Automatons . . . 42
4.2.2 Mapping profiling information to VCL and functional blocks . . . 43
4.2.3 Extraction of coding information via function names . . . 45
4.2.4 Extraction of coding information via instrumentation . . . 46
4.2.5 Implementation . . . 46
4.3 Profiling environment and test sequences . . . 47
4.3.1 Reference architecture . . . 47
4.3.2 Test sequences . . . 50
4.4 Experimental results for runtime analysis and visualization . . . 55
4.4.1 Complexity of processing VCL coding elements . . . 55
4.4.2 Complexity of processing functional blocks . . . 58
4.4.3 Analysing complexity within individual subregions of a frame . . . 60
4.5 Summary . . . 62

5 Virtual prototyping of parallel video coding systems **63**
5.1 General aspects and design goals . . . 63
5.2 Concept . . . 64
5.2.1 System specification . . . 65
5.2.2 Characterisation . . . 72
5.2.3 Simulation . . . 73
5.3 Implementation of the Partition Assessment Simulation . . . 75
5.3.1 Time domains within PAS . . . 75
5.3.2 Task generation based on data-driven profiling . . . 76
5.3.3 Rule-based specification of data-dependencies . . . 76
5.3.4 Partitioning of video coding application . . . 77
5.3.5 Simulation process . . . 78
5.4 Summary . . . 81

6 Concept verification and design space exploration results **83**
6.1 Specification of a dual-core video coding system . . . 83
6.2 Characterisation of virtual hardware . . . 84
6.3 Verification using a functional dual-core decoder splitting . . . 87
6.4 Design space exploration . . . 91

6.4.1 Functional partitioning 91
6.4.2 Data-parallel partitioning 93
6.4.3 Alternative processor for parsing 93
6.5 Summary 94

7 Conclusions and future work **95**
7.1 Conclusions 95
7.1.1 Analysis of VCA runtime behaviour 95
7.1.2 Modelling and simulation of virtual architectures 96
7.2 Open topics for future research 97

A Detailed description of test sequences **99**

Bibliography **105**

List of Figures

2.1 A simple control flow graph (CFG). 9

3.1 Historical development of international digital video coding standards. 18
3.2 H.264 encoder and decoder structure. 20
3.3 Hierarchical structuring of a video stream in the H.264 standard. 22
3.4 H.264 intra-prediction modes. 23
3.5 Temporal prediction of macroblocks between frames. 24
3.6 H.264 inter prediction macroblock partitioning. 24
3.7 GOP-Coding. 26
3.8 Visualisation of block edge deblocking in H.264. 28
3.9 Macroblock dependencies in H.264 decoding 29
3.10 Functional split of an H.264 decoder. 30
3.11 The Single-row splitting approach. 31
3.12 Example of the Single-row splitting approach used with two cores 31
3.13 Inter-processor dependencies in a multi-core system 33
3.14 The Multi-column splitting approach. 33
3.15 Example of the Multi-column splitting approach 33
3.16 The Slice-parallel splitting approach 34
3.17 Example of the Slice-parallel splitting approach in the blocking version 34
3.18 Example of the Slice-parallel splitting approach in the non-blocking version 34
3.19 The Diagonal splitting approach 36
3.20 Example of the Diagonal splitting approach 36
3.21 Dependencies in the Diagonal and Multi-column splitting approaches 36

4.1 Data-driven profiling at macroblock level 41
4.2 Example of a state transition diagram. 42
4.3 Example of a state transition diagram for an H.264 decoder. 44
4.4 Structure of the SVENm architecture. 48
4.5 Floorplan and board of the SVENm architecture. 49
4.6 Visualization of test sequences used in work. 51
4.7 GOP-Coding of a sequence with 25 frames. 52
4.8 Bitrates of the 16 test sequences coded at a Y-PSNR of 40 db. 53
4.9 Dynamic variations in the decoding time of individual macroblocks. 57

4.10 Dynamic variations in the runtime of H.264's functional blocks. 59
4.11 Visualization of runtime complexity for the individual MBs of I/P/B-frames. 61

5.1 The Partition Assessment Simulation (PAS) 64
5.2 System specification in the PAS 66
5.3 Simple dependency graph for two macroblocks' decoding tasks. 68
5.4 Sequential task order for a macroblock's decoding tasks. 69
5.5 Mapping of VCA graph onto hardware. 71
5.6 Algorithm for simulating parallel task execution in a VCA 74
5.7 Functional partitioning of four macroblocks. 79
5.8 Visualisation of the internal simulation process in the PAS. 80

6.1 Relative and absolute runtime differences when calibrating the PAS. 85
6.2 Relative runtime differences in percent during PAS calibration. 86
6.3 Absolute runtime differences in clock cycles during PAS calibration. 87
6.4 Absolute runtime differences during PAS verification. 88
6.5 Relative runtime differences during PAS verification. 89
6.6 Absolute runtime differences during PAS verification. 90
6.7 Runtime of the simulated decoder partitioning approaches. 92

List of Tables

2.1 Example of dynamic function trace profiling. 11

4.1 Profile for the tasks of the decoder's individual functional blocks. 42
4.2 Transition table of a state machine. 43
4.3 Macroblock-based H.264 profile information. 45
4.4 Detailed size and quantisation values for the test sequences' frames. 54
4.5 Complexity dynamics during the decoder's runtime. 56

A.1 Test sequences 1 to 4. 100
A.2 Test sequences 5 to 8. 101
A.3 Test sequences 9 to 12. 102
A.4 Test sequences 13 to 16. 103

Abbreviations

AI	Automatic Instrumentation
ASLI	Automatic Source Level Instrumentation
BB	Basic Block
CABAC	Context Adaptive Binary Arithmetic Coding
CAVLC	Context Adaptive Variable-Length Coding
CFG	Control Flow Graph
DAG	Directed Acylic Graph
dB	decibel
DCC	Display Content Controller
DCT	Discrete Cosine Transform
DDP	Data-Driven Profiling
DDPL	Data-Driven Profiling Library
DMA	Direct Memory Access
DPCM	Differential Pulse Code Modulation
DR	Dependency Rules
FAF	FIFO Assignment Function
FB	Functional Block
FIFO	First-In-First-Out buffer
fps	frames per second
FSM	Finite-state Machine
GOP	Group Of Pictures
HVS	Human Visual System
HW	Hardware
ICACHE	Instruction Cache
IDCT	Inverse Discrete Cosine Transform
IEC	International Electrotechnical Commission
IP	Instrumented Profiling
ISO	International Organization for Standardization
ISS	Instruction-Set Simulator
ITU	International Telecommunication Union
ITU-T	International Telecommunication Union - Telecommunication
kB	kilobyte
LoP	Level of Parallelisation
MAF	Memory Access Function

MB ..... Macroblock
MCP ..... Motion-compensated prediction
mDDR ..... Mobible Double Data Rate Memory
MoE ..... Model of Execution
MSE ..... Mean Square Error
MV ..... Motion Vector
NAL ..... Network Abstraction Layer
OoE ..... Order of Execution
PAF ..... Processor Assignment Function
PAS ..... Partition Assessment Simulation
PDA ..... Pushdown Automaton
PSNR ..... Peak Signal-to-Noise Ratio
PU ..... Processing Unit
QP ..... Quantisation Parameter
RISC ..... Reduced Instruction Set Computer
SBRP ..... Simulation-based Runtime Prediction
SIMD ..... Single Instruction Multiple Data
SIT ..... Software Instrumentation Tool
SoC ..... System-on-Chip
SRAM ..... Shared Random-access Memory
STO ..... Sequential Task Order
SW ..... Software
TS ..... Transport Stream
UID ..... Unique Identifier
VA ..... Virtual Architecture
VCA ..... Video Coding Application
VCL ..... Video Coding Layer
VCS ..... Video Coding System
VHDL ..... Very High Speed Integrated Circuit Hardware Description Language
VHSIC ..... Very High Speed Integrated Circuit
VLIW ..... Very Long Instruction Word
VSD ..... Virtual System Definition
Y-PSNR ..... Peak Signal-to-Noise Ratio of the luma channel

CHAPTER 1

Introduction

1.1 Design of parallel video coding architectures

State-of-the-art video standards such as H.264 [ITU12] are used in a wide range of industrial and consumer applications. This includes for example digital television broadcasting, video surveillance and video conferencing. Compared to preceding video coding standards such as MPEG-2 and MPEG-4 SP/ASP, improved coding efficiency could be reached by introducing more advanced pixel processing algorithms (e.g. quarter-pixel motion estimation, integer-based block transforms) as well as by the use of more sophisticated algorithms for predicting syntax elements from neighbouring macroblocks (e.g. context-adaptive variable-length coding). These new coding tools result in significantly increased CPU and memory loads required for coding a video stream. In environments of limited processing power such as embedded systems, the high computational demands pose a challenge for practical video coding implementations [FG01]. Multi-core System-on-Chip (SoC) design provides an elegant solution to overcome these performance limitations.

A SoC design combines multiple components such as processors and memories on a single chip. The usage of existing and well-tested components can reduce the costs and the developing time and results in a short time-to-market. The programmability of most SoCs allows later modifications of the algorithm's software which offers high flexibility and is of prime importance for video coding. For example, for adapting the software when a new extension of a video standard becomes available or to run different video coding algorithms on the same platform.

The high computational demands of state-of-the-art video coding application (VCAs) pose serious challenges on current SoC architectures. A natural way to tackle this problem is the use of multi-core systems. However, the efficient distribution of video coding algorithms among multiple processing units (PUs) is a non-trivial task. For using the available processing resources efficiently, an equally balanced distribution of the coding tasks onto the hardware units must be found. The system designer has to consider data dependency issues as well as intercommunication and synchronisation between the PUs. Furthermore, the resource limitations in an embedded environment such as low computational power, small-sized memories and low bus

bandwidth require an efficient software design. A parallel VCA approach must be able to work under these resource restrictions.

1.2 Motivation and objectives

A major source for uncertainty within most VCA designs is that the software development and partitioning is typically addressed at a late phase of the SoC design. At this stage, significant resources already went into the system's hardware design and integration and changes are expensive and only possible in a limited scope. However, from what has been outlined above, multiple questions arise already at early design phases of a multi-core video processing system when the components of the SoC design are chosen:

1. Can we reach the performance requirements of the VCA on the available hardware?
2. What hardware is required by the VCA to handle a specific set of video streams?
3. What is the optimal VCA partitioning for using the architecture's resources most efficiently?

In previous work, the first two questions have been widely discussed for single-core architectures. Various complexity estimation and runtime prediction techniques with their respective advantages and disadvantages have been proposed. We describe existing complexity and runtime estimation techniques and their strengths and weaknesses in detail in Section 2. However, the existing profiling techniques for single-core VCAs are only suitable to a limited extent for addressing multi-core SoC design for parallel video coding applications in an efficient way. They typically provide coarse profiling information on the larger functional blocks (e.g. the absolute runtime spent in a decoding function and all sub-functions) but do not provide means for efficiently analysing the runtime complexity of an VCA at the level where the parallelisation would be implemented. This makes it hard to exploit the available profiling information when making predictions about the single-core VCA runtime performance on a multi-core architecture.

Addressing this weakness of existing profiling techniques is one major objective of this thesis. We provide a method to efficiently derive runtime information from single-core VCAs that typically affects the system designers choice of parallelisation. For example, detailed execution times for the video coding elements that are processed during the VCA execution and runtime variation of individual functional blocks are provided. We introduce a new complexity analysis technique that equips system designers with a toolset to extract runtime information at a level where parallelisation will take place later on. We extend traditional complexity estimation and profiling techniques in a way that enables more detailed analysis of a VCA's parallel execution behaviour. A special focus shall be given to the hierarchical data structures and functional blocks of VCAs. They determine where parallelisation mechanisms can be integrated within the VCA and must be considered carefully when analysing parallel execution behaviour.

The third question above introduces the need for predicting the runtime of "virtual" VCA partitionings, which typically is not accurately possible with traditional analytical models or too time- and labor-intensive using existing hardware simulation techniques. Prediction techniques

that base their runtime estimation on formal algorithm definitions (i.e. analytical runtime prediction techniques) or runtime observations from single-core implementations typically cannot consider parallel task concurrency and inter-task dependencies appropriately when estimating the complexity of a VCA in a parallel architecture configuration. More powerful prediction techniques that can simulate the hardware and software execution for virtual platforms typically require an already partitioned VCA implementation. Estimating the runtime performance of a parallel VCA's software partitioning without starting the labor-intensive implementation work is not possible. Furthermore, many simulation techniques require circuit-based hardware simulation which is very time-intensive. These limitations due to (i) labor- and time-intensive implementation aspects and (ii) time-intensive simulation limit the possibilities to estimate parallel VCA designs at an early stage of the design process.

This thesis tackles these problems in two steps. Firstly, we will introduce a modelling technique that allows the system designer to describe a VCA in an abstract way. We will combine *complexity estimation* and *virtual prototyping* techniques for describing "virtual" architecture configurations. A framework that allows virtual prototyping of abitrary software and hardware architectures for video coding that overcomes the need for implementing software or hardware partitionings is developed. Secondly, a simulation framework that enables the simulation of this "virtual" platform is introduced. It enables the system designer to obtain accurate estimates for the runtime complexity of the VCA when decoding a video stream.

In summary, the capabilities of current runtime analysis and prediction techniques are typically not suitable for predicting the runtime behaviour of single-core VCAs on a parallel architecture in an accurate and fast way. The techniques introduced in this thesis equip the system designer with a new toolset for tackling this problem and to reduce the technological risk during the system design.

1.3 Contributions

The methods and applications contributed in this work shall enable system designers to efficiently explore the behaviour of VCAs on parallel hardware architectures. The main contributions of this thesis are summarized in the following:

- We investigate what information is provided by traditional single-core profiling techniques and introduce an innovative technique for mapping this information onto the functional blocks and coding elements of VCAs. This Data Driven Profiling (DDP) mapping technique enables the system designer to derive essential information on the VCA's execution behaviour and for making assumptions about the runtime behaviour of the VCA on a parallel hardware platform.
- We introduce a modelling technique for describing VCAs' coding elements, functional tasks and the data-dependencies between these tasks. We introduce a high-level simulation methodology, the Partition Assessment Simulation (PAS), for the modelling and simulation of parallel VCA hardware architectures. This methodology estimates the performance of a VCA for arbitrary virtual hardware and software configurations and enables design space explorations of parallel video processing architectures.

- We provide a simulator for analysing implementation aspects of the PAS methodology. We verify the methodology on an existing hardware architecture and analyse its accuracy for a real-world H.264 decoder scenario.

- We perform design space exploration for an H.264 decoder and evaluate the runtime performance of various decoder partitionings on a virtual architecture.

We believe that the proposed high-level methods for estimating the computational complexity of multi-core video coding systems is preferable over existing techniques, since these are typically not suited for the complex nature of multi-core systems. They can often not consider the concurrency and inter-processor dependencies inherent to multi-core systems. A valid alternative to our method is represented by the simulation-based prediction techniques described in Section 2.3. These methods can handle concurrency and inter-processor dependencies. However, for simulating the runtime behaviour of a parallel VCA, these approaches typically require a well-defined or completely implemented architecture and a partitioned software (i.e. low-level specification of the interfaces and components). Due to the vast amount of work that is required to implement each VCA partitioning approach, early high-level complexity estimations are difficult to realize and the flexibility to explore many different software designs is limited. The methodology introduced in our work aims to enable fast design space exploration and to estimate complex multi-core video coding systems in a flexible, time- and labor-efficient way.

1.4 Resulting publications

The following publications in scientific journals and at conferences have resulted from the work presented in this thesis:

Journals

- F. H. Seitner, M. Bleyer, M. Gelautz, R. M. Beuschel: Evaluation of data-parallel H.264 decoding approaches for strongly resource-restricted architectures, *Journal on Multimedia Tools and Applications*, Springer, volume 53, issue 2, pages 431-457, 2011.

- F. H. Seitner, M. Bleyer, M. Gelautz, R. M. Beuschel: Development of a high-level simulation approach and its application to multi-core video decoding, *IEEE Transactions on Circuits and Systems for Video Technology*, volume 19, issue 11, pages 1667-1679, 2009.

Conferences with proceedings

- F. H. Seitner, M. Bleyer, R. Schreier, M. Gelautz: Evaluation of data-parallel splitting approaches for H.264 decoding, *Proc. of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pages 40-49, Linz, 2008. (oral presentation)

- F. Seitner, M. Bleyer, M. Gelautz: Development of multi-core video decoding platforms based on high-level architecture simulations, *Proc. of the Junior Scientist Conference*, pages 71-72, Vienna, 2008. (oral presentation)

- F. Seitner, J. Meser, G. Schedelberger, A. Wasserbauer, M. Bleyer, M. Gelautz, M. Schutti, R. Schreier, P. Vaclavik, G. Krottendorfer, G. Truhlar, T. Bauernfeind, P. Beham: Design methodology for the SVENm multimedia engine, *Proc. of the Austrochip 2008*, page 113, Linz, 2008. (poster presentation)

- F. H. Seitner, R. M. Schreier, M. Bleyer, M. Gelautz: A high-level simulator for the H.264/AVC decoding process in multicore systems, *Electronic Imaging*, SPIE, volume 6821, pages 5-16, San Jose, 2008. (oral presentation)

- F. H. Seitner, R. M. Schreier, M. Bleyer, M. Gelautz: A macroblock-level analysis on the dynamic behaviour of an H.264 decoder, *IEEE International Symposium on Consumer Electronics*, pages 1-5, Dallas, 2007. (oral presentation)

Patents

- R. Schreier, F. Seitner: Method and apparatus for encoding and decoding of video streams, *US Patent*, Application number 20080152014, filed 12/2007.

Technical reports

- F. H. Seitner, R. M. Schreier, M. Bleyer, T. Albrecht, M. Gelautz: Analysis of video algorithms, *FIT-IT Project VENDOR*, WP2.2, Vienna University of Technology, 2007.

- F. H. Seitner, R. M. Schreier, M. Bleyer, T. Albrecht, M. Gelautz: Literature survey of state-of-the-art video algorithms, *FIT-IT Project VENDOR*, WP2.1, Vienna University of Technology, 2007.

1.5 Organization

The content of this thesis is organized into seven chapters. The current chapter provided a general overview about the motivation and contributions of this work. In the following, we briefly describe the chapters in the remainder of this thesis and provide links to the publications listed before.

- In Chapter 2 we provide an overview of existing complexity estimation and runtime profiling techniques. We discuss the individual techniques and explain their limits when it comes to multi-core architecture design and design space exploration. The main text of this chapter is taken from our published papers [SSBG11, SSBG09].

- Chapter 3 outlines the fundamentals of hybrid video coding standards. In this chapter, we derive the characteristics of hybrid video coding algorithms and describe the design challenges of parallel video coding systems. The main text of this chapter is taken from our published papers [SBSG08, SSBG11, SSBG08], with more details and additional explanations.

- We exploit these characteristics in Chapter 4 to introduce a novel runtime profiling method. It addresses various short-comings of traditional dynamic profiling techniques and enables the correlation of runtime complexity with specific coding elements and functional blocks of a hybrid video coding algorithm. This provides important insights into the complexity and can be exploited for identifying bottlenecks and potential challenges in the design of parallel coding solutions at an early stage of the development. The main text of this chapter is primarily a compilation of our published papers [SBSG08,SSBG11,SSBG09,SSBG07], with additional results and experiments.

- In Chapter 5, we introduce a virtual prototyping methodology, the Partition Assessment Simulation (PAS) technique. We explain the design goals and theoretic fundamentals and describe an implementation of this prototyping concept. The main text of this chapter is primarily a compilation of our published papers [SBSG08, SSBG11, SSBG09], with additional details and explanations of the PAS concept and its implementation.

- In Chapter 6, we analyse the PAS in more detail and evaluate and verify this technique's accuracy using a real-world H.264 decoder. We use the PAS for modelling a virtual VCS for demonstrating the possibilities towards efficient design space exploration using examples of functional as well as data-parallel partitioning approaches. The main text of this chapter is primarily a compilation of our published papers [SSBG11, SSBG09], with additional experiments and results.

- Chapter 7 provides conclusions and an outlook on future work. The main text of this chapter is primarily a compilation of our published papers [SSBG11, SSBG09].

CHAPTER 2

Prior work on complexity and runtime estimation

Obtaining information about the runtime complexity of an algorithm is typically highly important when developing the hardware or software components that compute the algorithm's individual processing steps. Accurate performance analysis supports important stages of a development process such as the system design, optimisation, functional verification and testing. For estimating runtime complexity, various prediction techniques have been developed. We can divide the existing estimation techniques into three major groups: the analytical, the profiling-based and the simulation-based approaches. In the Sections 2.1, 2.2 and 2.3, we will describe these techniques in more detail. Section 2.4 provides an overview of high-level design exploration techniques. In Section 2.5, we put the techniques contributed in this thesis in the context of prior work.

2.1 Analytic runtime prediction

Based on the fundamentals of the Computational Complexity Theory [FH03], advanced analytic methods for analysing an algorithm's complexity have been introduced. For example, the Static Algorithm Analysis [PK89] and Worst Case Execution Time (WCET) estimation [LM95, MML97] have been evolved. These techniques analyse formal definitions of an algorithm, for example its source code, for estimating the algorithm's computational complexity.

Most theoretic complexity approaches describe the complexity of an Algorithm A using an *instance-based* complexity measure $T_A[\cdot]$ [ST09]. This measure defines the complexity $T_A[x]$ of an Algorithm A for an input instance x. For each Algorithm A, an input domain Ω containing all possible input instances x is provided. For a finite input domain Ω, this complexity measure defines a $|\Omega|$ dimensional vector (i.e. each element in this vector represents the complexity $T_A[x]$ of an input instance $x \in \Omega$). Depending on the number of input instances $|\Omega|$, the effort for estimating and describing an algorithm's complexity strongly varies.

For a more specific analysis of an algorithm's complexity, the input domain Ω is usually viewed as the union of a set of sub-domains $\{\Omega_1, \Omega_2, ...\Omega_n\}$. Each sub-domain Ω_i represents all input instances of size i. For example, in the context of sorting algorithms, Ω_i refers to the set of all tuples containing i elements.

Based on these sub-domains, the complexity of an algorithm A is often described as a function of the input size of the problem A aims to solves. We can define a scalar $T_A(n)$ which summarizes the complexity $T_A[x]$ for all instances $x \in \Omega_n$. In our example with the sorting algorithm, $T_A(n)$ describes the complexity of sorting an input instance with n elements. In *Theoretical Computer Science*, the WCET is one of the most commonly used metrics for summarizing the complexity of an algorithm. The WCET of Algorithm A can be derrived in the following way:

$$WCET_A(n) = \max_{x \in \Omega_n} T_A[x] \tag{2.1}$$

It describes the maximal execution time of an Algorithm A processing an input instance consisting of a tuple of n elements.

Runtime prediction based on static analysis measures has multiple shortcomings. First, the execution paths of most algorithms depend on the data values of the input instances. For example, input-dependent recursions and branches in an algorithm can cause *dynamic* variations in an algorithm's execution path and its runtime. Consequently, theoretic complexity measures such as the WCET do not necessarily reflect an algorithm's runtime behaviour under real working conditions [ST09].

Second, analytical complexity predictions cannot easily be bound to a specific hardware platform. The runtime of an algorithm depends on the processing resources of the executing platform (e.g. instruction set, processing pipeline, clock rate). An algorithm that performs well in theory not necessarily does this on a platform with physical processing resources and architectural limitations.

2.2 Runtime prediction based on dynamic profiling

Dynamic profiling aims to address the limitations of analytical runtime prediction by observing the execution of an algorithm on a physical *reference platform*[1]. Most hardware platforms provide tools for observing program execution during runtime and for *measuring* the runtime of executing programs. Prediction techniques based on dynamic profiling exploit the knowledge gained from these *complexity measurements*. Based on observations of the execution behaviour of a Program P on a reference platform R, the system designer makes assumptions about the program's execution behaviour.

Similar to the input domain Ω used in analytical runtime prediction, runtime measurements obtained via dynamic profiling aim to reflect the complexity for a range of input instances. For example, for a video decoder the bitrate of a coded input stream typically has a strong impact on the runtime complexity. This can be used for defining input sub-domains $\{\Omega_1, \Omega_2, ...\Omega_n, ...\}$ for

[1] In the context of this work, we use the term program to refer to an algorithm's source code or binary representation on a physical hardware platform.

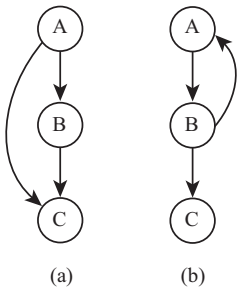


Figure 2.1: A simple control flow graph (CFG): Each node represents a basic block (BB) of a program. The directed edges between the nodes represent the jumps from one BB to another BB. (a) A CFG with three BBs: Block A represents the first BB of the program. After Block A either Block B or Block C are executed. Block C is the last BB executed within the program. (b) A CFG with three blocks and a loop between Block A and B.

our VCA where each sub-domain represents the decoding complexity of video streams within a well-defined bitrate range. For obtaining the runtime complexity of a subdomain Ω_i, we would profile multiple input streams within the domain's bitrate range. The longest or average execution times a VCA requires to decode these streams would reflect the WCET and the average runtime complexity for this input sub-domain, respectively.

In practice, dynamic profilers regard a program as a set of Basic Blocks (BBs). The term Basic Block has been introduced by Allen [All70] and refers to a linear sequence of program instructions that has no jump instructions contained within it. The first and last instructions of each basic block are called *entry point* and *exit point*, respectively. For entering a BB, the entry point of this BB may be entered from one or more exit points within the program. [2]

BBs are usually the basic units a compiler works with during the optimisation phase and also enable profiling of the individual program regions. The program is regarded as a graph where the BBs form the graph's nodes and the jumps between the BBs the transitions between the nodes. These graphs are called Control Flow Graphs (CFGs). In Figure 2.1a and 2.1b, we show two simple CFGs with three states each. In Figure 2.1a, a transition from State A to State B as well as to State C is possible. In State B, only a transition to State C is possible. Figure 2.1b contains a loop where the States A and B can be executed multiple times before reaching the final State C.

The execution time of a Program P is the sum of execution times of all its BBs multiplied by the number of executions of the BBs. We can compute the execution time t_P of Program P that consists of n BBs ($BB_1..BB_n$) in the following way:

$$t_P = \sum_{i=1}^{n} t_{BB_i} * f_{BB_i} \tag{2.2}$$

[2] A BB Y that is entered after the execution of a BB X is called a *successor* of BB X. BB Y is called the *predecessor* of BB X.

The terms t_{BB_i} and f_{BB_i} refer to the execution time and the number of executions of BB_i, respectively.

In practice, dynamic profiling techniques can be classified into two major groups: *Statistical* and *instrumented* profiling techniques. In the following sections, we describe these techniques in more detail.

2.2.1 Statistical profiling

In statistical profiling, the program counter (PC) of a program is observed during the program's execution. The value of the PC represents the position where the execution of the program currently takes place. By sampling the PC in regular time intervals, conclusions about the executed parts of the program and the frequency at which these parts are executed can be derived. The time interval between each sample (i.e. the *sampling period* T_S) is typically known to the profiling environment. It is measured in seconds and is the inverse of the *sampling frequency* F_S (i.e. the number of samples taken per second):

$$T_S = \frac{1}{F_S} \tag{2.3}$$

Based on the total number of samples n_P that lie within memory blocks assigned to a Program P, we can estimate the total runtime $\hat{t}_P$ of this program:

$$\hat{t}_P = T_S * n_P \tag{2.4}$$

In this estimation, each sample is counted as a period of T_S seconds.

Since the memory location of each BB of the program is typically known, we can unambiguously assign each PC sample to a BB. This allows us to derive the number of samples n_{BB_i} that occurred within a block BB_i during the execution of P and to estimate the total runtime $\hat{t}_{BB_i}$ spent in BB_i during the program's execution:

$$\hat{t}_{BB_i} = \hat{t}_P * \frac{n_{BB_i}}{n_P} \tag{2.5}$$

The factor $\frac{n_{BB_i}}{n_P}$ is the percentage of the BB's runtime on the total runtime. $\hat{t}_P$ is the total runtime of Program P

Note that $\hat{t}_{BB_i}$ is statistically approximated and does not necessarily represent the exact runtime of BB_i. Especially, when using a low sampling frequency F_S and when measuring small and rarely executed BBs, the number of observed samples can strongly vary between measurements. Furthermore, no accurate information on the number of executions of each BB during the execution of P can be obtained by statistical profiling techniques.

2.2.2 Instrumented profiling

Instrumented Profiling (IP) techniques extend the target progam with additional program instructions [GKM82, BL94]. *Instrumentation* refers to the task of inserting instructions for profiling

Listing 2.1: Example of a dynamic runtime trace: For each execution of a function, the times when the function is entered and exited are retrieved by the profiler.

```
1  CALL StartH264Decoder                time =     1
2    CALL DecodeFrame                   time =     2
3      CALL DecodeMacroblock            time =  1000
4      ...
5      RETURN                           time = 10000
6      ...
7      CALL DecodeMacroblock            time = 10020
8      ...
9      RETURN                           time = 23000
10     CALL DecodeMacroblock            time = 23001
11     ...
12     RETURN                           time = 35000
13   RETURN                             time = 35001
14 RETURN                               time = 35002
```

Function name	Calls	Gross runtime		Net runtime		Net runtime (Cycles/Call))		
		Cycles	%	Cycles	%	Min.	Avg.	Max.
StartH264Decoder	1	35002	100.00	2	0.1	2	2	2
DecodeFrame	1	35000	99.9	1019	2.8	1019	1019	1019
DecodeMacroblock	3	33981	97.1	33981	97.1	9001	11327	12981

Table 2.1: Dynamic profile: For a VCA consisting of three functions, the table provides the profiled gross and net runtimes. Additionally, the table shows the minimal, average and maximal runtime for each function call. More details are provided in Section 2.2.2.

purposes into a program. These instructions collect information about the behaviour of the program during runtime such as the program's execution path. Instrumented Profiling at BB level can gather information about the time when a BB is entered and exited, the frequency f_{BB_i} and duration d_{BB_i} of a basic block BB_i.

The output of an instrumented profiler typically contains a stream of recorded events such as calls to the BBs of a program. This set of events is referred to as the profiler's *trace*. For a more intuitive interpretation by the system designer, modern profiles typically maps the events of a trace to the program's *functional level* (i.e. source code functions).

Listing 2.1 shows an example of a simple trace at functional level of a VCA. In this example, the VCA consists of three functions: *StartH264Decoder*, *DecodeFrame* and *DecodeMacroblock*. The trace provides insights when a function $f_j \in F_{VCA}$ is called or left. We refer to function f_j as *callee* and the function which called f_j as *caller*.

Dynamic profilers typically provide a summary of a trace's observations (i.e. the *profile*). A functional profile of the trace from Listing 2.1 is given in Table 2.1. This table provides a summarized complexity information that can be exploited for optimising the VCA's computational expensive parts. For each function, information such as the number of function calls, the absolute and relative runtime in cycles and percentage of the total program runtime are typically

obtained. Gross runtime (i.e. cumulative runtime of all functions that occur during a function execution), net runtime (=function's gross runtime without runtime spent for sub-function execution) and statistical information on minimum, average and maximum runtime per function call enable us to concentrate on runtime expensive functions during the optimisation. Time-intensive optimisations (e.g. hardware-dependent code optimisations using assembler code) for functions with insignificant complexity can be avoided.

However, placing profiling instructions inside a target program can cause changes in the runtime performance. Additional profiling instructions are executed during the program execution which increases the runtime complexity. The increased number of instructions can further cause changes in the platform's instruction caching strategy and result in significant changes in the total runtime. Modern architectures provide efficient hardware support for reducing the impact of instrumentation on a program's runtime. They provide specific instructions for tracing the program's execution with a minimal execution overhead. Furthermore, profiles have been evolved that estimate the complexity overhead caused by the profiling and correct the profiling results based on this.

Despite the advanced hardware support of instrumented profiling, the additional profiling instructions can have a significant impact on the compile process and the resulting binary. Impacts on the program runtime behaviour occur. For keeping this impact low, hybrid profiling techniques based on statistical sampling (Section 2.2.1) and IP are used in practice. This keeps the instrumentation overhead low and results in more accurate runtime measurments. For example in gprof [GKM82], instrumentation is used for collecting the information about function frequency and function entry/exit times and statistical sampling for measuring the runtime.

Instrumentation of profiling instructions

For instrumentation, manual as well as automatic techniques exist. *Manual Instrumentation* refers to the manual insertion of the profiling instructions into a target program's source code. This is typically used for profiling and debugging specific parts of a program. The manual insertion of profiling instructions can be highly labor- and time-intensive and for more extensive profiling, *Automatic Instrumentation* (AI) techniques have evolved.

AI automatically inserts profiling instructions at relevant positions of the program. Various AI techniques have been introduced that differ in the way the insertion of the profiling instructions into the program is done. *Automatic Source Level Instrumentation* (ASLI) analyses the source code of a program and inserts the profiling instructions directly into the program's source code (i.e. before the program's compilation into binary code). An example of ASLI has been introduced by Ravasi and Mattavelli [RM05]. They have developed the Software Instrumentation Tool (SIT) which extends traditional C source code to *instrumented* C++ classes. This instrumentation provides detailed information on the number of arithmetic and memory load/store operations executed during a program's execution.

In a similar spirit, the ATOMIUM tool [NCK+96] performs high-level transformation of C code. The focus of this tool lies on memory analysis. The *Data Transfer and Storage Exploration* methodology (DTSE) is introduced. Based on this methodology, C code can be optimised in terms of execution time, memory size and power consumption.

Apart from AI of the program's source code, techniques for instrumenting the binary of a program exist. This instrumentation can take place during the compiling [GS04] as well as the binary linking [SE94] stages of a program. Furthermore, dynamic binary analysis tools such as Valgrind [NS07], PIN [LCM+05] and DynamoRIO [BGA03] exist that instrument programs at runtime.

2.3 Simulation-based runtime prediction

In Simulation-based Runtime Prediction (SBRP), a *simulator* mimics the physical hardware platform and its behaviour over time. This enables the designer to model a hardware platform before it is physically created and to simulate a program's runtime execution on this "virtual" platform. Since the hardware is simulated, very detailed observation of the program's runtime behaviour on this platform is possible.

The existing simulation approaches can be divided into four major groups: *Hardware simulation, instruction set simulation, HW/SW-codesign* and *high-level simulation techniques*.

2.3.1 Hardware simulation techniques

For describing the hardware logic (i.e. the electronic circuits), the design and the temporal behaviour of a hardware design, Hardware Description Languages (HDLs) are typically used. Examples of HDLs are VHDL [VHD88], Verilog [TM91] and SystemVerilog [SDF06]. In contrast to software languages such as C, important characteristics of HDL languages are (i) the explicit notion of time and (ii) the capability to describe concurrent events in a formal way. Both characteristics are primary attributes of hardware and enable accurate specifications of circuits and physical hardware blocks. A simulator interprets the semantics of the HDL statements and mimics the behaviour of a hardware design's individual circuits over time.

The simulation of HDL descriptions allows the system designer to specify, test and verify the hardware logic before the design is physically built. However, HDL simulations of complex hardware designs are computationally very expensive and time consuming. This limits the ability to simulate the execution of complex software applications on an HDL-based hardware design simulator. Instruction Set Simulators are typically more suitable for this purpose.

2.3.2 Instruction set simulation

An Instruction Set Simulator (ISS) is a program that simulates the execution of a program on a programmable processor. The system designer describes the individual registers, the operations and the decoding pipeline of this processor. The ISS mimics the "virtual" processor's progress over time by simulating the execution of the program's individual instructions in the decoding pipeline.

Compared to an HDL simulator, an ISS regards each register as a "virtual" variable. Detailed profilings about a program can be retrieved without simulating the underlying hardware logic [CK94, WR96]. This reduces the simulation complexity and hence the simulation time

significantly. The simulation at a higher level of abstraction makes ISS computationally less expensive than HDL simulations and more suitable for analysing and developing complex software applications.

In [HS09], an accurate profiling tool based on ISS for fast and accurate performance, power, and memory access analysis of embedded systems is introduced. This approach simulates hardware and software at an instruction level which enables the exploration of different low-level hardware configurations setups.

2.3.3 HW/SW-codesign

The flexibility of software (SW) design compared to hardware (HW) implementations have resulted in the development of advanced HW/SW-codesign methods such as described in [KM96, CLN+02, YYS+04, WPH+05]. These methods enable the systematic integration, testing and verification of new HW design implementations. Verified SW implementations typically serve as a starting point for HW/SW-Codesign techniques. Compared to a hardware design, implementing a complex algorithm in software has multiple advantages. This includes, for example, a faster and more flexibe development using high-level programming languages and easier correction of design errors. One prime intention of HW/SW-Codesign is the systematic transition from a functionally verified complex SW implementation to a corresponding HW design. This is typically done in the following way:

First, the system designer verifies the functional correctness of the SW implementation and the individual functional components using e.g. an ISS or another physical platform the SW can be compiled and executed on. Second, one SW component after the other is transferred into a corresponding HDL description and simulated using a hardware simulator. By connecting the HW simulator with the simulation environment where the SW verification has taken place, each HW component can be tested in the context of the whole implementation. The designer can find differences between the software and hardware implementation and verify the correctness of new HW blocks. For example, by comparing the data that is exchanged via the interfaces between the SW and the HW components or by comparing the results between the “pure” SW and the HW/SW design.

Apart from verification and migration from SW to HW, HW/SW-codesign approaches exist that target the modelling of virtual prototypes for new system designs. Examples are the OVPsim simulator [Agr09], the M5 simulator system [BDH+06] and the simulation platform Simics [MCE+02, VAG05]. Typically, multiple processor simulation models are connected with each other. By simulating parallel execution and inter-communication of the SW components running on these processors, these approaches mimic the parallel system’s execution behaviour. They predict the real protoype’s runtime behaviour and provide means for efficiently developing real-world design concepts. This enables evaluation and improvement of the design as well as investigation of design alternatives before a real and expensive prototype is built.

These simulators simulate the HW as well as the SW components of the system. However, these methods have two major shortcomings. First, the simulators often mimic the exact behaviour of each HW component (i.e. processor pipeline, caches, memory subsystems, etc.). This results in high computational complexity and limits the possibilities for simulating many HW and SW configurations. Second, each SW component has to be implemented for its specific

target processor and requires an individual SW partitioning for each virtual design. This is time-consuming and reduces the flexibility to explore many SW partitionings and different HW/SW mappings.

2.4 High-level design exploration

High-level design exploration aims to reduce the design effort for complex systems by introducing abstract algorithm models that can be efficiently simulated and verified on virtual platforms. The Ptolemy II software environment [EJL+03, Lee10] takes a step towards event-oriented modeling of heterogeneous and embedded systems. This framework focuses on hierarchical description of complex heterogeneous systems. The main focus of Ptolemy is the hierarchical structuring and combining of multiple models into a heterogeneous system. This includes efficient ways to define nested models and sub-models and the unambiguous definition of heterogeneous systems using multiple simulation models.

In the context of high-level simulation and video coding, the area of Reconfigurable Video Coding (RVC) [CAM09, BEJ+11] has evolved recently. The prime goal of RVC is implementation independency and retargetability of video coding algorithms. It uses the CAL actor-language [EJ03] for describing the functionality of a video coding algorithm in an abstract way without taking into account any concrete implementation. Based on a CAL high-level description, an automatic transformation into an implementation language such as C or SystemC and further into a low-level representation is possible. This enables fast implementation of video coding tools in a platform-independent way.

2.5 Partition Assessment Simulation in context of prior work

In a spirit similar to Ptolemy, the Partition Assessment Simulation (PAS) that is introduced in this thesis uses an event-oriented modeling approach for mimicking the execution of parallel architectures. The underlying concept behind PAS combines traditional profiling techniques and high-level modelling and simuation approaches for obtaining accurate runtime estimations of complex and virtual multi-core VCSs. While more details on the PAS will be provided in Sections 4 and 5, this section aims to set the PAS concept into the context of prior work.

The PAS can be seen as an extension of traditional dynamic runtime profiling and high-level simulation techniques. The principle of traditional dynamic profiling techniques is extended in a way that runtime complexity can be set in the context of a VCA's data structures, functional blocks and the input data that is processed. A VCA and its data-processing behaviour can be defined in an abstract way and runtime profiling information can be mapped onto these definitions. Especially for data-intensive and parallel applications such as multi-core video coding, this technique can provide essential insights into a program's runtime behaviour.

In the context of high-level simulation, the availibility of such a detailed runtime profiling information opens up new means for estimating the runtime behaviour of virtual and distributed VCAs. Multi-core HW platforms as well as VCAs can be described by high-level models. By introducing means for simulating these models and by exploiting the obtained profiling information, accurate runtime predictions become possible. Prototyping of many new virtual designs

and exploration of different parallelisation approaches can be done without needing to adapt the software design.

In contrast to CAL/RVC, PAS focuses only on modelling aspects essential to the parallel execution behaviour of a VCS and less on detailed functional description. This enables a clear focus on high-level design aspects of parallel systems without the need to specify low-level functionality (i.e. below the level where the parallelisation takes place) and results in simplicity and descriptive clarity. The PAS exploits available hardware profiling information during the high-level simulation and can make accurate runtime predictions without the need for a detailed system description.

2.6 Summary

Various techniques for runtime estimation have been introduced in previous works. These techniques can be grouped into analytical, profiling-based and simulation-based methods.

Analytical estimation techniques enable performance estimations without any concrete hardware platform and only based on formal algorithm definitions. However, they are not well-suited for estimating dynamic and input-data dependent runtime behaviour of more complex video coding algorithms in an accurate way.

Statistical and instrumented profiling techniques can address this shortcoming but require a reference platform where measurements can be obtained. However, these techniques are not applicable for making runtime predictions for virtual architectures during the design phase since at this stage no implementation exists.

Simulation techniques that model the hardware architecture and runtime behaviour in a detailed (bit-accurate) way such as hardware simulation and instruction-set simulation techniques can provide accurate runtime predictions in this case. However, the modelling of a VCS using existing simulation techniques is typically too time-intensive to be usable for virtual prototyping in early design stages. The focus of these simulation approaches is on accurate modelling of the functionality and less on obtaining runtime estimates in a fast way.

The PAS methodology introduced in this thesis tries to combine existing profiling techniques and the idea of simulated runtime estimation to provide a high-level vitual prototyping solution. Efficient runtime predictions become possible in a flexible way. This is essential for fast VCS design, which has to address the short development cycles of today's video coding applications.

CHAPTER 3

Characteristics of modern video coding algorithms

In this chapter, the characteristics of state-of-the-art video coding algorithms and architectures are described. Understanding the fundamental structure and processes of VCAs and their impact on the hardware architecture is essential for this work. It enables us to derive the methods for performance profiling and simulation of virtual video coding systems that are introduced in later chapters. We focus on the video decoder design since parallelisation of this part of the coding process is typically more challenging than the encoder side. This results from the fact that video coding standards typically specify the decoding part (e.g. coding tools, maximum bitrate and resolution, etc.) very precisely and place strong constraints on the decoder. The encoder's functionality is rarely specified, which provides more flexibility when implementing the encoder's design. Furthermore, decoder solutions are typically located in consumer products and run on computationally less powerful hardware. This results in a high demand for computationally efficient decoder solutions.

After a short historical overview on *digital video coding* in Section 3.1, characteristics of modern video coding algorithms are introduced in Section 3.2. We use the H.264 video coding standard for characterising the structure and mechanisms of hybrid video coding. In Section 3.3, we describe how video data is typically structured in a hierarchical way for achieving resource-efficient data processing. Section 3.4 provides more details on video coding tools available in state-of-the-art video coding standards. In Section 3.5, we describe various parallelisation approaches for H.264 decoding and explain the challenges of parallel decoder designs.

3.1 Historical development of digital video coding

In 1984, the H.120 video coding standard [ITU93] was introduced by the ITU-T (International Telecommunication Union - Telecommunication). This coding standard was based on DPCM (Differential Pulse Code Modulation) coding and achieved video compression by reduc-

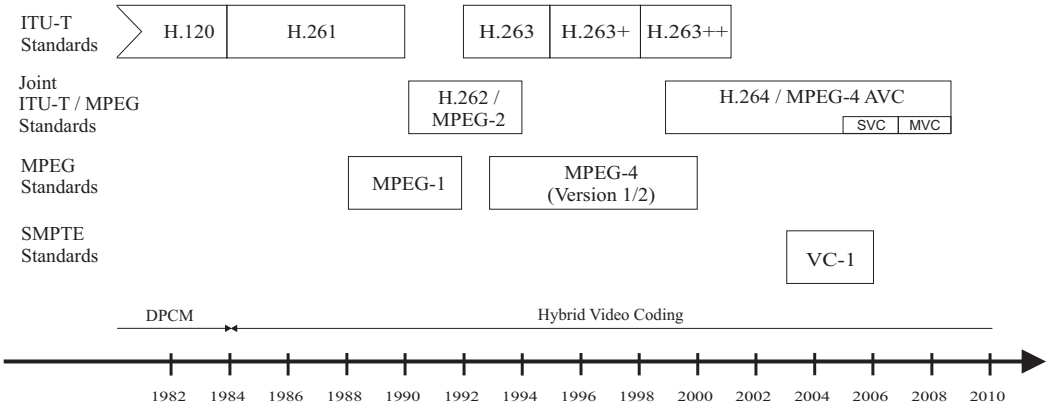


Figure 3.1: Historical development of international digital video coding standards based on the overview in [Beu10].

ing short-term redundancy in the video signal. Even though the H.120 standard was extended in 1988 and 1993 by adding more advanced coding tools such as motion-compensation, the high computational complexity of DPCM at encoder and decoder side and low compression efficiency led to the emerging of more advanced *hybrid video coding schemes*. These hybrid schemes typically combine temporal prediction (i.e. inter-prediction between frames) and local prediction (i.e. intra-prediction inside a frame) for removing temporal as well as spatial redundancy. This scheme is referred to as hybrid video coding and builds the foundation of most video coding standards used today.

The concept of hybrid video coding was first applied in the H.261 [ITU88] video conferencing standard, which was released in 1988 by the ITU-T standardisation organisation. Conceptual elements of this algorithm included hierarchical structuring of video data into macroblocks (MBs), MB-based motion compensation and transform as well as variable-length code (VLC) entropy coding schemes. These techniques were further extended by the MPEG-1 video standard (ISO/IEC 11172-2) by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The MPEG-1 standard was released in 1992 by the ISO/IEC and used the same concepts and coding tools as H.261. The major improvement compared to H.261 was the introduction of half-pixel accurate and bi-directional motion prediction resulting in higher compression efficiency at the cost of increased processing complexity.

New tools for coding of interlaced images as well as high bitrates and high resolution images up to *High Definition* (i.e. 1920 x 1080 pixel) were introduced in the MPEG-2 standard. MPEG-2 is a joint standard of the ISO and ITU working groups and referred to as ISO/IEC 13818-3 [ITU00] and ITU-T H.262. In contrast to most video coding standards, MPEG-2 was backward compatible with its predecessor MPEG-1 (i.e. MPEG-2 decoders support decoding of MPEG-1 streams). MPEG-2 was widely accepted for media distribution and broadcasting, for example on Digital Versatile Discs (DVDs) and for Digital Video Broadcasting (DVB), and has been used within a wide range of consumer and professional products for video storage and digital broadcasting.

In 1995, the H.263 [ITU05] standard for video conferencing applications was released. Essential coding tools for video conferencing and low-bitrate coding applications such as variable prediction block sizes, advanced deblocking and forward error correction were introduced. Later enhancements of H.263 were released in 1998 and in 2001 and are known as H.263+ and H.263++, respectively.

The MPEG-4 standard [ISO01] released in 1999 introduced a wide range of advanced coding tools for describing and coding of mixed media formats such as audio and video coding, 3D graphics content, animation and fonts. Originally, video coding functionality was specified in Part 2 of MPEG-4. This part provides two coding profiles, the Simple Profile (SP) which targets low bitrate scenarios and the Advanced Simple Profile (ASP) targeting higher bitrate coding. In 2003, the MPEG-4 Part 10 Advanced Video Coding standard was released. This standard has been jointly developed by ITU and ISO and is also known as H.264/AVC [ITU12]. The primary development targets of H.264 were significant improvements in coding efficiency, an bit-exact match between encoding and decoding for avoiding drifts between encoder/decoder side and advanced error robustness. Due to the strong improvements of H.264 in terms of coding efficiency and its flexibility to address many applications such as low-bitrate and low-latency transmission efficiently, H.264 has gained a dominant position amongst current video standards. This can be seen in its obligatory support in the Blu-ray standard and for DVB broadcasting and as an HTML5 video standard.

Various amendments have been added to H.264 over the last years. For example, in 2007 the Scalable Video Coding (SVC) amendment and in 2009 the Multi View Coding (MVC) amendment have been added. SVC provides coding capabilities to H.264 to efficiently encode video signals at multiple spatial resolutions, multiple temporal resolutions and multiple quality levels. This increases the flexibility when distributing content over heterogeneous media channels with different transmission capabilities or to displaying devices with strongly different displaying capabilities. MVC is targeting efficient coding of multiple video signals where redundancies between the signals exist. An application for MVC would be coding of stereoscopic 3D content where streams for left and right eyes and from similar viewing positions are encoded simultaneously.

As a competing standard to H.264, the VC-1 video coding standard [SMP06, KL07, JBH08] was released in 2006 by the Society of Motion Picture and Television Engineers (SMPTE) under the name SMPTE 421M. Originally, VC-1 was based on Microsoft's Windows Media Video 9 (WMV-9) codec [SHH+04] and is functionally equivalent to this codec. Next to H.264, it is one of the obligatory standards used for coding video data on Blu-ray discs.

Apart from international video standards, a wide range of national standards such as the *Audio and Video coding Standard of China (AVS)* [WZ06, BJR+07] or open-source codecs such as *VP-8* [BWX11] have been introduced. The majority of these standards is based on a hybrid coding scheme and uses similar coding tools too those provided in H.264 and VC-1.

3.2 Concept of hybrid video coding

In this section, the characteristics of hybrid video coding algorithms are explained. In the context of this work, we use the H.264 video standard [ITU12] for explaining the fundamental

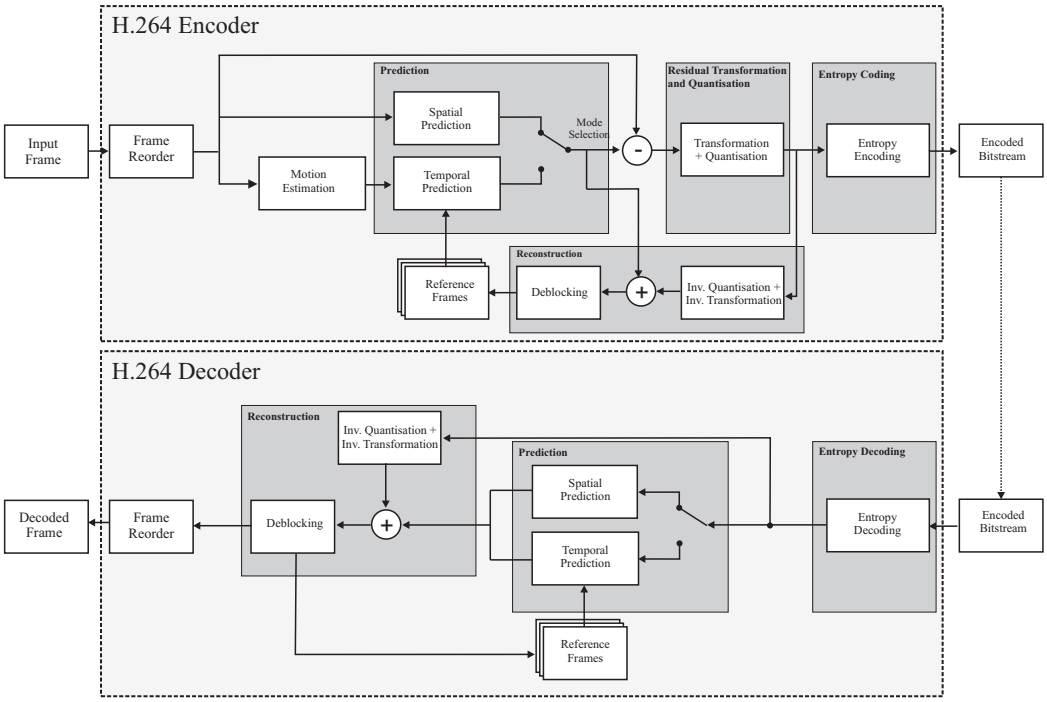


Figure 3.2: H.264 encoding and decoding processes: The major functional blocks of the encoder can be divided into motion estimation, spatial and temporal prediction, residual transformation and quantisation, inverse quantisation and inverse transform, frame deblocking and entropy encoding. The decoder contains a subset of the encoder's functional blocks: Entropy decoding, inverse quantisation and inverse 2D transform, prediction and deblocking.

mechanisms underlying hybrid video coding. The strong structural similarities between hybrid video coding algorithms make the results of this thesis also applicable to other standards such as MPEG-2, VC-1 or AVS. Apart from structural similarity, H.264 represents the development of video coding over the last three decades and belongs to the most efficient but also computationally most demanding video coding algorithms available. This makes parallelisation an attractive option for H.264 encoder/decoder designs.

Figure 3.2 visualises the H.264 encoding and decoding processes. The first stage of all hybrid video coding algorithms is the prediction of each frame's pixel values based on information from neighbouring pixels within the same frame (i.e. spatial prediction) or neighbouring frames (i.e. temporal prediction). Second, the encoder computes the difference between predicted and original pixels and transforms this *residual information* using a *discrete cosine transform* (DCT). This DCT transformation results in a spatial decorrelation and an efficient represention of relevant information in a few coefficents. After the DCT, all coefficients are quantised. This step aims to remove information the human visual system (HVS) is less sensitive to. Video coding algorithms such as H.264 that typically lose information during this quantisation are referred to as *lossy coding* algorithms. The rate control of an encoder is located at this quantisation stage since adapting the "aggressiveness" of the quantisation enables the encoder to control the amount of

information that is discarded and the bitrate used for encoding the video data. Third, in the reconstruction the coded residuals are inverse quantised and inverse transformed. Accumulated with the prediction data, this information is used for temporal prediction of future frames. H.264 uses a deblocking filter for removing blocking artifacts at the boundaries between MBs. The filter aims to remove blocking artifacts while maintaining the sharpness of true edges. In H.264, an advanced deblocking method was introduced, which increases the subjective quality significantly and results in a bit rate reduction of 5%-10% for the same objective quality compared to the non-filtered video. In contrast to previous standards, the deblocking filter is applied within the reconstruction loop of the en-/decoding process (Figure 3.2) and is also referred to as *in-loop deblocking*. Fourth, the decoder uses an entropy coding algorithm for removing statistical redundancy within the coded elements of prediction and residual data.

On the decoder side, these steps are done in a reverse order. First, the decoder entropy decodes the coded bitstream and retrieves the residual information and the prediction information from the encoded bitstream. Second, the decoder predicts each MB's pixel data using either spatial or temporal prediction. Third, the predicted pixel information is combined with the inverse quantised and inverse transformed residual information. As a last step, debocking of the decoded frame removes blocking artifacts that occur at the borders between neighbouring MBs.

It is important to note that video coding standards aim to ensure interoperability and syntax capability [SW05] between encoding and decoding sides, and that all ITU-T and ISO/IEC JTC 1 video coding standards only specify the decoding process. Each specification typically includes (i) a specification of all data structures and coding elements known to the video standard such as slices and macroblocks and (ii) algorithmic desriptions of the coding tools that can be applied to the individual data elements. In Sections 3.3 and 3.4, these two essential aspects of modern video coding standards are explained in more detail.

3.3 Hierarchical structuring of video coding elements

For flexibility and the need to address different coding application requirements, the H.264 video coding standard specifies a *Video Coding Layer (VCL)* and a Network Abstraction Layer (NAL). While the VCL defines the coding elements that are used for representing the video data in a hierarchical way, the NAL defines how each VCL element (and additional header information) can be formated into a data representation that is suitable for network transmission. For parallelisation the VCL is of prime importance since data-parallelisation is typically implemented by parallel processing of multiple VCL coding elements.

Since H.201, the VCLs of all ITU-T and ISO/IEC JTC 1 video coding standards have been based on block-based hybrid video coding schemes [SW05]. The VCL coding elements defined in these standards are based on similar hierarchical structures as depicted in Figure 3.3. This figure shows how H.264 divides each video sequence hierarchically into *GOPS* (Group of Pictures), frames, slices, *macroblocks* (MBs) and blocks. A GOP represents a set of consecutive frames within a video.

Each frame is divided into squared regions of 16x16 pixels, the *macroblocks* (MB). MBs form the core coding elements of the H.264 standard and most coding tools are defined in the

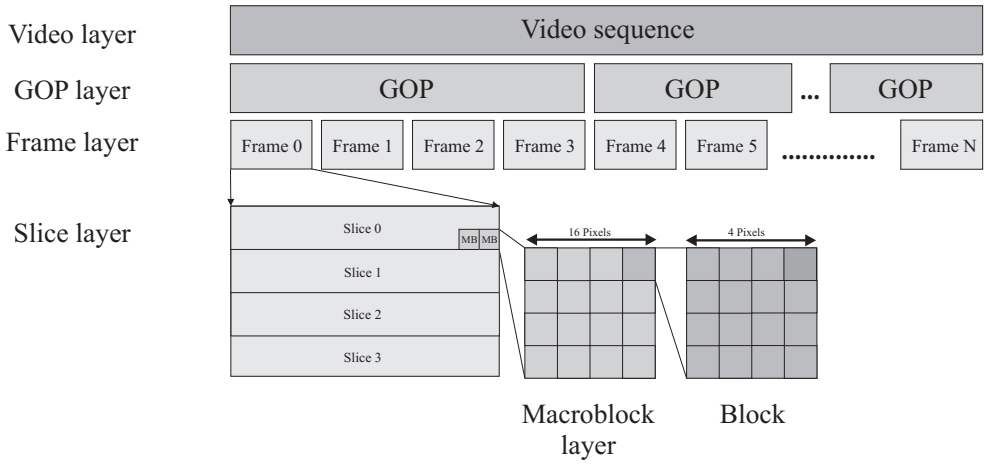


Figure 3.3: The H.264 video standard hierarchically structures the video stream into multiple layers. It divides the video stream into groups of frames, namely the Group-of-Pictures (GOPs). Each frame can be divided into multiple slices. A slice can be further divided into regions of 16×16 pixels, namely the Macroblocks (MBs).

context of MBs. Each video frame can contain one or multiple *slices*, whereas each slice represents a region within an image that can be coded independently of the other frame's slices.

The strong hierarchical structuring of the video content allows processing at various levels of granularity. The size of the available memories (e.g. external memories, processor caches) and the requirements on the processing latency typically influence at which level of granularity the processing effectively can be implemented. Parallelisation often takes place at a MB level since most architectures can process, transfer and store these blocks of data in an efficient way and within their architecure's memory limitations.

3.4 Coding tools

Hybrid video coding standards today use a wide range of advanced coding tools for coding VCL elements such as MBs efficiently. This section provides a brief description of the most essential coding tools used in the H.264 video standard. These tools introduce means for removing spatial and temporal redundancies when representing video content and enable efficient representation of VCL elements such as MBs.

3.4.1 Spatial prediction

The concept behind spatial prediction is based on the fact that pixels within the same frame, especially if spatially close, often poses a high similarity. Spatial techniques can exploit this similarity and reduce *spatial redundancy*. Spatial prediction provides means for deriving a pixel's information from other regions within the same frame. Spatial prediction is often referred to as *intra prediction* since no referring to other frames of the video sequence takes place. In H.264, a set of *intra prediction modes* is provided. Each mode describes a specific pattern

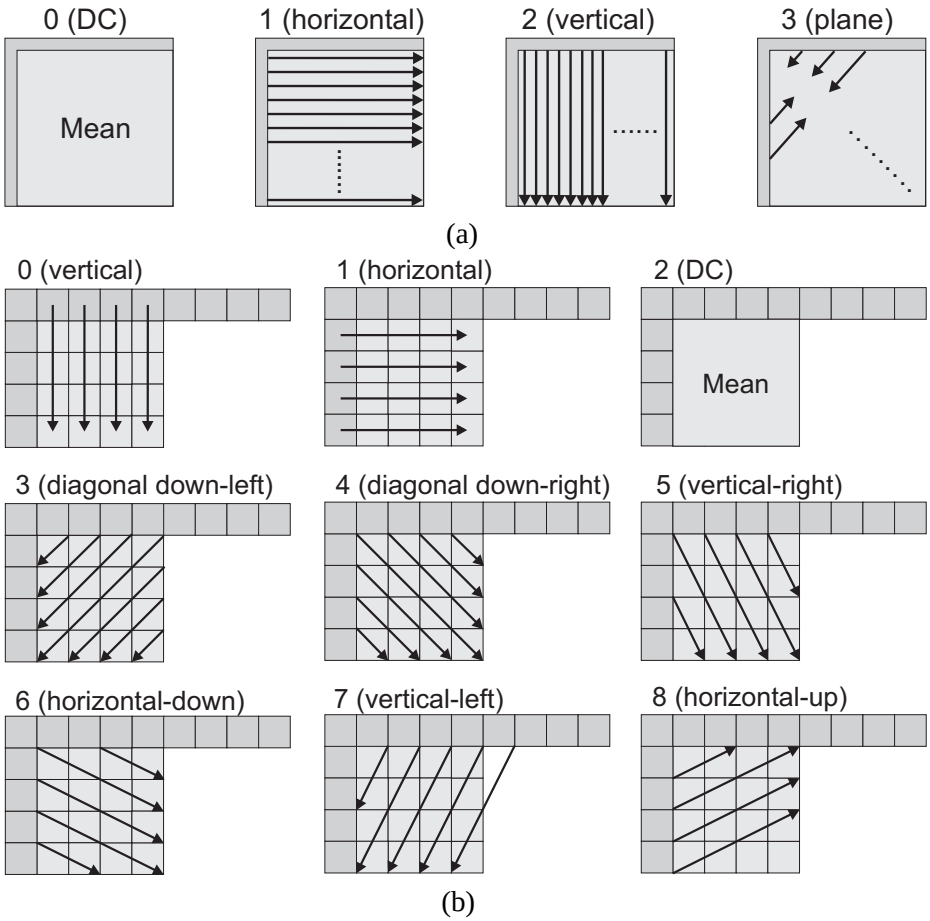


Figure 3.4: H.264 intra-prediction modes for blocks of (a) 16x16 and (b) 4x4 pixel size.

to derive a MB's pixels of the border pixels of spatially adjacent MBs (Figure 3.4). For example, the pixels of a 16x16 MB can be predicted using one of 4 prediction modes visualised in Figure 3.4a (Intra16x16). Pixels from neighbouring macroblocks are for example propagated in a vertical, horizontal or diagonal direction. Apart from Intra16x16, each MB can be divided into smaller blocks of 4x4 pixels and predicted with additional prediction patterns visualised in Figures 3.4b (Intra4x4). While Intra4x4 is more suitable for fine structures, Intra16x16 typically provides a more efficient coding for large homogeneous regions.

While in previous standards such as H.263 and MPEG-4/ASP intra prediction has been done in the transform domain (i.e. prediction of frequency coefficients), a paradigm change to intra prediction in the spatial domain (i.e. prediction of luminance/chrominance pixel information) has been introduced in H.264.

3.4.2 Motion-compensated prediction

Motion-compensated prediction (MCP) aims to exploit the similarity of consecutive frames within a video sequence. Changes between consecutive frames are typically caused by object or

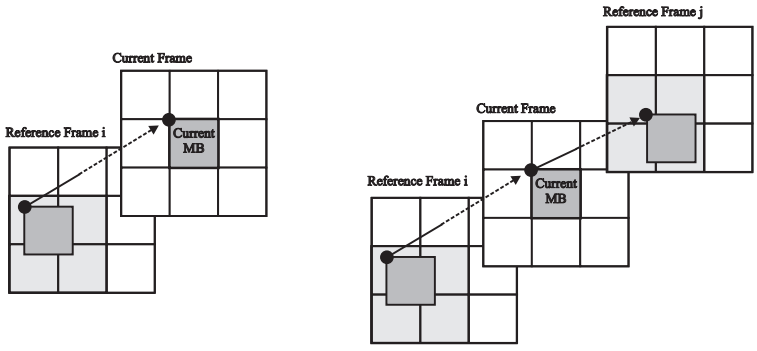


Figure 3.5: Temporal prediction of macroblocks between the current frame and reference frames. (a) Uni-directional temporal prediction between two frames. (b) Bi-directional temporal prediction between the current frame and two reference frames.

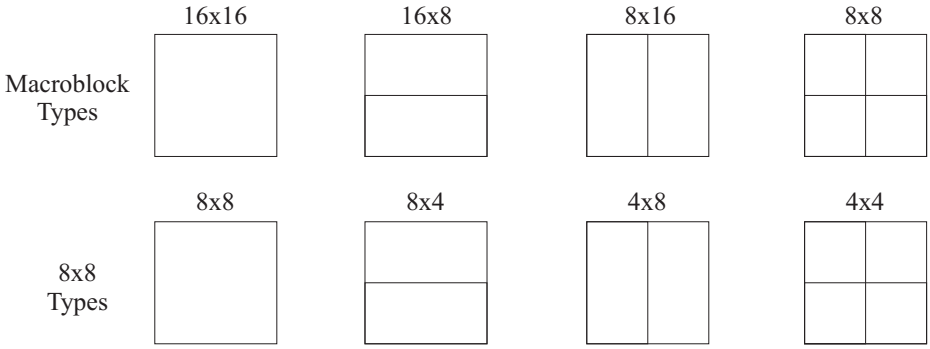


Figure 3.6: For temporal prediction, the macroblocks can be divided into smaller subblocks of 16x8, 8x16 or 8x8 pixels. For macroblocks with an 8x8 partitioning, a further partitioning of each 8x8 block into 8x4, 4x8 or 4x4 blocks is possible.

camera movement and the encoder estimates this movement during MCP. This *inter-prediction* estimates the spatial displacement of each MB between frames and describes this displacement using motion vectors (MVs). This can be seen in Figure 3.5a. Here, a MB region in the current frame is predicted based on a region in a reference frame. Using the motion of a MB between frames can be used for predicting the MB's pixels. The MV is used as part of the inter-coding representation of this MB.

An innovative extension of motion compensated prediction has been *bi-directional prediction*. This coding tool has been available since MPEG-1 and in all succeeding standards and enables the use of prediction signals from past and future frames. Bi-directional motion compensation is visualised in Figure 3.5b. Here, the prediction of a MB in the current frame is based on averaging the MCP signals from previous and future frames.

Newer standards such as H.264 address the obvious fact that motion is typically not the same for all pixels of a MB by introducing *variable block sizes* for motion prediction and compensa-

tion. By using *macroblock partitioning*, a MB can be divided into blocks of 8x8 or even down to 4x4 pixels with individual motion vectors for each block. Figure 3.6 visualises the MB partitions available in H.264. Each MB can divided into smaller blocks of 16x8, 8x16 or 8x8 pixels. Each MB with 8x8 partitioning can be further partitioned into 8x4, 4x8 and 4x4 blocks. The small block sizes aim to describe very fine-structured motion efficiently and can increase the coding efficiency significantly.

Apart from macroblock partitioning to target the motion of fine structures, most VCAs after H.261 support *sub-pixel accurate motion compensation*. The motion of each block is estimated on a full-, half- or quarter-pixel accurate position, which enables a more accurate motion representation and typically results in a higher coding efficiency. While for MPEG-1, MPEG-2 and H.263 full- and half-pixel accuracy was provided, new standards such as H.264 provide up to quarter-pixel precision for MCP.

It should be noted that due to the large amount of temporal redundancy, inter-prediction typically can achieve a higher compression efficiency than spatial prediction but at the cost of higher computational complexity.

Slice types

Hybrid video coding frameworks such as H.264 typically provide three ways for coding the slices of a frame:

1. **Intra-coded slices (I-slices):** In an I-slice, all MBs of the slice are intra-coded. This avoids data dependency on MBs of other slices and enables decoding of MBs within I-Slices independently of other slices. This independence allows the decoder to use I-slices as entry points into a coded bitstream or for recovering from transmission errors.

2. **Uni-directional predicted slices (P-slices):** For predicting blocks in a P-slice, inter-prediction between the current frame and a *reference frame* and based on one MCP signal can be used. In addition, all coding types of I-Slices can be used for coding MBs in P-slices.

3. **Bi-directional predicted slices (B-slices):** In addition to all coding types available in P-slices, inter-prediction based on two MCP signals can be used for predicting MBs in B-slices.

Depending on the coding of the slices within a frame, the frame is referred to as I-, P- or B-frame. In terms of coding efficiency, P-frames typically achieve a reduction of 50% in data rate compared to I-frames. B-frames typically achieve a higher data reduction than P-frames, but at the costs of higher computational complexity.

Each GOP typically contains a set of I-, P- and B-coded frames and can be decoded independently of any previous GOP. The length and the coding structure of a GOP are not specified by the standards and can be chosen by the encoder. The coding structure is characterised by its length and the type of prediction used for the individual frames in the GOP. In Figure 3.7, two different GOP-codings are visualised. Figure 3.7a represents a IP-coded GOP structure with one

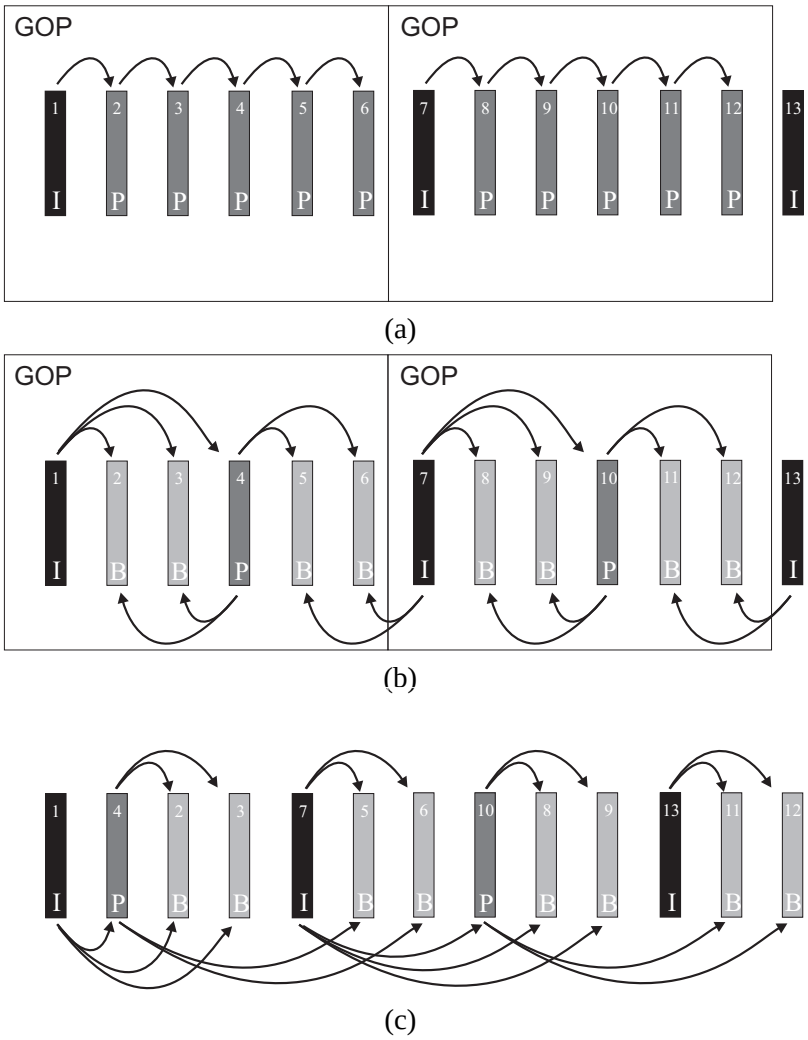


Figure 3.7: Different GOP-codings: Intra (I), predicted (P) and bi-directional predicted (B) frames are used for coding. (a) IP-Coding, (b) IBBP-Coding: View order and (c) IBBP-Coding: Coding order. More details on the coding order are provided in Section 3.4.2.

I-frame at the start of each GOP and dependent P-frames afterwards. Figure 3.7b represents a IBBP-coding where bi-directional B-frames are used within the GOP. It should be noted that in order to use future frames as references in B-frame prediction the coding order must be adapted. In Figure 3.7c, this reordering of the coding order is visualised. Frames that are used for prediction of other frames are encoded earlier in the bitstream. The decoder has to make sure that these frames are encoded/decoded in the correct coding order.

Typically, the GOP-coding structure is chosen according to the video application. Aspects that typically determine the choice of GOP-coding structure are, for example, compression efficiency, quality, latency and error robustness. For editing and cutting applications, I-frame only

coding or IP-coding with short GOP sizes is often used. This provides frequent entry points into the coded bitstreams and enables fast cutting, fast backward/forward and quick previews. For low-bitrate video conferencing applications, typically higher compression efficiency and low latency are targeted. For these applictions, longer IP-coded sequences are often used. When distributing high-resolution video content without the need to meet low latency targets, compression efficient coding orders based on B-frames (e.g. IBBP) are often used.

3.4.3 Transformation and quantisation of residual data

After the prediction, a spectral decomposition of the original residual data is typically done by all hybrid video coding standards. This transforms pixel residuals into frequency components and reduces spatial correlation between pixels. It concentrates relevant information in a few significant transform coefficients that can be represented by a few variables and stored efficiently. In H.264 spatial transform coding of the residuals is used. In contrast to preceeding coding standards that have been using 8x8 block transforms, the transformation in H.264 is done on 4x4 blocks. For this transformation, H.264 uses a 2D DCT (*2D discrete cosine transform*) of the following form:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \tag{3.1}$$

It should be noted that H.264 uses this exact *integer transform* H for the transformation of 4x4 pixel blocks. This transform has the significant advantage that inverse-transformation mismatch can be avoided. Furthermore, the transformation can be computed using only a combination of additions, substractions and bit-shifting. These operations can typcially be implemented efficiently on most hardware architectures.

For quantisation of the resulting transform coefficients, H.264 uses a quantisation parameter (QP) with values ranging from 0 to 51. QP controls the quantisation steps and has been designed in a way that an increase of QP by a factor of six results in approximately the doubling of the bit rate. The rate control of the encoder selects the QP in such a way that the targeted bitrates can be achieved.

3.4.4 Deblocking filter

The block-based coding of video content results in visible artifacts at the block boundaries. The main causes are the block-based MCP and the block transforms, and especially for low-bitrate applications these artifacts become obvious. H.264 introduces an adaptive in-loop deblocking filter that reduces blocking artifacts while retaining the sharpness of true edges. For true edges, the filtering would be turned off while artifical edges should be filtered.

For H.264, the decision whether filtering should be done and which filtering strength is applied is based on the coding mode of the filtered blocks (e.g. prediction type, number of residuals, motion strength) and on the pixel samples that are located at the filtering position. In

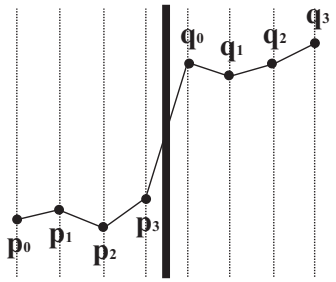


Figure 3.8: One-dimensional visualisation of a block edge in a typical situation where the H.264 deblocking filter would be turned on [LJL+03].

Figure 3.8, this sample-based filtering is visualised. This one-dimensional visualisation represents the filtering process across a border of two neighbouring blocks. The border is located between q_0 and p_0 and four pixels from the left block and four pixels from the right block are used by the filter for determining the filtering strength. This filtering is done on the vertical as well as the horizontal borders of each MB.

In [LJL+03], a detailed description of this filter can be found. In the context of parallel video coding it is important to note that the H.264 deblocking filter introduces dependencies between MBs. Furthermore, the filter is applied after the MCP of a MB and across slice boundaries. This can generate dependencies between MBs at the border of I-slices and must be considered in parallel decoding schemes.

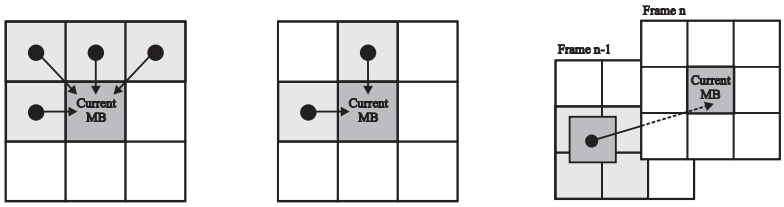
3.4.5 Entropy coding

Entropy coding schemes such as Variable-Length Coding (VLC) take advantage of the relative probabilities of the possible values within our coded video representation and reduce statistical redundancy. In H.264, two entropy coding algorithms have been included: Context-Adaptive Variable Length Coding (CAVLC) and Context-Adaptive Binary Arithmetic Coding (CABAC).

The coding of the syntax elements using the CABAC arithmetic coding scheme is typically 10-15% more efficient compared to CAVLC [SW05] but computationally more demanding as well. Both algorithms adapt to the data statistics using either context-adaptive mapping to different VLC tables (CAVLC) or adjusting of probability estimates in a context-adaptive way (CABAC).

3.5 Parallel video decoding

Despite the fact that parallel H.264 decoding has been investigated in a large number of scientific publications [HJKH03, LHH03, SYT04], parallelisation of this algorithm is highly challenging. Most of the H.264 coding tools strongly adapt to the video content and come at the cost of strong variations in the runtime of the decoder [HJKH03, SBSG08]. Furthermore, the coding tools in H.264 introduce a large number of data-dependencies between the individual VCL coding elements. For an efficient parallelisation as well as when predicting the runtime behaviour of a



(a) (b) (c)

Figure 3.9: Macroblock dependencies in H.264 decoding. Arrows mean that the macroblock at the origin of the arrow needs to be processed before decoding the other macroblock. (a) Intra-prediction dependency. (b) Deblocking filter dependency. (c) Inter-prediction dependency.

parallel H.264 encoder or decoder, these runtime variations and dependencies must be resolved appropriately.

In previous work, various approaches for parallelizing the H.264 decoding process have been introduced. Van der Tol et al. [vJG03] have investigated methods for mapping the H.264 decoding process onto multiple cores. Functional splitting of an H.264 decoder and the use of MB pipelining at thread-level have been demonstrated in [CHC+05, CTGG04, SFLB07]. Zhao et al. [ZL06] exploit frame parallelism in the Wavefront technique. A hierarchical approach working at group-of-picture and frame level is demonstrated in [RGM06]. In [LLCW10], a parallel embedded H.264 decoder processes the video stream on a slice-level. The scalability of H.264 for a data-parallel decoding approach operating on the MB-level and on multiple frames in parallel has been investigated by Meenderinck et al. [MAJ+09]. The same study introduces an efficient technique for H.264 frame-level parallelisation, the 3D-Wave strategy.

These papers primarily focus on parallelisation in terms of algorithmic scalability. Upper limits on the number of processors and frames processed in parallel are given. However, the memory-restrictions of embedded environments make these approaches hardly usable for mobile and embedded architectures. More resource-efficient H.264 splitting approaches have been introduced in [SSBG08, SWC07, WPH+03]. The focus of these authors has been put on efficient decoder implementations for embedded architectures.

In the following, we investigate the dependencies that video coding tools generate between MBs. This has a major impact on all parallelisation approaches. After this section, we describe the two fundamental paradigms that parallel video decoder approaches can be based on: Functional- and data-parallel partitioning.

3.5.1 Dependencies between macroblocks

Partitioning of a video decoder and distributing the MBs' coding tasks onto multiple PUs is challenging due to dependencies between spatially as well as temporally neighbouring MBs. These dependencies originate from three sources as illustrated in Figure 3.9, and are described as follows.

Firstly, in spatial prediction of the current MB, unfiltered pixel information from up to four spatially neighbouring MBs is used. These dependencies are depicted in Figure 3.9a. In gen-

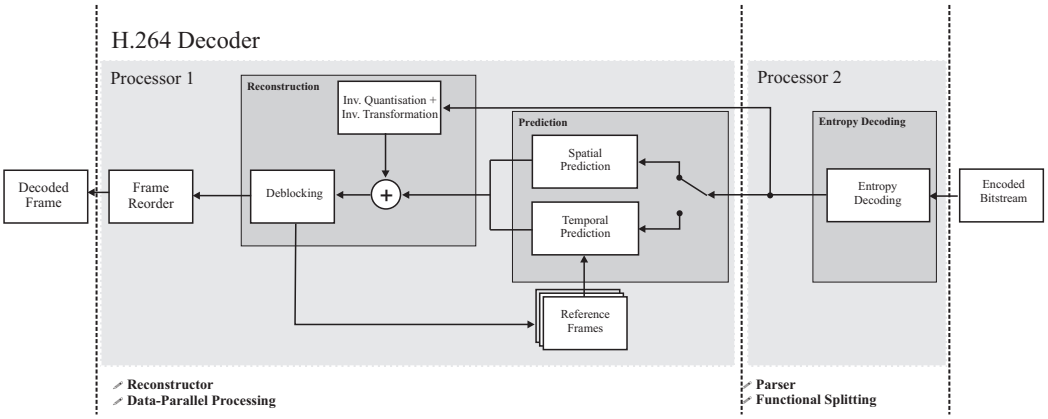


Figure 3.10: Functional split of an H.264 decoder: Parsing and entropy decoding tasks are executed on one PU, pixel-based processing tasks such as prediction and deblocking on another PU.

eral, it is a good option to gather the current MB and its reference MBs on the same PU to avoid expensive inter-processor communication for resolving this dependency. For an efficient parallelisation, this dependency must be addressed carefully.

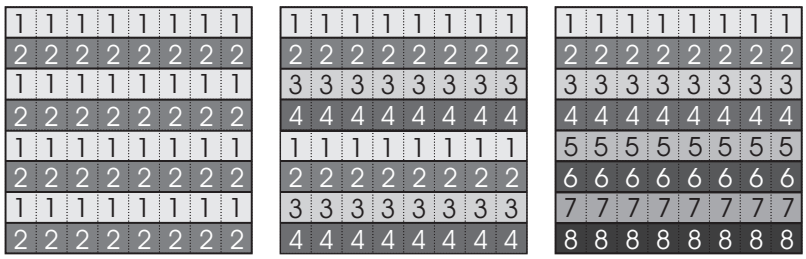
Secondly, the deblocking filter imposes additional spatial dependencies. For filtering the outer edges of the current MB, up to four pixel rows/columns from the upper and left neighbouring MBs are used as filter input. These MB dependencies are visualised in Figure 3.9b. An efficient parallelisation method will focus on avoiding these dependencies having to be resolved across individual processors.

The third MB dependency arises from the inter-prediction. The inter-prediction reads reference data from MBs of previously decoded frames. Obviously, it is required that processing of these reference MBs has already been completed before they can be used for inter-prediction of the current MB. This results in the temporal dependency depicted in Figure 3.9c. In fact, the current MB can depend on a rather large number of reference MBs. H.264 allows splitting of the current MB into small sub-blocks, for each of which a separate motion vector is computed. In P-slices, each inter-coded MB can contain up to 16 motion vectors and point to one reference frame. For bi-directionally predicted MBs in B-slices, a maximum of 32 motion vectors and two reference frames is possible.

3.5.2 Functional partitioning

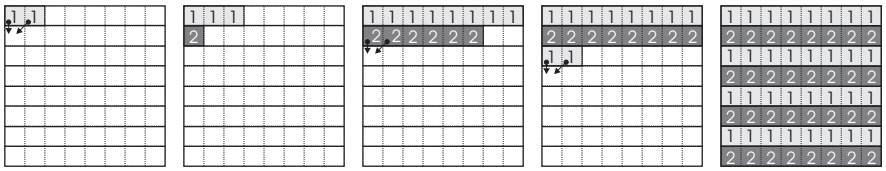
In functionally partitioned decoding systems, the decoding tasks such as parsing, motion compensation or deblocking are executed on individual PUs. Typically, the individual coding tasks of each MB are processed by one processor after the other. The multiple PUs allow the next MB's decoding tasks to be started before computation of the current MB has finished.

This splitting method has the advantage that each PU can be optimised for a certain task (e.g. by adding task-specific hardware extensions) and minimal-sized instruction caches. In contrast to data-dependent parallelisation, also strongly sequential tasks such as entropy decoding can be



(2 Cores) (4 Cores) (8 Cores)

Figure 3.11: The Single-row splitting approach. The assignment of processors to macroblocks is shown.



($t = 2$) ($t = 3$) ($t = 8$) ($t = 10$) ($t = 34$)

Figure 3.12: Example of the Single-row splitting approach used with two cores. Processed macroblocks are shown at different instances of time t. It takes a constant value of 1 unit of time to process a macroblock.

accelerated by this strategy. The disadvantages are typically an unequal workload balancing and high transfer rates for inter-communication. Figure 3.10 visualises a functionally split H.264 decoder. The parsing and entropy decoding task has been assigned to Processor 2, the pixel-based processing tasks to Processor 1.

3.5.3 Data-parallel partitioning

As opposed to functional splitting methods, data-parallel systems do not distribute the functions, but the macroblocks among multiple PUs. Figure 3.11 illustrates an example of data-parallel stream decoding. This splitting strategy distributes horizontal lines of macroblocks among different processors. For efficient parallelisation, the MBs' core assignment algorithm has to address the following issues:

- The data-dependencies between different PUs must be minimised and data locality must be exploited (i.e. supporting of caching strategies).
- The MB distribution onto the PUs must achieve an equal workload balancing.
- Generic MB core assignment for different frame sizes must be possible.

Scalability in parallel systems requires minimal data-dependency between the PUs. A compromise between small memory size and data-dependencies can be reached by grouping the MBs as described in [vJG03]. To support caching strategies at a more global scale, the groups

of MBs assigned to one core must be aligned closely together in each frame. By introducing a centralised and constant MB assignment for each frame, this global caching issue can be addressed efficiently. Additionally, a constant MB assignment allows the parsed MBs to be written directly to the First-In-First-Out (FIFO) input buffers of the PUs executing the corresponding reconstruction tasks.

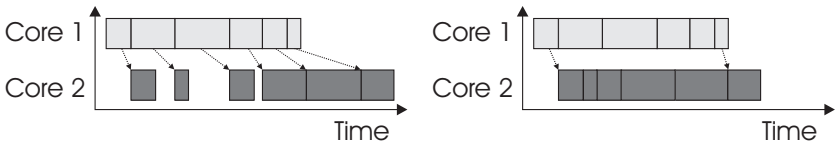
However, introducing data-independencies for data-parallel processing support has its limitations. First, increasing the number of independent data-blocks reduces the coding efficiency since similarities between blocks are not exploited. Second, the encoder does not necessarily use multiple slices for coding a frame and the availability of slices typically cannot be guaranteed at the decoder side. Methods for data-parallel processing have been introduced that typically do not need data-independent blocks. These methods specify a processing order for the data-blocks that tries to minimize the dependencies between consecutively processed data. An example is the *wavefront* method [MAJ$^+$09, SSBG11, SSBG09]. Here, the data is divided into multiple sets of macroblocks where each set is called a *wave*. Data-dependencies only exist between blocks from different waves but not between blocks of the same wave. Consequently, blocks in a wave can be processed in parallel since they only depend on data from previously processed waves.

In the following, we provide examples of the most commonly used data-parallel decoding approaches.

Single-row approach

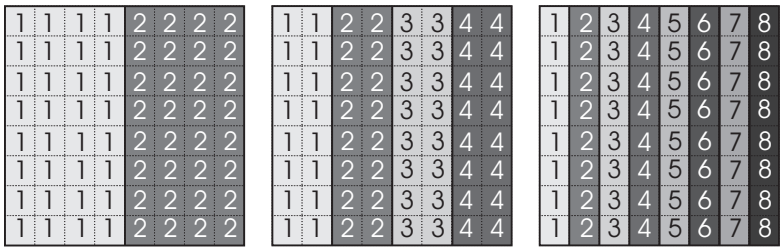
To illustrate the Single-row (SR) approach, we give an example with two processors on an image divided into 8×8 MBs in Figure 3.12. Let N be the number of processors. Processor $i \in \{0, \cdots, N-1\}$ is then responsible for decoding the yth row of MBs if $y \ mod \ N = i$. In this example, we assume that it takes a constant value of 1 unit of time to process each MB. It is, however, important to notice that this is a coarse oversimplification. In real video streams, there are large variations in the processing times of individual MBs, which makes it difficult to evaluate the effectivness of a parallelisation approach. In Figure 3.12, only PU 1 is able to decode MBs at time $t = 2$, since all other MBs are blocked as a consequence of the dependencies illustrated in Figure 3.9. After the first two MBs of Row 1 have been computed, the second core can start processing the first MB of the second row, since the dependencies for this MB are now resolved ($t = 3$). The next interesting event occurs at $t = 8$ when PU 1 has finished the computation of the first row. MBs of the second row have already been computed and therefore PU 1 can start decoding MBs of Row 3 that are dependent on their upper neighbours. At time $t = 10$ we obtain the same situation as at $t = 2$, where the first core unlocks the second one. Finally, the whole frame has been decoded at $t = 34$.

The advantage of the Single-row approach lies in its simplicity. It is very easy to split the frame among the individual processors. There is only a small start delay after which all processors can effectively work. The potential downside of this approach is that there are many dependencies that need to be resolved across processor assignment borders. This has played no role in our example where we have assumed constant processing time for each MB. It, however, will be noticeable for real videos streams that contain MBs of considerably different runtime characteristics. In fact, each MB processed by core i depends on its upper neighbours that are processed by a different PU $i-1$. If PU $i-1$ fails to deliver these MBs at the right time, this



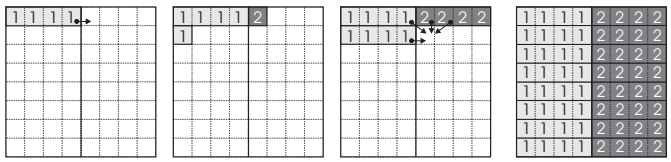
(a) (b)

Figure 3.13: The number of inter-processor dependencies is crucial for the overall performance of the multi-core system. Rectangles represent MBs. A MB's width indicates the required processing time. Arrows between two MBs mean that processing of the MB which the arrow points to can only start after the other MB has been decoded. (a) A large number of inter-processor dependencies slows down the system. (b) Due to the low amount of inter-processor dependencies, different running times of individual MBs become averaged out. This should improve the overall performance.



(2 Cores) (4 Cores) (8 Cores)

Figure 3.14: The Multi-column splitting approach.



($t = 4$) ($t = 5$) ($t = 8$) ($t = 36$)

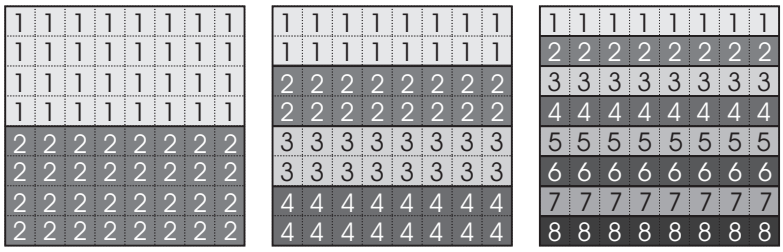
Figure 3.15: Example of the Multi-column splitting approach.

will immediately produce stalls at PU i. This behaviour is shown in Figure 3.13(a). On the other hand, this strong coupling of PUs can potentially lead to low buffer requirements.

Multi-column approach

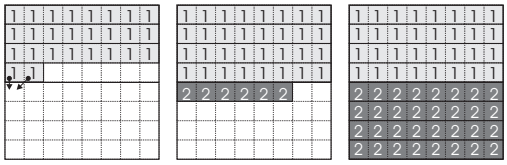
The Multi-column (MC) splitting strategy divides the frame into equally-sized columns as shown in Figure 3.14. Let w denote the width of a multi-column that is typically derived by dividing the number of MBs in a row by the number of processors. More formally, let N be the number of processors. Processor i is then responsible for decoding a MB of the xth column if $iw \leq x < (i+1)w$. A similar method to partition the image has recently been proposed for the H.264 encoder in [SWC07].

To illustrate the MC approach, we give an example with two processors on an image divided



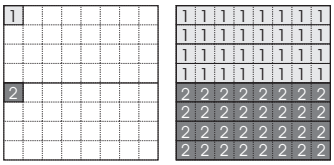
(2 Cores) (4 Cores) (8 Cores)

Figure 3.16: The Slice-parallel splitting approach.



($t = 26$) ($t = 32$) ($t = 58$)

Figure 3.17: Example of the Slice-parallel splitting approach in the blocking version.



($t = 1$) ($t = 32$)

Figure 3.18: Example of the Slice-parallel splitting approach in the non-blocking version.

into 8×8 MBs in Figure 3.15. In this example, we assume that it takes a constant value of 1 unit of time to process each MB. Processor 1 thereby starts processing the first row of MBs until it hits the border to the MBs assigned to Processor 2 ($t = 4$). Since the dependency for the leftmost MB of PU 2 is now resolved, Processor 2 can finish decoding its first MB at $t = 5$. We obtain a similar situation at $t = 8$. The dependencies of the leftmost MB of the second row have been resolved, and PU 2 can therefore continue its work. Decoding of the frame is finally completed at $t = 36$.

The basic idea behind using the Multi-column approach is to obtain a looser coupling of processor dependencies. In fact, the processor assignment borders are significantly reduced in comparison to the Single-row approach. One processor has to wait for the results of another one only at the boundary of its multi-column. Within the multi-column, MB dependencies can be resolved on the same processor. This should lead to reduced inter-processor dependencies and could therefore improve the overall runtime behaviour of the multi-core system as is depicted in Figure 3.13(b).

Slice-parallel approach

The Slice-parallel (SP) is a widely-used splitting approach. It is a 90-degree rotated version of the Multi-column approach that divides the frame into even-sized rows. This method is depicted in Figure 3.16. Formally spoken, let h denote the height of a multi-row. A MB of the yth row is then assigned to PU i if $ih \leq y < (i+1)h$.

The runtime behaviour of the SP approach is illustrated in Figure 3.17. Here, PU 2 has to wait for a relatively long time ($t = 26$) until the dependencies for its first assigned MB are resolved. While the first processor can complete its work on the current frame at $t = 32$, it still takes 26 units of time until the second PU finishes processing the remainder of the frame at $t = 58$. In the following, we refer to this approach as the blocking Slice-parallel technique.

In a recent work [MM08], a non-blocking encoder version of the SP approach has been presented. The authors encode their video streams so that slice borders coincide with horizontal lines in the frames. Since neither dependencies introduced by intra-prediction nor dependencies introduced by the deblocking filter occur across slice borders, the multi-rows can be processed independently from each other.

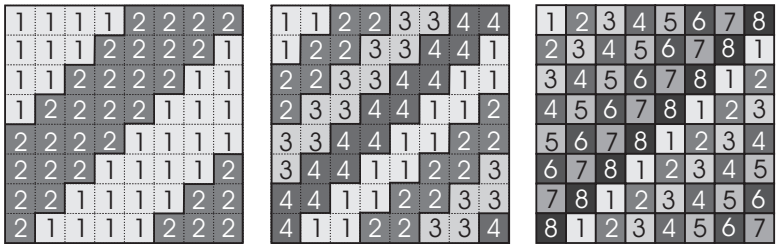
Obviously, this non-blocking SP approach (NBSP) requires having full control over the encoder, which will not be the case for many applications. For completeness, we also give an example of this approach in Figure 3.18. Here, both PUs can start processing their assigned MBs immediately ($t = 1$) and finish decoding the complete frame at $t = 32$.

Diagonal approach

The Diagonal (DG) approach depicted in Figure 3.19 represents another popular splitting method. This processor assignment is obtained by dividing the first line of MBs into equally-sized columns. The assignments for the subsequent lines are then derived by left-shifting the MB assignments of the line above. This procedure leads to diagonal patterns.

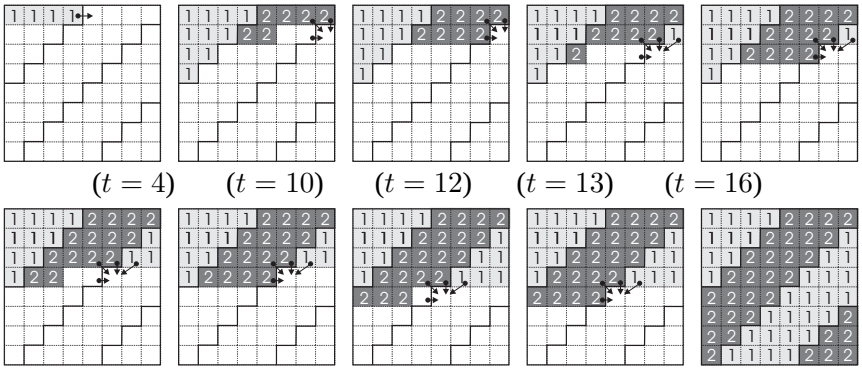
Figure 3.20 gives an example of the DG approach using two processors. Here, the second PU stalls until its dependencies become resolved by PU 1 at $t = 4$. The first PU completes computation of its first image partition at $t = 10$. Unfortunately, it cannot directly start processing the second partition, but has to wait for PU 2 to resolve dependencies until $t = 12$. The following images ($t = \{13, 16, 18, 20, 23, 24\}$) show situations where the first PU has to wait for MBs decoded by PU 2. For legibility, we do not show subsequent states where one processor blocks the other one, but directly proceed to the final result derived at $t = 43$.

The Diagonal splitting method is regarded as an approach that "respects" the dependency patterns spanned by the intra-prediction and the deblocking filter. (Dependencies are shown in Figure 3.9). We illustrate the idea behind the Diagonal splitting method in Figure 3.21. The figure compares the inter-processor dependencies introduced by the Diagonal and the Multi-column splitting techniques. The Diagonal method therefore only shows dependencies on MBs from its left neighbouring processor, which is in contrast to, for example, the Multi-column method that contains dependencies on MBs of both neighbouring PUs. These reduced inter-processor dependencies could lead to an improved runtime behaviour of the multi-core system.



(2 Cores) (4 Cores) (8 Cores)

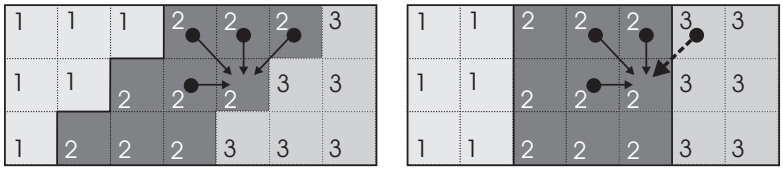
Figure 3.19: The Diagonal splitting approach.



($t = 4$) ($t = 10$) ($t = 12$) ($t = 13$) ($t = 16$)

($t = 18$) ($t = 20$) ($t = 23$) ($t = 24$) ($t = 43$)

Figure 3.20: Example of the Diagonal splitting approach.



(a) (b)

Figure 3.21: Processor dependencies in the Diagonal and Multi-column splitting approaches. (a) In the Diagonal method, dependencies for PU 2 originate solely from MBs assigned to PU 1. PU 2 therefore never has to wait for PU 3. (b) In the Multi-column approach, MBs assigned to PU 2 are also dependent on Processor 3 as indicated by the dotted arrow.

3.6 Summary

In this chapter, we have described the historical developments of hybrid video coding standards. We have shown that modern hybrid video coding standards such as MPEG-4/ASP, H.264 and VC-1 share strong structural similarities and conceptually similar coding tools. Furthermore, most hybrid video standards use similar hierarchical coding elements and VCL definitions for representing video content. Understanding and exploiting the fundamental architectural of modern video coding standards plays an essential role when doing VCA runtime analysis. It can

provide important insights into the runtime behaviour of a VCA at the level where the parallelisation would take place in a functional or data-parallel partitioning. In Section 4, we will exploit this for analysing the runtime performance of a VCA in relation to its underlying algorithm structure. Section 5 will introduce a framework for describing VCAs in an abstract way based on the structural similarity of hybrid video coding.

In addition, the current section has described splitting approaches for distributing H.264 decoding tasks onto multiple PUs. We have introduced various functional- as well as data-parallel splitting approaches and outlined the challenges of resolving data-dependencies efficiently in parallel decoder designs. The runtime performance of the individual VCA partitionings will strongly vary for different architecture hardware (e.g. number and type of processors, transfer buffer sizes) and the software implementation of the VCA's functional blocks. In Section 6, we will introduce techniques for quickly estimating the runtime of different VCA partitionings at an early stage of the design process. We will use several of the VCA partitioning approaches described in the current section in order to demonstrate the proposed techniques.

CHAPTER 4

Data-driven runtime analysis

In this section, we analyse the runtime behaviour of an existing H.264 decoder running on a *single* core. By profiling a single-core VCA, important insights for runtime optimisation can be obtained. Starting from an initial VCA, the developer iteratively profiles the VCA and optimises the functions with the highest potential for a runtime reduction. While this optimisation on a functional level is highly efficient for runtime optimisations of single-core VCAs, exploiting the obtained information when designing a parallel VCA is not straightforward. To address this stortcoming of functional profiling, we will introduce a new analysis technique, the Data-Driven Profiling (DDP) method. This method represents one key contribution of this thesis and puts traditional function profilings in the context with the actual VCL coding elements that are processed during a VCA's execution. The complexity information derived by DDP provides insights into data-parallel complexity aspects of a VCA and can serve as an efficient tool during parallel VCA design. Compared to an analysis on a functional level, it provides capabilities to investigate the complexity of a VCA at the level where the parallelization takes place (i.e. the macroblock level for most VCAs).

The chapter is structured in the following way: In Section 4.1, we introduce the concept behind DDP and explain how traditional function traces can be mapped onto the coding elements specified by the H.264 coding standard. Section 4.2 describes how DDPs can be derived in an automatic way. In Section 4.3, we provide an overview of the test environment used in context of this thesis. We apply the DDP method in Section 4.4 and demonstrate how data-driven profiles can be analysed before starting a parallel VCA design.

4.1 Data-driven profiling

Traditional dynamic profiles contain information about a VCA's complexity at a function level (Chapter 2.2). For each function of the VCA, a complexity statistic based on different metrics is provided. For example, the mean and average number of processing cycles spent in a specific function. This information is typically sufficient for single-core optimisation.

The idea behind DDP is to map the function traces obtained during a VCA's execution onto the VCL coding elements. These elements represent the processing levels where the system designer can introduce parallel processing mechanisms when designing the parallel VCA. In contrast to a traditional profiling, the VCA runtime can then be analysed directly at this level. In this work, we investigate VCA processing at MB-level since most parallel VCA approaches implement partitioning schemes at this level. However, the methods introduced in this work are not restricted to MB-level analysis. Depending on the granularity that is most suitable for analysing a VCA's dynamic behaviour and integrating parallelisation mechanisms, different VCL coding elements such as slices or frames can be used.

A first step towards macroblock-based profiling of an H.264 decoder has been taken in [KF05]. The authors determine the frequency of each macroblock coding type when decoding a video stream. The authors interrelate the runtime complexity with the macroblock frequency. However, complexity is only investigated at the frame level. Our work extends this idea and provides a generic framework for a detailed complexity analysis on all hierarchical levels of a VCL (e.g. MB-, slices-, frame-level). This is essential for making assumptions about parallel VCA implementations.

This step towards a more data-focused representation of traditional profilings has numerous advantages: First, the partitioning of the VCA and the distribution of the workload to multiple processors take place at this level. Runtime effects resulting from varying workload and intercommunication between PUs must be analysed at this processing level. By carefully addressing these variations (i.e. by introducing buffers between the processors), we can distribute the data in a way that results in an efficient usage of the PUs.

Second, by analysing a VCA at VCL-level, complexity can be assigned to specific positions within a frame and to frames in the video. This enables the system designer to interrelate complexity with spatial and temporal positions within a video. Based on this information, efficient load-balancing methods for data-parallel coding solutions where multiple video regions and frames are processed in parallel could be developed.

Figure 4.1 visualises this concept for a VCA which processes the video data one MB after another. Function calls that occur during the processing of a macroblock MB_i can be assigned to the VCL coding elements (i.e. VCL mapping) within the video stream such as MBs, slices and frames.

Apart from assigning function calls to specific VCL coding elements, the system designer can define Functional Blocks (FBs) that represent major coding tasks. Each function is assigned to a single FB FB_j (i.e. mapping to FBs). For example in Figure 4.1, we have chosen 4 FBs ($parse, predict, IDCT, deblock$) that divide the H.264 decoding process into 4 functional blocks. In the following, we will use the term *task* to refer to the execution of a specific FB of a MB. Each task has a unique number (task ID) and represents the execution of all function calls that have been assigned to this MB's FB. The computational complexity (e.g. the number of cycles) of a task is the cumulated complexity of all function calls that have been assigned to this task.

This mapping of function calls to the individual MBs and the classification into FBs can be done easily for simple VCA structures. For example, let us assume that only a single function call *IntraPrediction* represents the complete intra prediction FB of each macroblock and that this

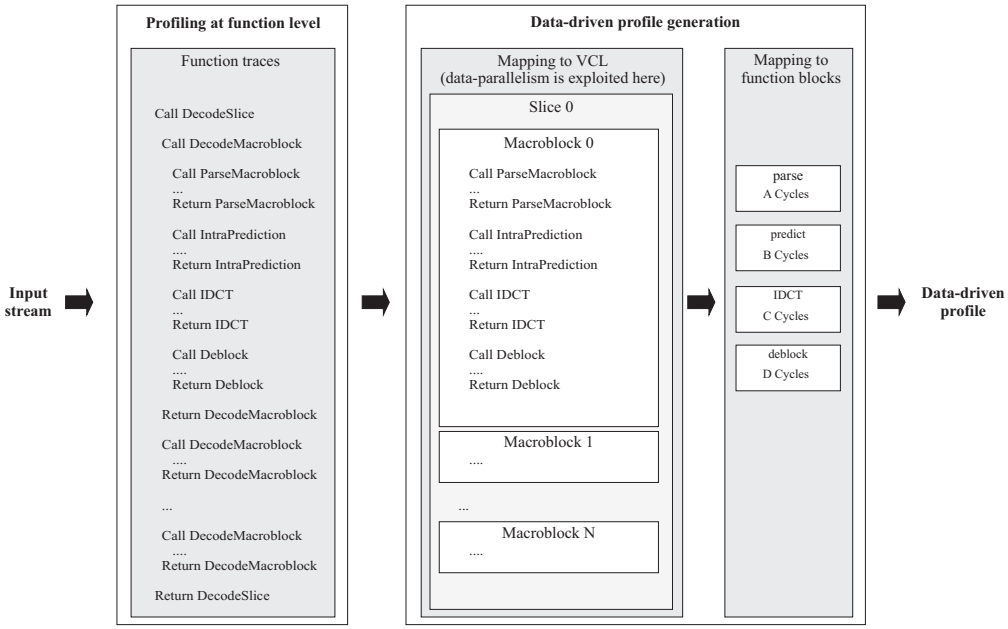


Figure 4.1: Data-driven profiling: The function traces are assigned to VCL coding elements and further onto functional blocks. In this figure, four FBs have been defined: Parse, predict, IDCT and deblock.

function is called exactly once per MB. If this function is called for every MB, then the processing time for the third invocation of the function *IntraPrediction* will consequently represent the time that MB 3 spends for the FB "intra". Table 4.1 provides an example of a DDP at MB level. The function trace information that has been gathered during a traditional dynamic profiling is mapped onto individual VCL coding elements, decoding FBs and tasks. Each task specifies the complexity of a MB's FB.

However, for more complex VCAs, mapping of the function traces to individual FBs and VCL coding elements is not straightforward. Each FB can consist of a large number of calls to individual program functions. Each function can be called multiple times, from within different FBs (i.e. the same function is used by multiple FBs) and at different hierarchy and recursion levels. Furthermore, the FBs of a VCL coding element do not necessarily occur sequentially. For example, in many decoder implementations, the deblocking of a frame is done after all the frame's MBs have been reconstructed. This makes it challenging to determine the MB a function call belongs to and, consequently, which task it should be assigned to. We will address this issue in the next section where we introduce a method for assigning function traces to the specific FBs and MB in an automatic way. This enables the generation of DDPs for highly complex VCAs.

4.2 Automatic generation of data-driven profiles

In this section, we will introduce a method for mapping the function traces to the FBs of a VCA's individual MBs. This method is conceptually similar to the idea of pushdown automa-

Table 4.1: Profile for the tasks of the decoder's individual functional blocks: The table visualises the extracted computation time in cycles for each MB and functional block executed during the VCA's execution. The execution of a MB's FB is represented by a task with a unique task ID.

Task	**#MB**	**FB**	**Complexity**
T_1	0	parse	8730
T_2	0	IDCT	141
T_3	0	intra	1057
T_4	0	deblock	8711
T_5	1	parse	12463
T_6	1	IDCT	510
T_7	1	intra	701
T_8	1	deblock	15734
T_9	2	parse	19122
T_{10}	2	IDCT	110
T_{11}	2	intra	1418
T_{12}	2	deblock	13875
..	..	..	..

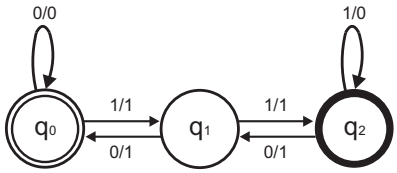


Figure 4.2: A state transition diagram of a simple state machine with an initial state q_0 and a final state q_2. It visualises a state machine consisting of three states and the state transitions. Each transition has input/output events. The input events determine the conditions that have to be met for a transition to another state. The output signals specify actions caused by this transition.

tons (PDAs) [Sip97]. PDAs are abstract machines that can describe sequential behaviour in an formal way. This provides a powerful and intuitive concept to describe the VCL coding element structure and the FBs of a VCA in a formal way. After a brief introduction to Finite State Machines (FSMs) and PDAs, we will explain how we can describe complex VCA structures and exploit these formal descriptions for automatic mapping of function traces to specific VCL coding elements and FBs.

4.2.1 Finite State Machines and Pushdown Automatons

State machines have been used extensively in software design, for example for studying problems of algorithm computability [WSWW06] and for developing event-driven software design approaches [WWW04]. State machines are formal models for describing sequential behaviour in an abstract way. The basic concept of a state machine has been introduced in [Mea55, Moo56].

Table 4.2: Transition table of the state machine visualised in Figure 4.2. For each control state of the machine, the table specifies transition rules (one rule per table row). In this example, the machine consists of three states q_0, q_1 and q_2. Each rule specifies which input transfers the machine from the current state into another state. Furthermore, the rule determines the output caused by this transition.

Condition		**Effect**	
Current state	*Input*	*Next state*	*Output*
q_0	0	q_0	0
q_0	1	q_1	1
q_1	0	q_0	1
q_1	1	q_2	1
q_2	0	q_1	1
q_2	1	q_2	0

State machines assume that a system is in one of multiple possible *states* and that the conditions for changing into another state can be expressed in a formal way. Starting from an *initial state*, the fulfilling of certain conditions triggers the transition to another state (i.e. *state transition*). Each condition is associated with an input event. A condition is met when this input event occurs. If there is only one possible transition for each input event of a state, the state machine is called *deterministic*.

Figure 4.2 visualises a simple state machine using a *state transition diagram*. Such a state machine with a finite number of states is called a FSM. For each state, specific input events can trigger transitions from the current state to another state of the FSM. In addition, for each state transition, output events can be specified.

Another common way to describe FSMs is a *transition table*. Table 4.2 defines the transition table for the state machine of Figure 4.2. A transition table contains all states $q_{1..|Q|} \in Q$ of the FSM. For each state q_i, the possible input events $x_{1..|X_i|} \in X_i$ are defined. Each input event x_j results in a transition $y_j : q_i \rightarrow q_{i+1}$ from the current state q_i to another state q_{i+1}. Note, that for simplicity only simple input events have been used in Table 4.2. In practice, also complex input events triggered by multiple conditions are possible.

In comparison to FSMs, a PDA employs a stack in addition to states and represents a more powerful type of abstract machine than FSMs. The PDA can push/pop tokens to/from this stack. Compared to FSMs, this stack enables PDAs to exploit information from previous states since each transition condition can consider the current input event, the current state and tokens on the stack.

4.2.2 Mapping profiling information to VCL and functional blocks

In order to process function traces in an automatic way, a formal description of the profiling data is necessary. We use an abstract machine with multiple states for representing the structure of a VCA and a call/return stack. Each state either represents the processing of a VCL coding

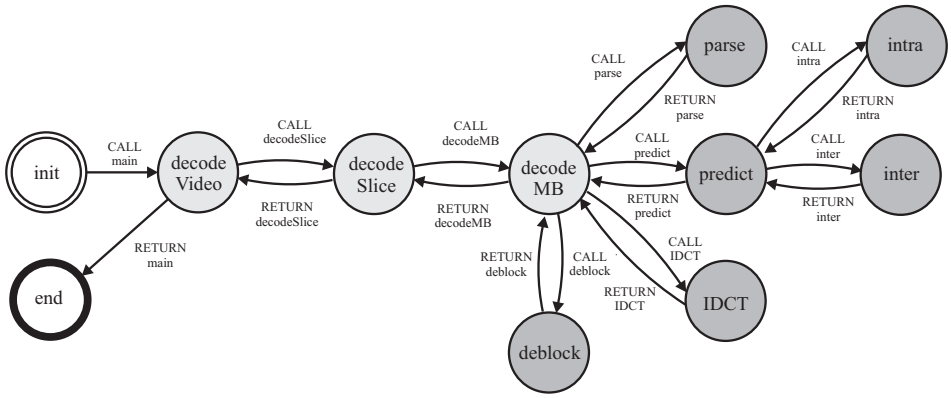


Figure 4.3: An example of a state transition diagram for an H.264 decoder. The states represent the processing of individual layers of the VCL (bright grey) and FBs (dark grey). Function calls and returns can trigger transitions between states. Furthermore, a call/return stack enables us to consider also information from previous states in transition conditions. Note that this is a simplified version of an H.264 decoder. Typically, the state machine contains individual states for the different macroblock types such as intra- and inter-coded macroblocks. This allows us to distinguish between the individual decoding functions of different macroblock types.

element or of one of its FBs. Every time this VCL coding element or this FB is processed during the VCA's execution, the state machine goes into the corresponding state. Figure 4.3 visualises the state transition diagram of an H.264 decoder implementation. The state machine consists of initial/end states, states for representing the decoding process of VCL coding elements (bright grey) and states for representing the FBs of each VCL coding element (dark grey). Note that the designer can choose VCL elements and FBs of the H.264 decoder that he considers most suitable for his VCA parallelisation strategy. We use the functional traces for defining the input events. Depending on the current state, the occurence of a specific function call/return and the tokens on the stack can trigger the transition to another state. For simplicity, in this example a transition from one state to another state can only be triggered by a single input event. In practice, more complex VCA structures with function sharing between FBs, nested function calls, recursions and complex function hierarchies can be addressed by this method. The usage of a call/return stack and transition condition that can access all tokens on the stack enables us to determine state transitions in a deterministic way.

For each state transition, the specific time when this transition occurred can be retrieved from the function traces. Whenever a return state transition to a previous state is invoked in a state q_i, the time difference between entering the state q_i and leaving it can be retrieved. This time represents the processing time that can be assigned to a particular task using the assignment described before. The total duration of a task can be retrieved by summing the times of all function calls that are associated with this task. We refer to this cumulative time as the task's *complexity* (Column 6 in Table 4.3).

Apart from assigning function traces to FBs, we can map FBs to individual VCL coding elements. Note that in the state machine in Figure 4.3, also states for the decoding of individual

Table 4.3: Macroblock-based H.264 profile information: The table visualises the information that has been extracted from a function trace. Each function call retrieved in the function trace is assigned to a task. Each task represents a unique FB of a MB. By summing up the complexity of all function calls assigned to a task, its complexity can be retrieved. Furthermore, coding information such as the coding type and mode for each MB can be extracted from the function traces and via instrumentation.

Trace information		**Macroblock-assigned profiling information**						
Function call	**Instr.**	**Task**	**#MB**	**FB**	**Complexity**	**MB Coding Information**		
					in Cycles	*Type*	*Mode*	..
parse(..)	-	T_1	0	parse	8730	-	-	..
IDCT(..)	-	T_2	0	IDCT	141	-	-	..
intra16x16(..)	0	T_3	0	intra	1057	intra16x16	0	..
deblock(..)	-	T_4	0	deblock	8711	-	-	..
parse(..)	-	T_5	1	parse	12463	-	-	..
IDCT()	-	T_6	1	IDCT	510	-	-	..
intra4x4(..)	1	T_7	1	intra	701	intra4x4	1	..
deblock(..)	-	T_8	1	deblock	15734	-	-	..
parse(..)	-	T_9	2	parse	19122	-	-	..
IDCT()	-	T_{10}	2	IDCT	110	-	-	..
intra16x16(..)	0	T_{11}	2	intra	1418	intra16x16	0	..
deblock(..)	-	T_{12}	2	deblock	13875	-	-	..
..	..	..	..	..	..	..	..	..

macroblocks and slices have been introduced. This enables us to determine how often the state machine has been in the MB decoding state. Every time this state is entered, a counter specifying the current MB number is updated. This enables us to assign each function call to a specific MB and all VCL coding elements which this MB is part of.

4.2.3 Extraction of coding information via function names

In addition to retrieving complexity information, we can also retrieve more VCA-specific information and assign it to the VCL coding elements and decoding tasks. For example, if individual function names indicate the type of prediction used for decoding a MB, these function names can be used for determining each MB's coding type (e.g. a function "decode_intra16x16" for MBs with "intra16x16" prediction). The complexity information of a task can be set in the context to the coding tools applied in this task. Table 4.3 provides an example. The retrieved MB coding type (Column 7) for each "intra" prediction task is extracted via the function name (Column 1). A call to a function with name "Intra16x16" indicates that the MB is intra-coded and uses intra prediction for a whole 16x16 pixel block.

4.2.4 Extraction of coding information via instrumentation

In addition to retrieving coding information via the function name, we can also use instrumentation to provide additional information about a VCA's runtime behaviour. In our system, the system designer can embed instrumentation instructions in the VCA's reference code and derive essential coding information as part of the function trace. Compared to extracting coding information via function names (Section 4.2.3), exploiting instrumentation has two advantages. First, the system designer can gain information about a VCA's algorithmic internals that would not be retrievable by observing the function calls/returns that occurred during the VCA's execution. For example, let us assume that there is only one function for intra decoding of a MB. A designer can only infer that the MB is intra-coded but could not gain information about, for example, the coding mode that is used for that specific MB. Second, we can retrieve complex information such as the data-dependencies between VCL coding elements from within the VCA. In particular, in order to estimate the execution behaviour of parallel application, knowledge about the data-dependencies is important.

4.2.5 Implementation

We implemented a library using the *Perl* scripting language [Wal00]. This language provides a powerful text parsing functionality and is available for most operating systems. The library which we will refer to as the data-driven profiling library (DDPL) generates a DDP in the following way: First, the user describes the VCA structure using the functions provided by the DDPL. The DDPL automatically generates an state machine and a corresponding parser that can map function traces via this state machine definition. Second, the designer provides the function trace information retrieved with a conventional profiler to the DDPL. Third, the DDPL remaps the profiling information into a DDP using the VCA's abstract state machine description.

Definition of a state machine

The DDPL provides simple functions for defining a set of states. Each state has an unique identifier (UID) and a set of transitions that are associated with this state. The user defines a transition by specifying the source and destination states of this transition and the function names that can trigger a transition between these states.

For each transition, one or multiple functions can be defined. These function remap the function trace profiling information when a transition occurs. Each time a transition is triggered by a function call/return, these functions are able to extract profiling information from the function traces and update the DDP correspondingly. The DDPL provides functions that support this extraction process in an intuitive and efficient way. For example, the time between entering and exiting a state can be retrieved. Furthermore, each transition function can create custom entries in the DDP for storing data-specific information. For example, an "intra" state's transition function can create an entry "coding" for a specific task and store the MB coding mode.

Mapping of function traces

After describing the VCA, the user has to provide the function trace profile to the DDPL. The function traces/instrumentation messages are either transferred directly from the profiled application's process to the DDPL's process via process pipelines or stored into a text file before being passed on to the DDPL process. Using the state machine description, the DDPL reformats all profiling information into a DDP representation and stores it into a database. This database contains a detailed information about the tasks that occur during the VCA's execution and provides a powerful starting point for analysing a VCA's complexity.

Merging of multiple DDPs

One drawback of using means of instrumentation is that the additional instructions used for generating profiling information can affect the runtime behaviour of the profiled application. The DDPL addresses this issue by providing tools for merging multiple DDPs. We can generate (i) a DDP based on function traces and (ii) an instrumentation-based DDP for extracting coding information separately and merge these profiles into a single DDP. The user can select which information shall be used from each individual DDP. The DDPL uses the tasks' UIDs, which are the same in both DDPs, for merging the selected sources. By using the complexity information from the DDP that has been generated without instrumentation, unbiased complexity profiling can be used in combination with detailed coding information derived by instrumentation.

4.3 Profiling environment and test sequences

In this section we describe the profiling environment that is used in this thesis. We generate a conventional profile using function traces and instrumentation. In the next section, we use the state machine definition shown in Figure 4.3 to generate a DDP from this information.

4.3.1 Reference architecture

All the profiling results presented in this work are based on an embedded video processing architecture. This reference architecture targets the efficient processing of audio- and video-based multimedia applications and represents a typical embedded platform used for video coding purposes. This should enable the reader to transfer results and conclusions of this thesis to other hardware platforms.

Hardware platform

Figure 4.4 visualises the SoC architecture that has been used throughout this work. The SVENm multimedia platform [SBG08] consists of two very long instruction word (VLIW) video processors named CHILIs, an ARM9 processor and a display content controller (DCC). All processors run at 300 MHz and can execute independent program code in parallel.

The CHILI is a RISC (i.e. Reduced Instruction Set Computer) processor that can process four instructions in parallel which can be any combination of 32-bit arithmetic instructions and

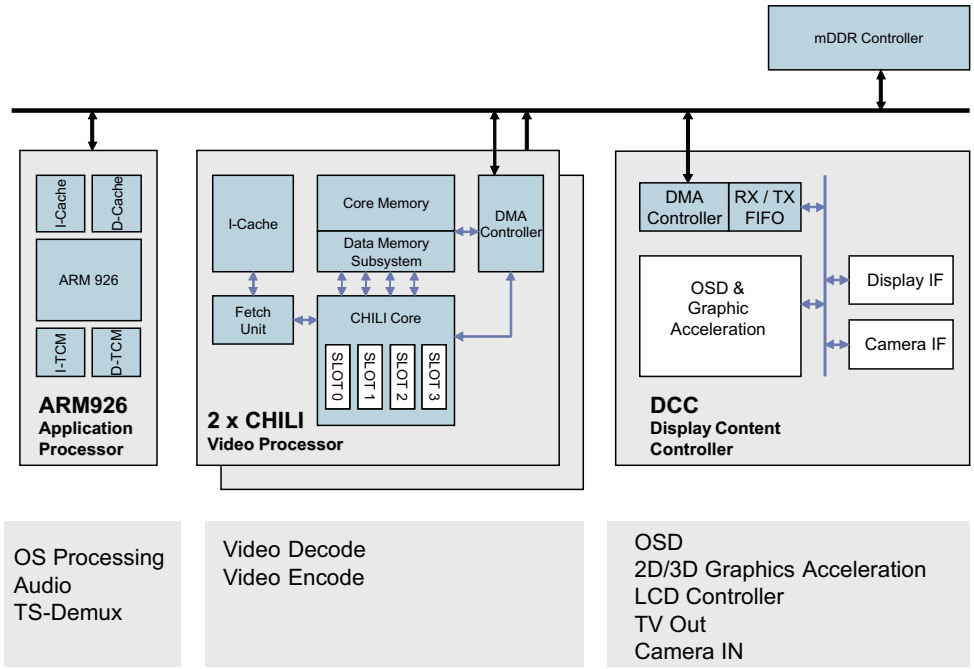


Figure 4.4: The structure of the SVENm architecture. The SVENm consists of two VLIW video processors named CHILIs, an ARM processor and a display content controller (DCC).

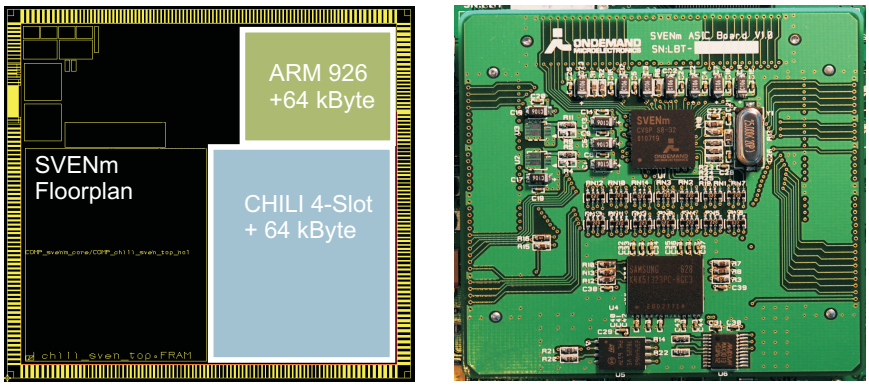
load-store operations. For parallel pixel operations, 16-bit SIMD (i.e. single instruction multiple data) instructions are provided. Each processor has a dedicated 64 kilobyte (kB) local memory for fast data access and a 64 kB instruction cache. Data is transferred by a direct memory access (DMA) controller or via direct memory access between the processor's local memory and 64 kB on-chip shared random-access memory (SRAM) as well as to the external mobile double data rate memory (mDDR). For efficient program execution, each CHILI uses a 64 kB instruction cache (ICACHE).

While the CHILI as a VLIW processor is designed for computationally intensive video processing tasks, the ARM9 processor architecture is more suitable for executing conditional code (e.g. the multiplexing of transport streams (TS)). For multi-core applications, the ARM9 can be used for controlling the communication and synchronisation between the individual processors.

A display content controller (DCC) handles the displaying of the video information on displays. It supports important displaying functions such as video scaling, color space conversion and buffered/unbuffered display output. A DMA controller supports the efficient data-fetching and transferring between the processors' and controller's local memories/buffers and the external memories.

H.264 decoder software

For this analysis, a commercial H.264/AVC decoder for embedded architectures has been adapted to this platform. This decoder supports all features of the H.264/AVC Main Profile such as B-



(a) (b)

Figure 4.5: SVENm architecture: (a) The floorplan of the SVENm architecture. Approximately half of SVENm is occupied by the two CHILI processors, the ARM processor and the processors' local memories. (b) A board with the SVENm multimedia chip attached at the board's centre.

frames and CABAC entropy coding. We started with a single-core C implementation of the H.264 decoder running on a PC platform. The decoder has been compiled to run on a single CHILI processor and been optimised in terms of memory usage and support of DMA transfers. Furthermore, the regular pixel-based processing functions of the decoder (e.g., interpolation, prediction) have been optimised at a low-level programming language level to make use of the SIMD processor commands. Intrinsic functions provided by the CHILI compiler have been used for integrating assembly code instructions within the decoder software's C code.

Decoder profiling on SVENm

For generating traces of the H.264 decoder on the SVENm reference hardware, a single-core ISS for the SVENm architecture is available. The simulator mimics the hardware behaviour of the processors and memories on the SVENm platform and enables cycle-accurate software runtime profiling on this architecture. We have extended the ISS to provide the time of each function call and each function return that has occurred during the execution of the H.264 decoder (Figure 4.1). Note that in this work we have used the CHILI profiler for obtaining this information. Nevertheless, this is no restriction of our approach, since profilers for other platforms are, in general, also able to provide this trace information.

Apart from function trace information, we have also instrumented our H.264 decoder to provide information on the coding process of the individual MBs (e.g. prediction type and modes). This information is extracted (i.e. profiled) and stored separately from the complexity profiling information and hence, does not alter the accuracy of the profiled complexity information.

4.3.2 Test sequences

To analyse our decoder, we have selected 16 HD video sequences from the Xiph.org test media website [Xip13]. The sequences are visualised in Figure 4.6. More details on the individual test sequences are provided in Appendix A. In compiling this test set, we aimed to reach a high diversity in the test sequences' contents and to cover the whole complexity range of typical H.264 sequences. In our test sequences, the recorded scenes vary in the amount of motion, texturedness and motion patterns. This results in strong variations in the sequences' bitrates and the applied coding tools, and enables a detailed analysis of the decoder's runtime behaviour for these scenarios.

All sequences are in progressive format with 720p resolution (i.e. 1280×720 pixels) and have a length of 49 frames. For encoding the test streams, we enabled the most commonly used coding tools supported in the main profile of the H.264 standard. With the exception of interlaced coding, all main profile coding tools such as I-, P- and B-slices as well as weighted prediction are supported. The test sequences are encoded using the JM12.2 encoder [Joi13] with parameters chosen as follows: H.264 main profile, 720p, GOP size 12 frames, IPB, VLC, deblocking active, all prediction modes allowed, SR +/–16 pixels, 5 reference frames. Figure 4.7a and 4.7b show the displaying and the coding order of the first 25 frames of an IPB-coded sequence when using a GOP size of 12 frames, respectively. We describe the coding of the test sequences in detail in the following sections.

Image quality metric

In this work, we use the Peak Signal-to-Noise Ratio (PSNR) for measuring the image quality. The PSNR is defined by the Mean Squared Error (MSE) between the original frame I_{orig} and the decompressed frame I_{decode}. The MSE is defined in the following way:

$$MSE = \frac{1}{xy} \sum_{i=1}^{x} \sum_{j=1}^{y} \| I_{orig}(i,j) - I_{decode}(i,j) \| \tag{4.1}$$

In this equation, x and y represent the width and the hight of the image, respectively. The absolute differences between pixels $I_{orig}(i,j)$ in the original frame and pixels $I_{decode}(i,j)$ in the decompressed frame are summed up. The indices i and j specify the horizontal and vertical positions within the frame, respectively. The MSE computes the average pixel difference occurring in a frame. Using the MSE, the PSNR is defined as:

$$PSNR = 10 \cdot log_{10} \left(\frac{MAX_I^2}{MSE} \right) = 20 \cdot log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \tag{4.2}$$

where MAX_I denotes the maximum possible signal value (i.e. in the case of 8-bit RGB pixel values, 255). The PSNR is measured in decibels (dB) and typically ranges between 20 and 40 dB.

Since the human eye is more sensitive to brightness/intensity than to color, only the PSNR between the luma channels (Y-PSNR) is typically used for quality comparison. Hence, in the extent of this thesis the term "PSNR" always refers to the Y-PSNR value.

Figure 4.6: The 16 test sequences used in work. We use these sequences for analysing the runtime behaviour of an H.264 decoder. Furthermore, we evaluate the high-level simulation methodology developed in this work using this test set.

A video encoder typically adjusts the coding process to address specific requirements or limitations targeted by the different applications. This enables, for example, specific bandwidth limitations, visual quality requirements or transmission latency limits to be met.

For evaluating the complexity estimation techniques proposed in the extent of this thesis, the test sequences must enable the analysis of the decoder's runtime behaviour over a wide range of complexity scenarios and for different coding tools. One prime parameter of the test sequences that affects decoding complexity in a significant way is the video stream's bitrate. The higher the bitrate of the coded video stream, the higher the number of video coding elements that must be processed by the decoder and hence the higher the decoder's runtime complexity. Hence, one prime focus when generating the test sequences for this thesis was to represent a wide range of test streams with different bitrates.

It should be noted that the amount of texturedness and motion within the scene strongly influences the bitrate, and thus multiple sequences coded with the same quantisation parameter (QP) settings do not necessarily have the same PSNR quality. For enabling a better comparison between test sequences and their runtime complexity at the decoder, we have chosen a QP setup for each sequence that achieves a constant PSNR quality of 40 dB. We have adjusted the values of the quantisation parameters QP_I, QP_P and QP_B that are responsible for quantisation of I-, P- and

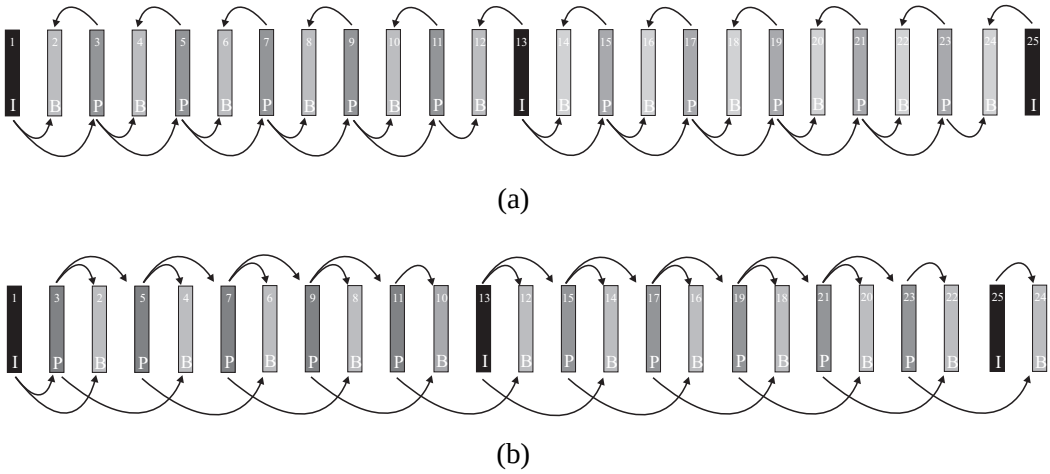


Figure 4.7: GOP-Coding of an IPB-coded sequence with 25 frames and a GOP size of 12 frames: (a) IPB-Coding: View order and (b) IPB-Coding: Coding order.

B-frames, respectively, so that each individual frame of the sequence exhibits a PSNR of approximately 40 dB. On one hand this enforces video coding at a constant and high quality of 40 dB. On the other hand, bitrates variations due to different content will still occur. In the following, we will describe this normalization of the PSNR quality and resulting bitrates in detail.

Normalised and average bitrates

In this thesis, we use the normalised bitrate r and the average bitrate R for data rate comparison. The normalised bitrate *r* of a video stream describes the average number of bits used for storing a pixel in a compressed video file. The normalised bitrate r is calculated in the following way:

$$r = \frac{b}{x * y * N}\ bits/pixel \tag{4.3}$$

In this equation, b refers to the file size, x and y to the frame width and height, respectively, and N to the number of frames stored in this file.

The average bitrate R refers to the average number of bits used for storing a second of video material. It is calculated by dividing the file size b by the time t (in seconds) of the video:

$$R = \frac{b}{t}\ bits/second = \frac{r * x * y * N}{t}\ bits/second \tag{4.4}$$

We can observe that the average bitrate R can be derived from the normalised bitrate r. In the following, the term “Bitrate” always refers to the average bitrate R.

Coding and bitrates of the test sequences

At a Y-PSNR of approximately 40 dB, the test sequences have average bitrates between 1.8 and 57.7 MBit/s (Figure 4.8). The bitrate is an indicator for the amount of texture and motion

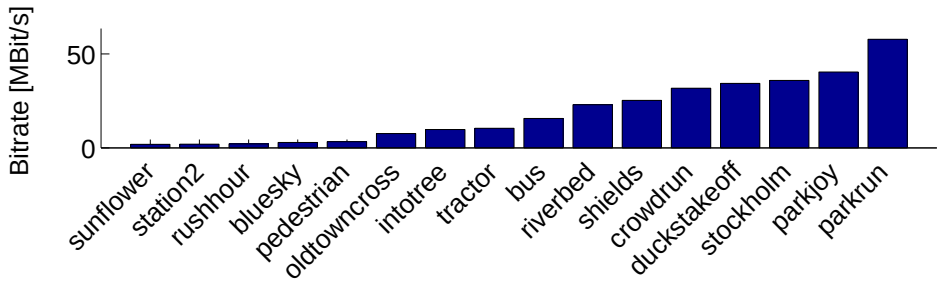


Figure 4.8: Bitrates for the 16 test sequences at a Y-PSNR of 40 db.

occurring in a sequence. The corresponding settings for the quantisation parameters for the first 25 frames of each test sequence as well as resulting bitrates for each individual frame are found in Table 4.4. The sequences are sorted according to their average bitrate R.

In the sequences with low bitrates such as “Sunflower”, “Station2”, “Bluesky” and “Pedestrian”, intra-coded frames are causing most of the total file size b. These sequences contain simple texture patterns such as the blue sky in the “Bluesky” sequence and inter-coded frames can be coded highly efficiently. For example for the “Bluesky” sequence, this results in low average bitrates of 2.3 MBit/s (Figure 4.8) and constant and low bitrates of around 500, 100 and 50 KBits for I-, P- and B-frames, respectively.

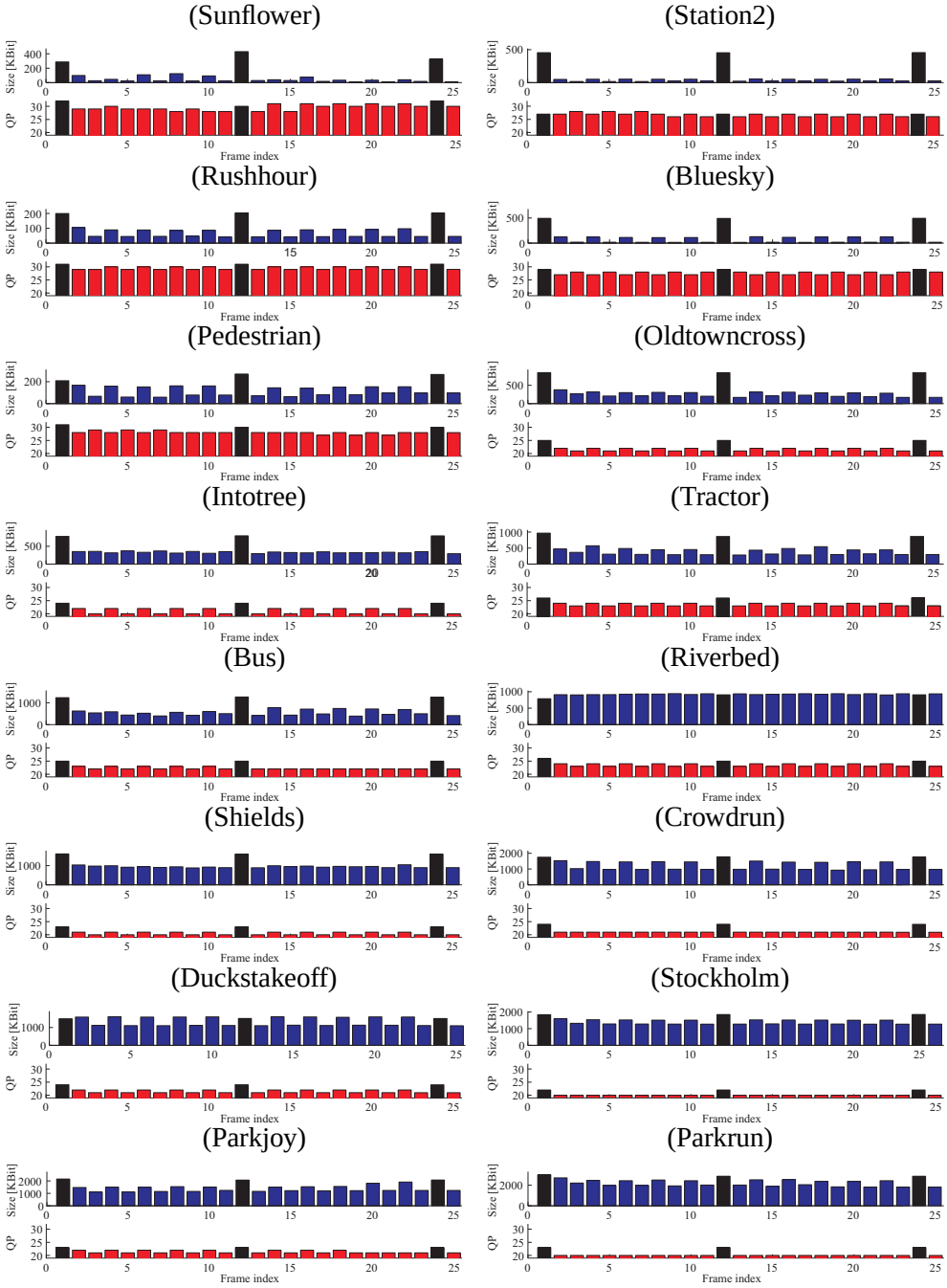
In the “Riverbed” sequence, high and similar data rates can be observed for P- and B-frames. Despite the fact that the sequence contains little texture, the motion prediction fails to efficiently predict the complex water flow. This results in a high number of intra-coded MBs in inter-coded frames as well as a high number of residuals for inter-coded MBs.

“Shields” and “Stockholm” are both moderately textured. The horizontal movement in the “Shields” sequence as well as the zooming operation of the camera in the “Stockholm” sequence result in a slightly higher motion activity than in the “Bus” sequence. The higher texturedness and motion activity for the “Shields” and “Stockholm” sequences lead to bitrates of 18.8 and 25.6 MBit/s, respectively.

“Parkjoy” and “Parkrun” are the two sequences with the highest average bitrates in our testset. In the “Parkjoy” sequence, the fast moving trees (motion > 32 pixels for some parts) in the foreground result in bad results for the motion prediction and frequent temporal occlusions of the strongly textured background. The encoder uses data-intensive intra-coded macroblocks for these blocks which also explains the high bitrate and frame sizes for this sequence. For the “Parkrun” sequence, the fine structures of the trees and aliasing effects result in a bad prediction and high residual information in this region. The background in the “Parkrun” sequence contains strong texture patterns and a strong horizontal motion with various temporal occlusions. This leads to the highest bitrate in our test set (50.8 MBit/s).

Despite similar PSNR values for all the sequences, strong variations in the bitrates can be observed. In the next section, we exploit information available in DDPs for analysing the decoding complexity for these sequences at a MB-level.

Table 4.4: Detailed size and quantisation values for the first 25 frames of each test sequence. For the first two GOPs of each sequence, the size of the encoded frames and the used quantisation value (QP) are shown. The frames are provided in coding order and the horizontal axis represents the frame index. All I-frames are highlighted as black bars.



4.4 Experimental results for runtime analysis and visualization

This section provides functional runtime profilings results for the H.264 decoder running on the reference architecture described in Section 4.3.1. One major aim is to gain information about the complexity and the dynamic runtime variations for the major blocks of the H.264 decoder for the 16 test sequences. Using the dynamic profiler for our reference architecture, we can extract detailed information about the runtime complexity of the H.264 decoder's individual MBs.

4.4.1 Complexity of processing VCL coding elements

The information obtained from DDPs enables us to interrelate runtime complexity with the decoding process of individual VCL coding elements. This is essential when designing data-parallel partitioning approaches, since differences in the decoding complexity of individual VCL coding elements affect the efficiency when distributing the workload onto multiple cores and, consequently, the overall runtime. Furthermore, it determines the need for buffers for compensating workload differences.

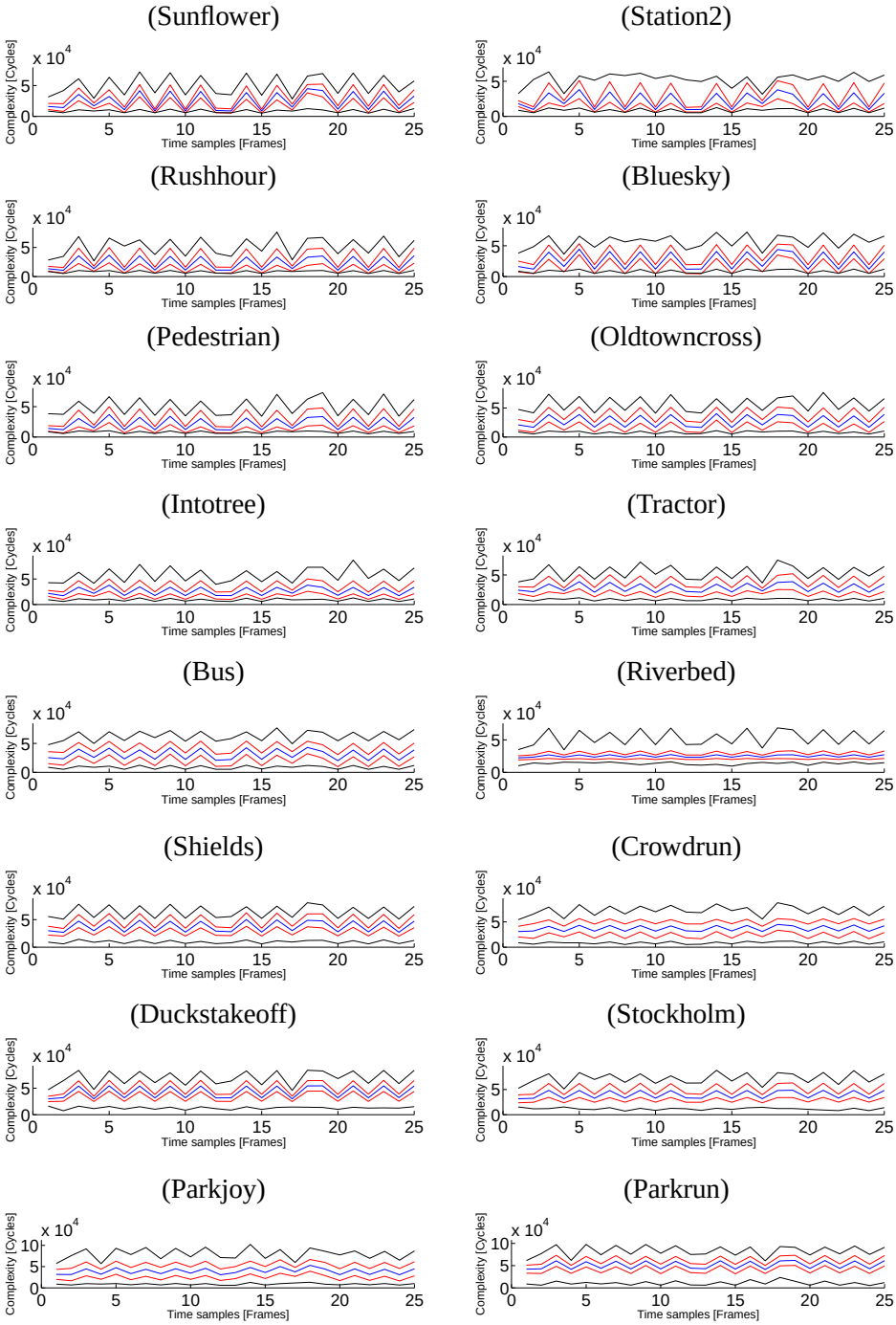
Table 4.5 provides an example of a MB-based DDP complexity profiling. For all frames of the individual test sequences, the complexity for processing MBs has been extracted. The figure provides the average, minimum and maximum processing complexity of the MBs decoded in the respective frames. Furthermore, the standard deviation of the MBs' decoding complexity is visualised by the red lines.

We can see that the average complexity for processing a MB in all sequences is similar between I- and P-frames, but higher for B-frames. We can observe that the complexity of sequences such as the "Bus", "Parkjoy", "Parkrun" and "Crowdrun" shows constant standard deviations throughout all frames and for different frame types. The "Riverbed" sequence posesses a very low variation in complexity and shows nearly the same runtime complexity for I-, P- and B-frames. In contrast to this, low-bitrate sequences such as "Sunflower" and "Station2" show stronger variations in the complexity between different frame types. Especially within B-frames a strong variation between MB decoding complexity seems to exist. In a data-parallel VCA implementation that works on a slice or frame level, variations in the decoding complexity will probably affect the parallelisation efficiency strongest when decoding low-bitrate sequences.

Looking at the minimum and maximum complexity (black lines), we can see that throughout all sequences MBs with the lowest/highest complexity occur during the decoding of B-frames. These MBs with exceptionally high runtime will - similar to the already observed variations - affect the performance when decoding B-frames in a data-parallel way and result in a less efficient load balancing or/and the need for larger memory buffers to compensate complexity variations.

Figure 4.9 provides more detailed information on the MBs' decoding complexity for the individual MBs. The complexity distributions shown in this figure outline that the decoding complexity of individual macroblocks increases with higher bitrates for all sequences. The important point in Figure 4.9, however, lies in the dynamic behaviour of macroblocks. It can be clearly seen that cycle counts are very different among individual macroblock coding types and video sequences. As is also shown in the figure, this observation can still be made when considering the classes of I-, B- and P-macroblocks alone.

Table 4.5: The decoding complexity of each macroblock varies significantly during the decoding. The blue line in each sequence plot shows the average of the macroblocks' decoding complexity in clock cycles for each frame (i.e. 3600 macroblocks). Additionally, the standard deviation and the minimum/maximum macroblock decoding complexity for each frame are indicated by the red and black lines, respectively. Frames 1, 12 and 24 are I-frames, Frames 3, 5, 7, 9, 11, 14, 16, 18, 20, 22 and 24 are B-frames and the remaining frames are P-frames.



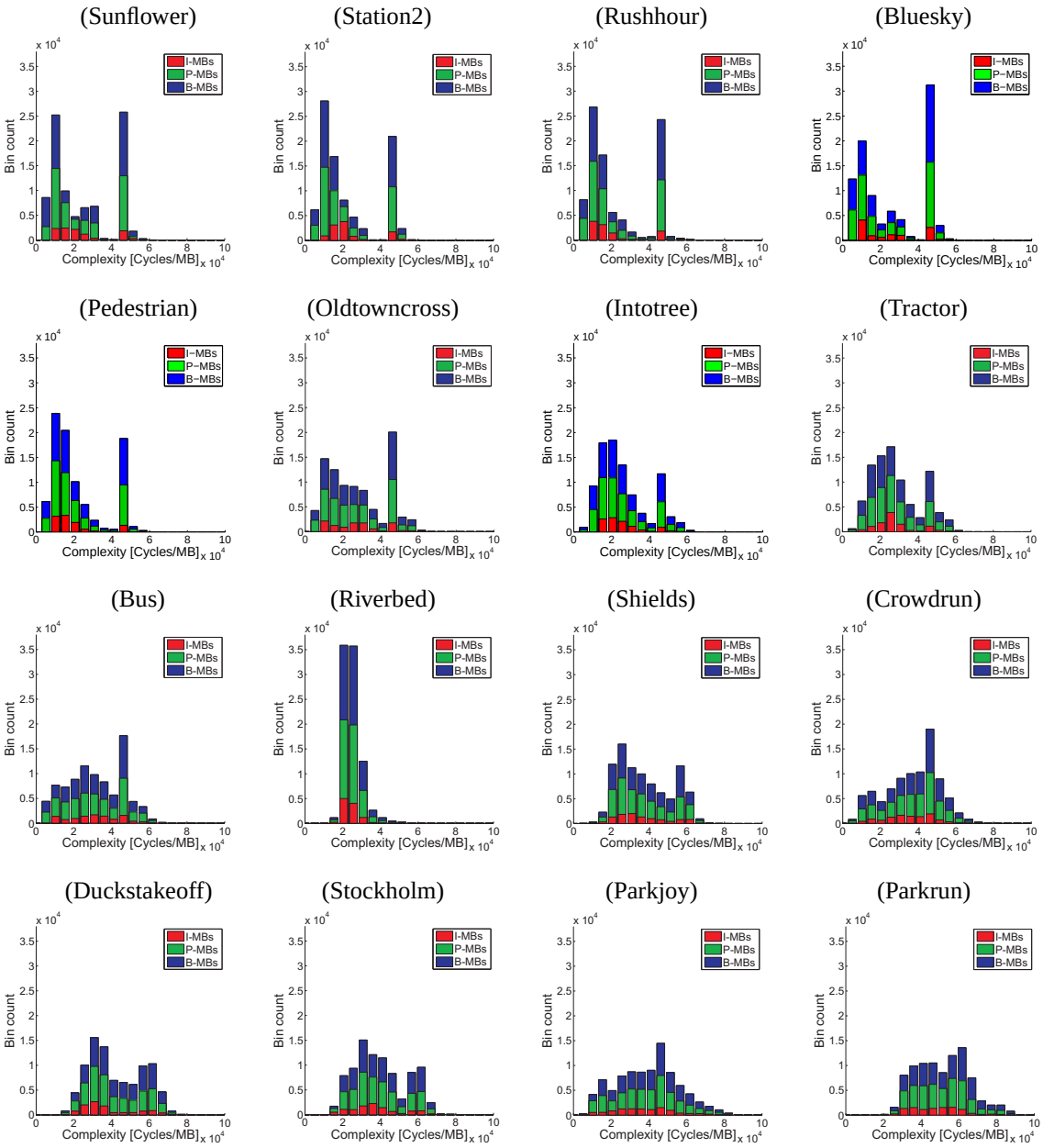


Figure 4.9: Dynamic variations in the execution times of the individual macroblocks in the H.264 decoding process. Histogram bins plot the number of macroblocks having similar runtimes. The colours indicate the contributions of macroblocks from I-, P- and B-slices to the overall bin counts. Histograms are shown for 25 frames of the sequences in Table 4.6. These sequences are IPB coded with the Group of Pictures (GOP) size being 11. It is observed that the runtimes of macroblocks vary considerably within a sequence. This observation is also made when considering I-, P- and B-macroblocks separately.

For low-bitrate sequences such as “Sunflower’, “Rushhour” and “Bluesky’, we can see in the complexity distribution of each sequence a clustered group of low-complexity MBs and a peak around approximately 40000 cycles. This peak corresponds to the variations observed in Section 4.4.1 and represents MBs coded by 16x16 prediction modes with large amounts of residual information. The parsing of this residual information accounts for most of these MBs’ decoding complexity and results in similar computational complexity amongst all coding types. For higher bitrates, the peaks within the distributions disappear and other prediction modes (e.g. based on 4x4 blocks) are more intensively used.

4.4.2 Complexity of processing functional blocks

Figure 4.10 shows the average and the standard deviation of the runtime complexity the individual decoding functions require for processing a MB. We have grouped the decoding functions into the FBs of Figure 3.2. The entropy decoding is thereby merged with the parsing block, since these functions are tightly connected during runtime.

For most sequences, only a small part of the runtime is spent on the IDCT and the prediction. The main reason for this is that the regular pixel-based operations such as inverse transformation, spatial and temporal pixel prediction are well supported by the CHILI processor architecture. The SIMD instructions in combination with the VLIW architecture allow the processing of 8 image pixels in one clock cycle. In combination with DMA transfers and fast pixel data transfers, the pixel-based operations can be tackled efficiently.

However, conditional code execution does not benefit significantly from the VLIW/SIMD architecture. Highly conditional parts of the H.264 such as the entropy coding and deblocking therefore do not perform well. This can be seen in the profiling results. For all sequences, most of the decoding time is used for parsing (i.e. bit parsing and entropy coding) and deblocking.

We can see in Figure 4.10 that the average runtime for parsing is correlated with the sequences’ bitrates visualised in Figure 4.8. For sequences with high bitrates such as Stockholm and Parkrun, the average runtime for the entropy decoding is significantly higher than for low-bitrate sequences. For example, between the “Sunflower” sequence with an average bitrate of 1.8 MBit/s and the “Parkrun” sequence with an average bitrate of 57.7 MBit/s an increase in runtime by a factor of 6 (i.e. 5000 cycles compared to 30000 cycles per MB) can be observed.

Variations in the execution times of the decoding functions will result in a highly dynamic system when developing parallel decoders and will impact the parallel execution of the H.264 decoder. The variations in the runtime become visible when analysing the standard deviation in the decoder’s runtime profilings. We can see that for all major functional blocks of the decoder, runtime variations occur. Especially, the entropy coding and the deblocking are highly sensitive to the bitrate of input data. The bitrate has significant impact on the complexity of these decoding blocks. For sequences with high bitrates, we can observe significant runtime variation for the parser functions. For example, the average runtime per MB for the “Parkrun” sequence with 57.7 MBit/s shows variations of +/-20000 cycles/MB. Compared to this, significantly smaller runtime variations of approximately 1600 cycles/MB are observed for the “Sunflow” sequence with 1.8 MBit/s average bitrate.

For pixel-based decoder blocks such as IDCT and prediction, small runtime variations can be observed. However, we can see that in relation to the average runtime, these runtime variations

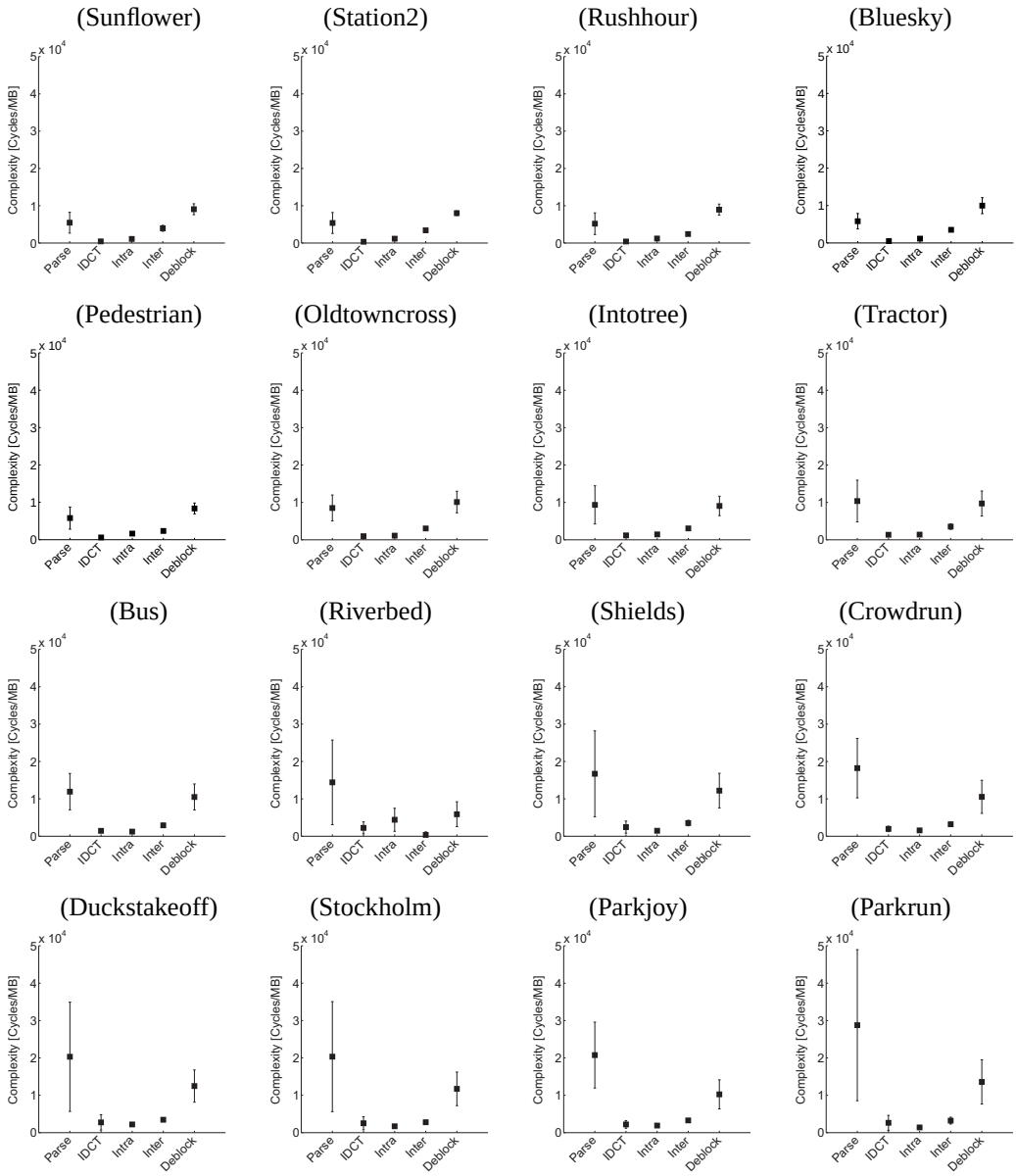


Figure 4.10: Dynamic variations in the execution times of the individual decoding functions in the H.264 decoding process. The decoder is divided into five functional blocks, namely parsing, inverse DCT, intra prediction, inter prediction and deblocking. The bars plot mean complexity and standard deviation for each of the decoding blocks. It is observed that runtime complexities of individual decoding functions considerably vary within a sequence due to different runtime behaviour of individual macroblocks.

can become relatively strong for high-bitrate sequences such as the "Riverbed" and "Duckstake-off". For example, the runtime of the "Riverbed" sequence's Intra block shows a standard deviation of approximately 60 percent of the average runtime (i.e. $1000/4000$ cycles/MB compared to 2500 cycles/MB average runtime).

The knowledge about the average runtime and runtime variations of an H.264 decoder provides us with means for roughly estimating the decoder's suitability for running on multiple cores. However, estimating the runtime of a partitioned decoder based on the dynamic profilings is not straightforward. An extended profiling method that provides more suitable means for multi-core runtime estimation is introduced in the next section.

4.4.3 Analysing complexity within individual subregions of a frame

We have seen in Section 3.5.3 that data-parallel VCA splitting partitions the processing tasks by assigning subregions of a frame to different PUs. By analysing the coding information available within the DDPs, the coding complexity of individual subregions of a frame can be derived. An example of how coding information can be interrelated with the runtime complexity is provided in Figure 4.11. This figure visualises the MBs' decoding complexity for 3 frames of the "Parkjoy" sequence. It should be noted that in Figure 4.11, coding information about the position of each MB within each frame has been extracted. For each MB, the bitrate, the total runtime, the runtime for parser, IDCT, prediction and deblocking FBs are visualised. The complexity has been normalised and white represents regions with high complexity and black with low complexity. We can see that most of the total runtime in I-, P- and B-frames is used for decoding the bright and textured regions in the background (e.g. trees). For frame regions with a high bitrate, a high runtime in the parser FB and a high total runtime can be observed. This indicates that the parsing FB's complexity highly correlates with the image structure.

The IDCT FB shows an interesting behaviour amongst I-, P- and B-frames. While for I-frames nearly the same time is required for the IDCT FB of all MBs of the frame, stronger differences between individual regions can be observed for P- and B-frames. In the P- and B-frames, strong differences between low textured regions and highly-textured regions can be observed.

In I-frames, the prediction of textured regions requires significantly more runtime than for untextured regions. This results from the strong complexity differences between the individual intra prediction modes. In B-frames, the runtimes are similar between the frame's MBs. However, a few MBs with high complexity occur in the textured regions while a slightly lower runtime can be observed in the untextured regions. In P-frames, a moderate runtime is spent for the textured regions in the background while little runtime is required for the untextured regions in the foreground. Single MBs within the frame with very high runtime can be observed.

Apart from the top and left border, the deblocking filter's complexity shows a very uniform complexity distribution for the I-frame. In the P-frame most of the runtime is spent on filtering the textured areas of the frame. This is in contrast to the B-frames where a high runtime is dedicated to processing untextured regions.

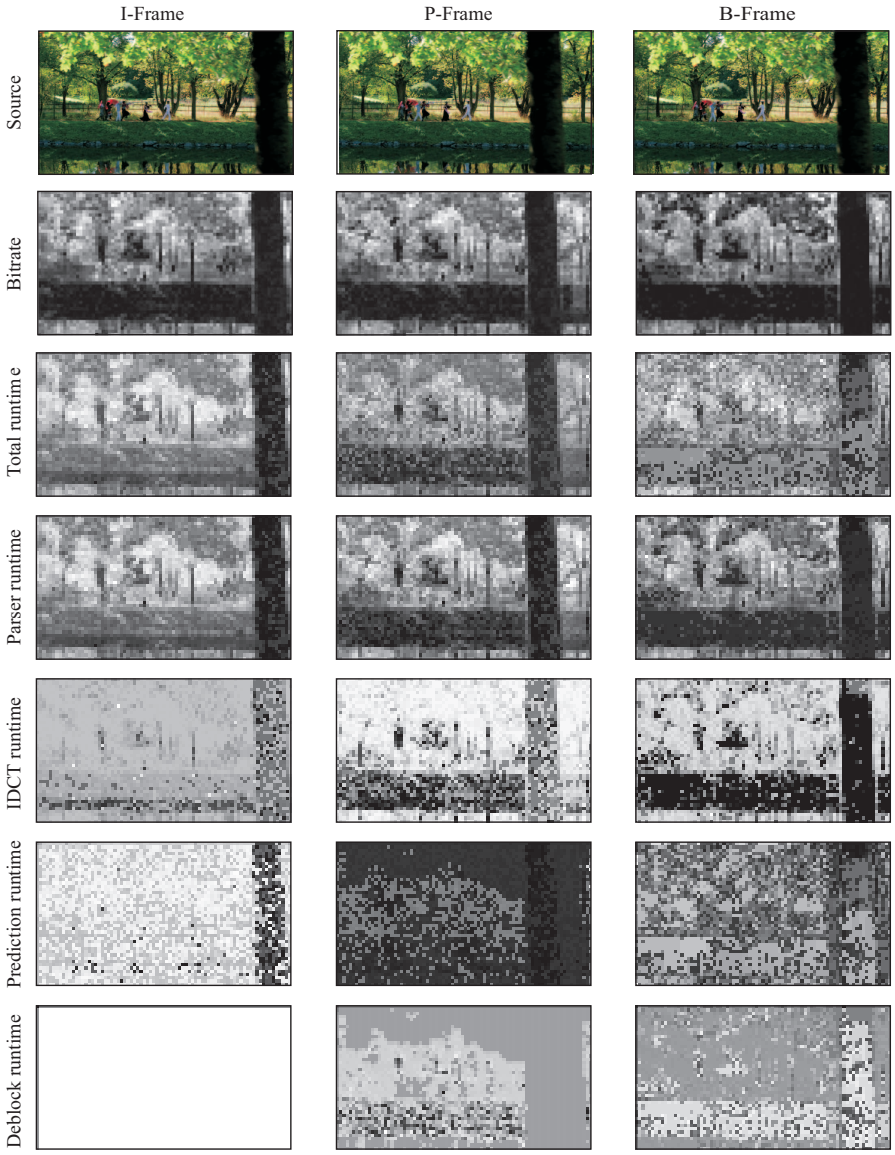


Figure 4.11: The runtime complexity for individual MBs of I/P/B-frames: For each frame, the bitrate, total runtime complexity and the complexity of parser, IDCT, prediction and deblocking FBs for each MB are visualised. Bright pixels indicate regions with a high bitrate/computational complexity. Dark values macroblocks with a low bitrate/decoding time.

4.5 Summary

In this chapter, we have introduced the Data-Driven Profiling method which maps traditional runtime profiling information onto the VCL coding elements and functional blocks of a VCA. This provides means for estimating a parallel VCA's complexity behaviour and for parallel VCA designing and enables the system designer to investigate critical aspects such as variations in the processing time of each coding element and individual functional blocks of the VCA. Furthermore, it enables the extraction of VCA-specific coding information such as the coding tools used when processing specific VCL coding elements.

After describing the test setup used throughout this thesis, we have demonstrated three ways of exploiting DDPs for analysing complexity and deriving essential information for parallel system design. First, we have demonstrated how complexity information about the processed VCL coding elements can already highlight potential problems in load-balancing for frame- and slice-based data-parallel approaches at an early design stage. Second, we have shown how complexity variations in the FBs of a VCA can be analysed. This provides a starting point for implementing functional partitioning techniques. For demonstrating the above contributions, we have exploited runtime profilings of an H.264 decoder to analyse the dynamic runtime variations in the decoder's functional blocks. We have shown that the runtime as well as the runtime variations for the individual H.264 decoder blocks increase with the bitrate. Decoding blocks with a large amount of conditional code such as the entropy decoding and the deblocking are more sensitive to bitrate changes than pixel-based blocks. Third, we have extracted coding information which determines the progam flow of a VCA and exploited this information to determine the processing time for individual image regions. Similar to the complexity analysis for VCL coding elements, these insights support the design of efficient data-parallel partitioning approaches.

Overall, the capability to analyse and visualise the runtime complexity of VCL coding elements and functional blocks provides an essential tool for parallel VCA design and when targeting an equal workload distribution in data-parallel and functional-partitioned designs. We will exploit DDPs in Chapter 5 for the development of a novel multi-processor simulation approach.

CHAPTER 5

Virtual prototyping of parallel video coding systems

In this chapter, we introduce a high-level simulation methodology for complex VCAs. This methodology represents a main contribution of this thesis and enables estimation of the VCA's parallel runtime behaviour on virtual hardware architectures. Section 5.1 describes the aspects and design goals that such a simulation method has to focus on. Section 5.2 introduces our new concept. The assumptions underlying this concept and its limitations are described. Section 5.3 describes the concept's implementation.

5.1 General aspects and design goals

A methodology for estimating the runtime of a partitioned VCA on a virtual platform has to address various aspects. According to Holzmann et al. [Hol91], the design process of a formal model must address three aspects efficiently: *descriptive clarity*, *modelling power* and *analytical power*. These criteria are considered as the prime indicators for the "quality" of a formal model.

Descriptive clarity in the context of virtual prototyping requires that clear and intuitive modelling of the virtual system and the VCA's execution is possible. The conceptual simplicity of building a VCA model has a major impact on the modelling process. It influences the ease and flexibility of the system designer to specify a virtual VCA. Especially for rapid design explorations, descriptive clarity is of prime importance. The effort involved in modelling a VCA partitioning will typically influence the number of partitioning approaches the system designer can consider in his analysis.

The *modelling power* of such a method must enable the system designer to describe a virtual system without restricting the designer's creative freedom. Describing the system's underlying processes in an accurate and technically feasible way is required. Consequently, the modelling has to cover all aspects that influence the total execution behaviour and time of a virtual VCA.

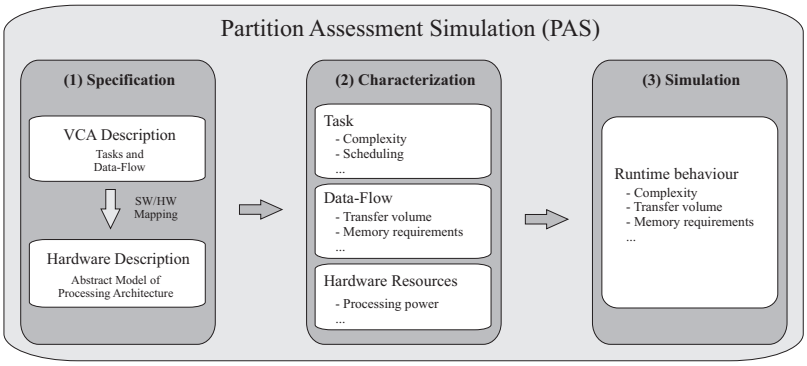


Figure 5.1: The Partition Assessment Simulation (PAS) can simulate virtual multi-core platforms running VCAs. First, an abstract description of a VCA and the system's hardware is provided. Second, the system's runtime behaviour is specified by providing information about complexity, execution order and communication between tasks. Third, the PAS uses this information to estimate the runtime behaviour of the VCA running on this virtual coding platform. Approximations of the VCA's performance (e.g., complexity, memory accesses, etc.) are derived without the need to fully implement the hardware or the VCA's software.

This includes, for example, defining and describing the VCA's tasks and task-interdependencies and setting the VCA's model in context of a virtual HW architecture (e.g. SW/HW mapping).

The investigation of dynamic runtime aspects in a parallel system cannot be done based on static analysis. Hence, the *analytical power* of a suitable method is strongly related to the method's capability to simulate the system's behaviour under runtime. It must be able to (i) simulate such a system design and (ii) automatically derive and transform the resulting simulator information into a human-interpretable representation. In the following, we present a concept that aims to address the conceptual objectives stated above.

5.2 Concept

In this chapter, we propose a high-level modelling approach that enables system designers to specify a VCA's *system* in an abstract way. It supports the specification of virtual multi-core coding platforms and can derive the runtime behaviour of a VCA running on this platform in an automatic way. The design of our methodology provides means for addressing important aspects of design space exploration in the context of parallel video coding systems:

1. Means to describe the high-level functionality of a video application (i.e. modelling of computational tasks) must be provided. In our work, the domain-specific aspects of video coding shall be addressed.

2. A simulation mechanism for mimicking the system's *parallel* runtime behaviour based on this formal description is necessary. Hardware-related aspects such as scheduling, data exchange and notation of time must be supported.

3. Means shall be provided that allow the designer to integrate knowledge about the runtime behaviour of VCA tasks on similar hardware (e.g. gained via profiling) within this model.

4. Adaptation of traditional complexity estimation and profiling techniques for extracting a VCA's runtime information that is typically available during the system design.

5. The simulation results regarding the performance of a design must be transformed into a human-readable representation.

The idea behind our methodology, which we refer to as the *Partition Assessment Simulation (PAS)*, is that a VCA running on a multi-core architecture executes the same tasks as its single-core implementation. However, in multi-core systems the parallel processing resources enable the concurrent execution of these tasks. This results in a different execution order of the tasks. The PAS simulates parallel processing, changing execution order, and estimates the implications on the VCA's runtime. Figure 5.1 illustrates the three major parts of the PAS: Specification, characterisation and simulation.

System specification: The system designer defines virtual hardware resources such as processing units (PUs) and memories, the computational tasks that are executed during a VCA's execution and the dependencies between these tasks.

Characterisation: Information about the VCA's tasks complexity and the exchange of data between tasks is provided. This expert knowledge can often be derived from profilings of the VCA running on single-core platforms. For System-on-Chip (SoC) architectures which are typically built from existing components, this enables us to put the SoC's simulation in context to hardware profilings from its already available single-core components.

Simulation: The PAS combines this information to set the tasks in the context of this new virtual system. It determines the impact on the individual tasks' runtime in a system with different hardware resources and software partitioning.

Using abstract models for describing hardware and software results in high flexibility when doing design space explorations. This includes the evaluation of unknown hardware configurations as well as software partitionings. For example, let us assume that we want to test a new multi-core hardware system. In this case, the system designer only needs to reformulate the description of the hardware and update the mapping of the tasks to the given processing cores.

In the following, Section 5.2.1 introduces a method based on *dependency graphs* for describing the tasks and inter-task dependencies of a VCA efficiently. We extend this method in such a way that the mapping of tasks onto specific HW units and hardware resources (e.g. processors and memories) can be described. Sections 5.2.2 and 5.2.3 explain how hardware profiling information is used to characterize this virtual system and how simulation of such a system can be done, respectively.

5.2.1 System specification

Specifying a VCA's system within the PAS can be divided into four stages (Figure 5.2). First, a formal representation that describes a VCA's execution behaviour in an abstract (i.e. HW independent) way and down to a level of parallelisation (LoP) where parallel coding approaches shall be implemented. This step involves the definition of the tasks that are executed during

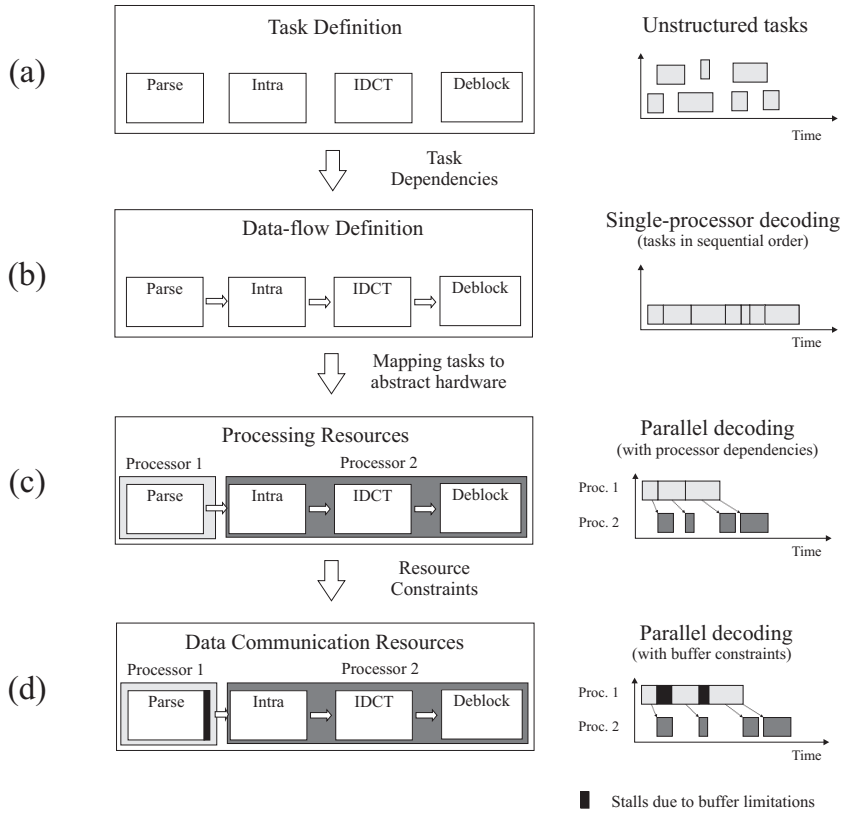


Figure 5.2: System specification in the PAS: (a) The designer defines the coding tasks that are executed during the coding of a video stream in an abstract way. (b) The dataflow and the resulting dependencies between these tasks are specified. (c) The tasks are mapped onto an abstract hardware platform. (d) The communication resources (i.e. FIFOs) for the communciation between tasks on different processors are included in our system's description.

the coding of a video stream and the dependencies between these tasks. This is visualised in Figures 5.2(a) and 5.2(b), respectively.

For data-driven applications such as VCAs, tasks can typically be derived from the FBs of the VCA. Each task reflects the execution of a FB for a particular VCL coding element. For example, a task could be the intra prediction of a macroblock. If the system designer is interested in investigating partitioning approaches for a decoder working at a slice level, the FBs and consequently the tasks are defined at a LoP that represents the decoding steps for individual slices. In our methodology the system designer models the FBs till a LoP that can describe the parallel processing of a decoder's partitioning approaches. This modelling at LoP has the advantage that partitioning approaches can be described without providing knowledge about the internal functionality of the underlying tasks themselves. This avoids the time-intensive task of modelling a detailed algorithm functionality that takes place inside the task. At the same time it does not place any limitation on the parallelisation itself since the FB granularity can be

increased arbitrarily.

After specifying tasks and dependencies, a virtual system definition (VSD) is defined (Figure 5.2(c)). The tasks are assigned (*mapped*) to virtual processing resources in this VSD. This allocates the tasks to physical processing resources and determines where the processing of the individual tasks takes place. In Figure 5.2(d), resources for simulating the communciation between the VCA's tasks on different processors are specified within our VSD. This allows the PAS to consider inter-processor communication during the system's simulation.

High-level algorithm description

As mentioned in the previous section, the first step in our modelling approach is to specify all tasks that occur at the LoP of a VCA. Specifying the tasks down to the LoP has a significant impact on the modelling complexity and hence on the methodology's suitability for fast prototyping. On the modelling side, no detailed information about the internal functionality of the individual tasks has to be provided. This significantly reduces the amount of information that is required when modelling a VCA's system. On the simulation side, the PAS can consider each task as *atomic* since no parallelisation inside a task takes place. This allows us to introduce the following two simplifications that affect the way our methodology describes and simulates parallel VCA systems:

First, we can assume that a task's execution cannot be interrupted nor distributed to multiple processors (i.e. *atomic execution*). Each task is considered as a sequence of instructions that is performed for a VCL coding element and once started by a processing unit (PU) is executed by this PU without an interruption till the end. It should be noted, that in a physical system hardware interrupts (e.g. for task scheduling or error handling) can occur during a task's execution. However, in a complex real-time VCA design, task scheduling can only be exploited in a very limited scope and the impact of task scheduling on the runtime performance can typically be neglected.

Second, we make the simplification that a PU requires approximately the same number of computational instructions and the same duration for executing a specific task, no matter whether the PU is the only PU in the system or part of a multi-processor system.

Based on these assumptions, the execution of task T can be specified by the task's start time $time_{start}(T)$ and end time $time_{end}(T)$ with

$$time_{end}(T) \geq time_{start}(T) \tag{5.1}$$

The task's duration $duration(T)$ is the time that passes (i.e. the difference) between start and end times:

$$duration(T) = time_{end}(T) - time_{start}(T), duration(T) \geq 0 \tag{5.2}$$

Let us assume that a VCA's execution consists of the individual executions of the VCA's tasks $T_1..T_N$. The total duration of this VCA running on a single PU is the sum of these tasks' durations:

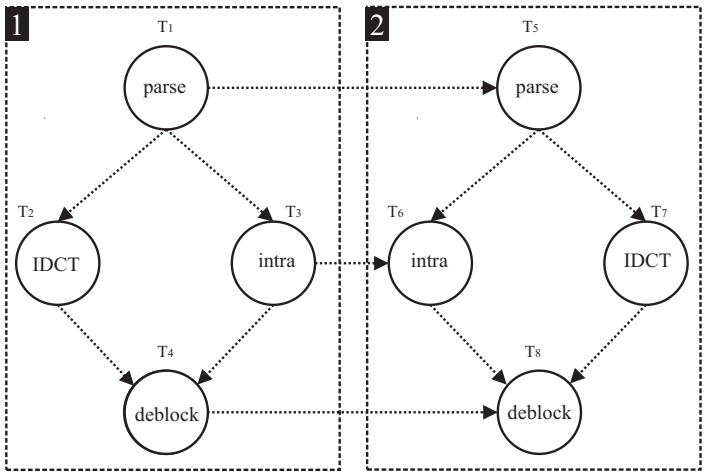


Figure 5.3: The figure visualises a dependency graph between two macroblocks' decoding tasks: In this example, a simple video stream consisting of 2 MBs is decoded. Each executed task is represented as a vertex. The directed edges represent the dependencies between the tasks.

$$duration(VCA) = \sum_{i=1..N} duration(T_i) \tag{5.3}$$

The exchange of data between tasks (i.e. the *dataflow*) results in data-dependencies and determines the tasks' execution order and start/end times. For simulating a VCA's execution, these data-dependencies must be considered.

For describing a VCA's dataflow in the PAS, we define a set of dependencies $D = \{D_1, D_2, ..., D_M\}$. Each dependency $D_j \in D$ describes a data-dependency between two tasks and is of the following form:

$$D_j : T_a \rightarrow T_b \tag{5.4}$$

In Equation 5.4, Dependency D_j determines that task T_b depends on task T_a. This means that the execution of T_b cannot be started until T_a has been finished:

$$D_j : T_a \rightarrow T_b \Rightarrow time_{start}(T_b) \geq time_{end}(T_a) \tag{5.5}$$

For describing tasks and data-dependencies of a VCA, *dependency graphs* provide a powerful and flexible concept. Figure 5.3 shows an example of a dependency graph of a simple decoder VCA. Each vertex in this dependency graph represents a task of our VCA. A directed edge between two tasks represents a dependency. It indicates that the execution of the task this edge is directed to cannot start before the other task has been finished.

Each task T_a in such a dependency graph represents a part of a VCA's overall runtime and is executed at a specific time intervall [$time_{start}(T_a)$;$time_{end}(T_a)$]. A cycle in our graph would mean that task T_a depends on itself and can only be executed after it has finished. This is not

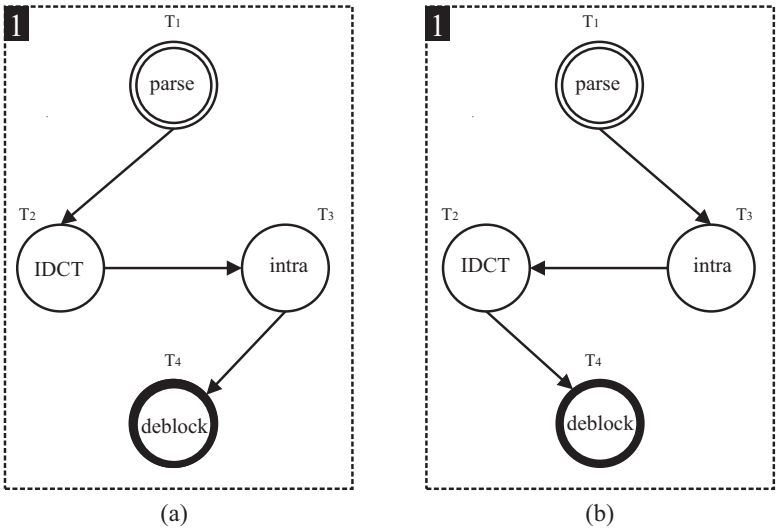


Figure 5.4: Sequential task order for a macroblock's decoding tasks: Figures (a) and (b) visualise two possible orders of executions of the 4 decoding tasks of Macroblock 1. The vertices representing the start and end tasks in this order of execution are marked by a double and a black circle, respectively.

possible for time-sequential programs. Hence, a VCA's dependency graph cannot contain cycles and can always be treated as a *Directed Acylic Graph* (DAG). Efficient algorithms for solving DAGs are available. In Section 5.2.3 we will explain, how the PAS exploits this for simulating a virtual system based on DAGs.

It is important to note, that while a VCA's dependency graph defines exactly which tasks must be executed before all data dependencies of a task T_i are solved, no unique *Order of Execution* (OoE) is derivable from such a DAG. For example, in Figure 5.3, after task T_1 has been executed two tasks T_2 and T_3 could be executed which shows the ambiguity in this description.

For describing a single-core VCA's execution in a deterministic way, we introduce the term *sequential task orders* (STOs). Each STO is a sequential list of tasks that determines an unique and sequential order between these tasks and defines a deterministic execution path that is taken during the VCA's execution. For our example in Figure 5.3 with a single MB and four tasks $T_1..T_4$, two STOs are possible:

$$STO_1 : T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \tag{5.6}$$

$$STO_2 : T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4 \tag{5.7}$$

Figures 5.4a and 5.4b visualise the two STOs in Equations 5.6 and 5.7, respectively. The solid arrows define the OoE which clearly states which task is executed after a dependency is resolved. An STO can be seen as a DAG where each task only provides data for another single

task (i.e. for each task T_i, only a single dependency $D_j : T_i \rightarrow T_b$ exists). The STO typically depends on the VCA's implementation for a specific platform and is defined consciously or not by the system designer during the software development stage. This will become more obvious in the next section when abstract hardware resources are added to our VCA model.

Abstract hardware resources

Next to a VCA's dataflow and the execution order between tasks, we must be able to describe the hardware components of our platform in an abstract way. In case of a parallel platfrom, multiple processing units (PUs) are available. We define a *processor set* $P_{VA} = \{P_1, P_2, ..., P_M\}$ that contains all PUs $P_{1..M}$ of a *virtual architecture* (VA). For each of the VCA's task, an *assignment* $A_i : T_i \rightarrow PU_j$ is defined which assigns task T_i to PU P_j.

Connecting the VCA's tasks with hardware resources introduces physical limitations into our VCA description. The assignment $A_i : T_i \rightarrow P_j$ determines that for computing task T_i, computational resources of the assigned PU P_j are used and that the execution of a task T_i can only take place while P_j is not executing another task (i.e. no parallel task execution on a single PU). In the following, we write $T_{i,j}$ to refer to task T_i that is executed on processor P_j.

Task execution order

While for single-core VCAs the choice of the STO does not alter the VCA's overall runtime, in a parallel environment with multiple STOs (i.e. one for each processor), dependencies between tasks of different STOs can have a significant impact on the overall runtime. For formally describing the execution order of a VCA in multi-processor environments, we define Models of Executions (MoEs). For a VCA running on M PUs, the MoE is a set of STOs $S_1, .., S_M$ that defines an STO for each PU. The MoE determines the execution within a VCA on a parallel system in a unique way and independently of the underlying hardware.

Figures 5.5a and 5.5b visualise the mapping of the VCA graph from Figure 5.3 onto 2 PUs (i.e. 2 MoEs). In Figure 5.5a, the decoding tasks of MB_1 have been assigned to PU P_1 and the tasks of MB_2 to P_2. This represents the case of a data-parallel decoding approach. Figure 5.5b shows a functional decoder partitioning where the *parser* decoding tasks have been assigned to PU P_1. The solid edges indicate the STOs for PUs P_1 and P_2. The dotted edges indicate data dependencies between tasks of different PUs. Each STO defines a start and end task for each PU. A PU stops the execution of his part of the VCA when all tasks that have been assigned to it have been executed. The first task of each STO can either depend on no other task of the VCA (i.e. the VCA's initial task) or tasks from other STOs.

Data communication between tasks

No physical limitations on the communication between tasks and PUs have been considered so far. For describing read and write transfers between tasks, we use *FIFO communication buffers*. Each VCA contains a set of FIFOs $F \in \{F_1..F_K\}$ where K specifies the number of FIFOs of this VCA. Each FIFO can be used to pass on data from one task to another. For parallel task execution on multiple PUs, communication buffers are essential. A task running on PU P_i can

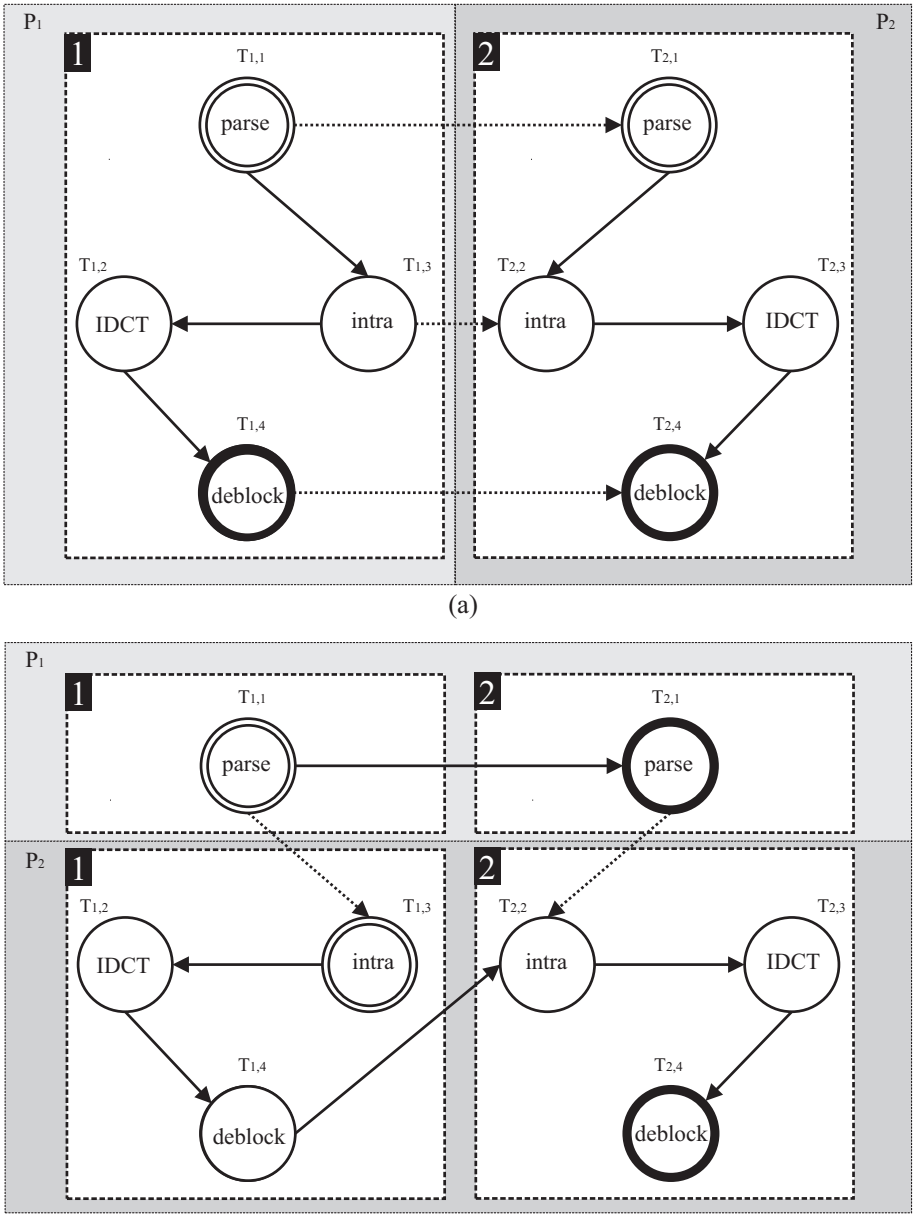


Figure 5.5: The figure visualises the mapping of a VCA's graph onto a platform with multiple PUs: (a) The decoding tasks of MB_1 and MB_2 are assigned to PUs P_1 and P_2, respectively. This represents a data-parallel macroblock decoding approach. (b) A functional partitioning of the decoder assigns the parser tasks of both MBs to PU P_1 and the remaining tasks to P_2.

store its results for dependent tasks of other PUs that are not yet able to process this data. P_i can continue executing its assigned tasks. Without communication buffers, it would have to stall

until all dependent tasks have read the data (i.e. all data-dependencies are solved).

Within our VCA description we assign two FIFOs, an input and an output FIFO, to each task T_i:

$$T_i \rightarrow (F_{in}, F_{out}), F_{\{in,out\}} \in F \tag{5.8}$$

We refer to the input and output FIFOs of task T_i as $F_{in}(T_i)$ and $F_{out}(T_i)$, respectively. Task T_i reads its input data from $F_{in}(T_i)$ and writes its results to $F_{out}(T_i)$. A data-dependency D_j between two tasks is solved by passing on data between these tasks using a shared FIFO:

$$D_j : T_a \rightarrow T_b \Rightarrow F_{out}(T_a) \equiv F_{in}(T_b) \tag{5.9}$$

While FIFO communication is obviously necessary for describing data communication between tasks of different PUs, also sequential VCA execution (i.e. running on a single-PU) requires a model for describing the data exchange between tasks. In a single-PU scenario such a model allows us to describe the storage of data that is too large to be kept in processor registers until the next task is able to process it. For example, a task that processes a whole image will store the image data in a local memory buffer where consecutive tasks can access it.

5.2.2 Characterisation

After the specification of the VCS's components, the *characterisation* takes place. In this step, the VCA's tasks are set in context of physical hardware. Currently, the PAS supports the following task information that is considered during simulation:

- *Processing time*: Duration of a task for executing on its assigned processor.
- *Transfer size*: Amount of data that is exchanged between the dependent tasks.
- *Transfer times*: Times required for transferring input data to a task and for writing the task's results to an output buffer.

For specifying how long a task is processed by a specific PU, information on each task's complexity must be provided to the PAS.

In Chapter 5.2.1, we have introduced the term duration. The system designer specifies the processing duration $duration_p(T_i, P_j)$ of a task T_i running on PU P_j by using either clock cycles or seconds, for example:

$$duration_p(T_i, P_j) \; = \; 2500\, cycles \; = \; \frac{2500}{clock(P_j)}\, seconds \tag{5.10}$$

where $clock(P_j)$ refers to the clock rate of PU P_j. Depending on the physical hardware the PU is based on (e.g. processor type, hardware extensions, etc.), the duration of task T_i can vary. For a VCS with N PUs, the PAS allows the system designer to specify the task's duration for each PU individually.

The PAS uses the FIFO communication buffers to specify the communication behaviour between tasks. Since in a real-world scenario FIFO buffers are limited in the amount of information

they can store, the system designer assigns a *buffer size* $size(F_j)$ (e.g. in bytes or number of MBs) to each FIFO F_j. This size determines the maximal amount of data that can be stored within this buffer.

A task can store its result data in its output FIFO if sufficient space within the buffer is available. Similar to dependencies, we define a set of write conditions $C = C_1..C_K$. Each write condition C_k determines an output resource that is required by a task T_i for writing its results to its assigned output FIFO $F_{out}(T_i)$:

$$C_k : size(T_i, F_{out}(T_i)) \leq free(F_{out}(T_i)) \tag{5.11}$$

In Equation 5.11, $size(T_i, F_{out}(T_i))$ refers, for example, to the number of bytes that task T_i writes into its output FIFO $F_{out}(T_i)$ and $free(F_{out}(T_i))$ to the amount of free memory in this FIFO. Task T_i has to stall if one of his write conditions cannot be fulfilled (i.e. task T_i cannot write its results into its output buffer due to insufficient memory).

Next to the memory requirements also the transfer time for storing/loading a block of data to/from a FIFO has to be provided. In the PAS, we specify the read duration $duration_r(T_i)$ that it takes task T_i to read its input data from its input FIFO $F_{in}(T_i)$. In the same way we specify a write duration $duration_w(T_i)$ for writing the task's output to its corresponding output FIFO $F_{out}(T_i)$.

5.2.3 Simulation

Based on the specification and characterisation of the VCS, simulation of the runtime behaviour and estimation of the overall runtime can be done. This section explains how the dependency graphs that build the foundation of our VCA description can be resolved efficiently.

In a realistic multi-core VCA, the tasks will strongly depend on each other. These dependencies stem from the simple fact that one task usually needs the results of one or more other tasks as an input. The proposed PAS is capable of computing the overall runtime correctly with respect to such dependencies. The basic algorithm to accomplish this simulation is introduced in Figure 5.6. In the example of Figure 5.6, we only consider a two-core system. However, the algorithm can easily be extended to handle an arbitrary number of processors. We go into more detail on the algorithm in the following.

The algorithm of Figure 5.6 maintains three different kinds of sets. First, a set $\mathcal{T}_i$ represents all tasks that have been executed on processor P_i. Second, we use sets $\mathcal{F}_t$ to keep track of all those tasks that are already completed at a specific instance of time t. Third, for each task T a dependency set $\mathcal{D}_T$ is introduced, which is a list of all tasks that already need to be computed before we can start execution of the task T.

The main loop of the algorithm (Lines 16-43) is iterated until all tasks have been executed. For each processor, we determine the task T that shall be executed next (Line 24). This task can only be executed if its dependencies are already resolved. We therefore iterate through the dependency list of T (Lines 28-33) to check whether all depending tasks are already member of the finished task list $\mathcal{F}_t$ at the current time t. If this is the case, T can be executed. We add T to the list of finished tasks at time $t + duration_p(T)$ and remove T from the list of tasks that are

1: // lists of tasks that are executed on two processors
2: $\mathcal{T}_1 = \{T_{1,1}, T_{2,1}, \cdots, T_{n_1,1}\}$;
3: $\mathcal{T}_2 = \{T_{1,2}, T_{2,2}, \cdots, T_{n_2,2}\}$;
4: // list of tasks finished at times $0, 1, \cdots, t_{max}$
5: $\mathcal{F}_0 = \mathcal{F}_1 = \cdots = \mathcal{F}_{t_{max}} = \emptyset$;
6: // specify intra dependencies (just shown for processor P_1)
7: $\mathcal{D}_{T_{1,1}} = \emptyset$;
8: $\mathcal{D}_{T_{2,1}} = \{T_{1,1}\}$;
9: $\vdots$
10: $\mathcal{D}_{T_{n_1,1}} = \{T_{1,1}, \cdots, T_{n_1-1,1}\}$;
11: // add an inter-dependency (e.g, task 3 of processor P_1
12: // needs to wait for task T_2 of processor P_2)
13: $\mathcal{D}_{T_{3,1}} = \mathcal{D}_{T_{3,1}} \cup T_{2,2}$;
14: // the current time
15: $t = 1$;
16: // loop while there are still tasks that need to be processed
17: **while** $\mathcal{T}_1 \neq \emptyset \wedge \mathcal{T}_2 \neq \emptyset$ **do**
18: // tasks that are already finished at time $t-1$ are also
19: // already finished at time t
20: $\mathcal{F}_t = \mathcal{F}_t \cup \mathcal{F}_{t-1}$;
21: // for both processors
22: **for** $i = 1$ to 2 **do**
23: // access the first entry in the task list of processor P_i
24: $T = \mathcal{T}_i[1]$;
25: // check if all dependencies for T are resolved
26: $dependencies\_resolved = true$;
27: // go through the dependency list of T
28: **for** $j = 1$ to $|\mathcal{D}_T|$ **do**
29: // check if the task on which T depends has already finished at time t
30: **if** $\mathcal{D}_T[j] \notin \mathcal{F}_t$ **then**
31: dependencies_resolved = false;
32: **end if**
33: **end for**
34: // in case that all dependencies are resolved
35: **if** dependencies_resolved == true **then**
36: // add T to list of finished tasks at time $t + duration_p(T)$
37: $\mathcal{F}_{t+duration_p(T)} = \mathcal{F}_{t+duration_p(T)} \cup T$;
38: // remove T from the task list
39: $\mathcal{T}_i = \mathcal{T}_i - T$;
40: **end if**
41: **end for**
42: $t = t + 1$;
43: **end while**
44: // computed execution time of the multi-core system
45: **return** t;

Figure 5.6: Algorithm for simulating parallel task execution in a VCA. A detailed explanation is given in the text.

still waiting for execution (Lines 37-39). After the algorithm has left the main loop, the overall computation is determined from the value of the variable t (Line 45).

It is important to note that the algorithm in Figure 5.6 only considers read-dependencies between tasks so far. For addressing FIFO buffer limitations and their impact on the overall processing time, we can extend this algorithm by adding an additional iteration after Lines 28-33 for validating that the data resulting from a task can be written to the task's output FIFO.

5.3 Implementation of the Partition Assessment Simulation

In this section, the implementation aspects of the PAS are described. Section 5.3.1 outlines how time and complexity are treated within the PAS. In Section 5.3.2, we describe how the PAS can exploit information from DDPs to automatically specify tasks and dependencies. Section 5.3.4 focuses on the implementation aspects related to virtual hardware prototyping and VCA partitioning onto a multi-processor platform. The simulation of our VCS is explained in Section 5.3.5.

5.3.1 Time domains within PAS

The general understanding of time and complexity in the context of a multi-core platform is of great importance. In a real-world VCA running on a multi-processor platform, it affects essential aspects such as the processing time required to execute a task or the synchronisation between the PUs.

The PAS implementation differentiates between *local time* and *global time*. The PAS considers each PU in the VCS as an independent system with its own local time system and its local time counter. Whenever a PU executes a task, the PU's local time counter is incremented by the duration of this task and corresponding data transfer times. The local time counter $t_{local,i}$ indicates until which time the PU P_i has been simulated by the PAS and what simulation data (e.g. the start and end time of all tasks that have been executed till $t_{local,i}$) is available.

The global time t_{global} works as reference to coordinate the individual components of our VCS during the simulation. Each value of a PU's local time counter can be mapped to the global time and vice versa:

$$t_{local,i} \Leftrightarrow t_{global} \tag{5.12}$$

This also enables the translation between different local time systems. For example, the local time $t_{local,i}$ of a PU P_i can be translated into the global time and then further into a local time $t_{local,j}$ of another PU P_j:

$$t_{local,i} \Rightarrow t_{global} \Rightarrow t_{local,j} \tag{5.13}$$

The ability to translate between local and global time enables us to analyse a VCS at a specific global time t_{global}. This enables the PAS to address an essential aspect of a multi-processor execution behaviour: the synchronisation between tasks of different PUs. For example, considering a scenario with two tasks T_i and T_j running on two different PUs P_k and P_m, respectively,

if T_i depends on T_j, then in order to evaluate whether this dependency is resolved, the time when the results from T_j become available must be known in relation to the local time system of PU P_i.

Since FIFO buffers can be used by tasks of different PUs and, consequently, within multiple local time systems, global time values are used to describe changes in the fill status of a FIFO. The translation from global into local time systems enables us to determine the fill status of a FIFO at each PU and to consider this when simulating task execution on this PU.

5.3.2 Task generation based on data-driven profiling

In the previous chapter, a simple VCA with a small number of MBs and tasks was used to outline the concept behind our modelling methodology. However, typical real-world VCAs have large numbers of tasks and dependencies, and so a manual specification and characterisation becomes intractable. In the PAS, means for automatically deriving the tasks of a VCA from a DDP are provided. Three major steps are automatically done by the PAS:

1. ***Task specification***: Generation of VCA tasks from the DDP based on profiling rules.
2. ***Complexity characterisation***: Assignment of complexity information to individual tasks.
3. ***Coding characterisation***: Assignment of application-specific coding attributes such as MB, slice and frame number to individual tasks.

Exploiting the information available in a VCA's DDP enables us to map the VCA's complexity onto specific tasks. We can use this information to automatically retrieve the vertices of our dependency graph from these profilings. For each task with a profiled runtime complexity greater than 0 cycles, a PAS task is created. The PAS stores all information that is required by the PAS for the VCS simulation in a data structure. After the automatic task specification, this structure contains information about the task processing duration ($duration_p(T_i)$) and VCA-specific coding information such as the MB/slice/frame number the task is processing.

5.3.3 Rule-based specification of data-dependencies

Specifying the dependencies of a VCA in an efficient and semi-automatic way is supported by the PAS. The designer can provide dependency rules (DRs) that describe the relation between a VCA's tasks and enable the PAS to automatically introduce task dependencies into a VCA's dependency graph.

In general, all DRs use the FB type and the coding information of a task for specifying the relation between the VCA's tasks. For example in the H.264 decoder, the entropy decoding of a MB always has to be done before the MB's intra-prediction. Hence, if the functional type (e.g. "entropy" or "intra") of the tasks is known from the DDPs, a DR for formalizing this task relation can be created. An example of how such a DR would be specified is visualised in Equation 5.14. In the PAS, tasks T_a and T_b are considered as structures that contain coding information such as the MB number $T_*.mb$ and functional blocks $T_*.fb$:

$$DR(T_a, T_b): \quad \begin{array}{ll} (T_a.mb = T_b.mb) & \wedge \\ (T_a.fb = 'entropy') & \wedge \\ (T_b.fb = 'intra') & \Longrightarrow D: T_a \rightarrow T_b \end{array} \tag{5.14}$$

After specifying the VCA's tasks using the DDP, the PAS uses the defined DRs for automatically specifying all VCA dependencies. It is important to note that the number of dependencies (and hence the number of required DRs) strongly depends on the granularity where the parallelisation takes place. For example, in a slice-based H.264 decoder partitioning, few dependencies, mostly between slices of consecutive frames (i.e. inter-prediction), exist. For a MB-based decoder partitioning, a larger number of DRs must be specified to describe the dependencies between the tasks. Apart from parallelisation granularity, also the VCA's underlying coding algorithm's complexity has a direct impact on the number of necessary DRs. Depending on the number of coding tools supported by a coding algorithm, the amount of data dependencies can strongly vary.

5.3.4 Partitioning of video coding application

For describing a virtual hardware platform, the PUs and the communication FIFOs must be specified. An abitrary number of PUs can be created within the PAS. For each PU, a local time counter is maintained by the PAS. During the simulation of a VCS, this counter specifies the time this PU has been simulated so far. The FIFOs for data-exchange between tasks are created by specifying the number of FIFOs and the size of each FIFO.

After the specification, the mapping of our VCA's tasks onto the hardware platform is done. The PAS supports this software-hardware mapping by three types of functions:

- Processor Assignment Functions (PAFs) for assignment of tasks to PUs
- FIFO Assignment Functions (FAFs) for assignment of tasks to FIFOs
- Memory Access Functions (MAFs) for describing the transfer behaviour between tasks and FIFOs

These functions can be defined in the PAS using the Matlab high-level language syntax, which results in a high flexibility when adapting the virtual hardware and software architecture to new partitioning approaches.

Processor and FIFO assignment

PAFs enable the assignment of the VCA's tasks to PUs after the tasks and dependencies have been specified. Similar to wildcards, each PAF searches for tasks with specific coding attributes (e.g. MB number, coding type) and assigns these tasks to a specific PU. This enables an efficient and abitrary partitioning of the VCA onto the PUs. For example, data-parallel and functional partitionings of the VCA can be defined by assigning MBs of a particular:

- slice/frame/frame row/frame column to a PU

- function type (e.g. “entropy’,’intra’) to a PU
- coding type (e.g. intra coded) to a PU

The flexibility of this assignment enables fast partitioning of the VCA. For simulating different partitionings of the VCA, only the adaptation of the PAFs is necessary. The PAS can automatically derive the new VCS and provide a runtime estimation for this new partitioning.

Each task within the VCA description has to be assigned to an input and output FIFO that enables the PAS to model data transfer behaviour between tasks. Similar to PAFs, this assignment is done using FAFs and uses coding information for assigning tasks with specific codings to the same FIFO. It should be noted that PAFs and FAFs need coding information for assigning the tasks to PUs and FIFOs. Consequently, these rules require an early characterisation of the tasks before an automatic assignment can take place.

Memory Access Behaviour

The PAS supports the modelling of the MAFs for describing the data transfer behaviour between FIFO and tasks on a specific PU. The PAS provides linear models for estimating the transfer time of moving data between FIFOs and PUs as well as more complex models that can describe abstract caching behaviour of task data. For example, a model for estimating the time required for executing a DMA data transfer could have the following form:

$$duration_w(T_i) = \frac{T_i.size_{out}}{transfer\_rate(T_i.fifo_{out}, T_i.pid)} + latency(T_i.fifo_{out}) \qquad (5.15)$$

In this equation, task T_i writes its output data to its assigned output FIFO $T_i.fifo_{out}$. The duration this write transfer requires is estimated based on the task’s output data’s size $T_i.size_{out}$, the transfer rate between the executing PU $T_i.pid$ and its output FIFO. Furthermore, the model assumes that a latency $latency(T_i.fifo_{out})$ for initiating a data transfer to the output FIFO occurs. For each task, the PAS automatically finds the corresponding MAFs by analysing the task’s assigned PU and FIFOs and computes the read and write duration for transferring data to/from the task.

Based on the available task information, more complex MAFs can be defined that also consider caching strategies and data cache sizes based on the PAS simulations. This makes it possible, for example, to estimate how well data locality can exploited for a partitioned decoder mapping.

5.3.5 Simulation process

The example in Figure 5.7 will be used to depict the simulation approach. In this figure, four MBs are decoded on two processing units. The processing of each MB consists of a parsing and entropy decoding task and a reconstruction task that performs all pixel reconstructions based on the parsed MB bitstream information. In this example, all parsing tasks are executed on processor P_1, and all pixel reconstruction tasks on processor P_2, which corresponds to a functional decoder splitting.

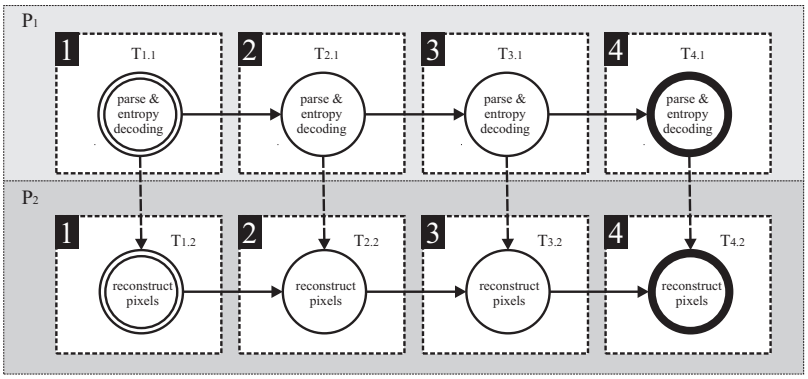


Figure 5.7: Functional partitioning of four macroblocks: For each macroblock two decoding tasks, "parse & entropy decoding" and "reconstruct pixels", are executed. The parsing and reconstructing tasks are executed on processors P_1 and P_2, respectively.

Figure 5.8 illustrates how the PAS simulation works for this example. The decoding process of the first three MBs is shown in six simulation steps. Processor P_1 executes the first decoding task $T_{1,1}$ (i.e. parsing and entropy decoding) and writes the results to FIFO F_1. In this example, the buffer size of F_1 is limited to one macroblock. Processor P_2 reads the results from FIFO F_1 and applies the pixel reconstruction task $T_{1,2}$. Internally, the PAS maintains a list for each PU and FIFO buffer. It stores the states and the filling levels of each PU and FIFO. The simulator sequentially processes one macroblock decoding task after another.

The PAS uses the task assignment to determine the PU executing a decoding task. It evaluates when this PU can start to execute a task. In Figure 5.8a, this is indicated by the white marker (S). The processor's execution counter is increased by the task's duration. The black marker (E) indicates the end of a task. At this point the processor has finished the task execution and written its results to the output buffer FIFO F_1. At this stage, the PAS does not know when the data is removed by another task and marks the state of FIFO F_1 as occupied.

In Figure 5.8b, processor P_2 reads the MB data from FIFO F_1 and frees the occupied memory in this FIFO. During the decoding of a MB, functional dependencies between the decoding tasks and data dependencies between the individual MBs exist. We have presented MB dependencies for the H.264 codec in Section 3.5.1. In Figure 5.8b, a read stall due to data-dependencies is illustrated. Processor P_2 cannot start its decoding operations simultaneously with processor P_1, but has to wait until the required data becomes available in the input buffer. The PAS uses the algorithm description for detecting this read stall. The start of the task execution is delayed automatically.

In Figure 5.8c, the second MB is executed by processor P_1 and written to FIFO F_1. After processor P_2 has finished its decoding operations for MB_1, it reads MB_2 and executes it (Figure 5.8d).

Until now only the impact of computational complexity on our multi-core decoding system has been considered. Additionally, the PAS checks for buffer size constraints. For each task,

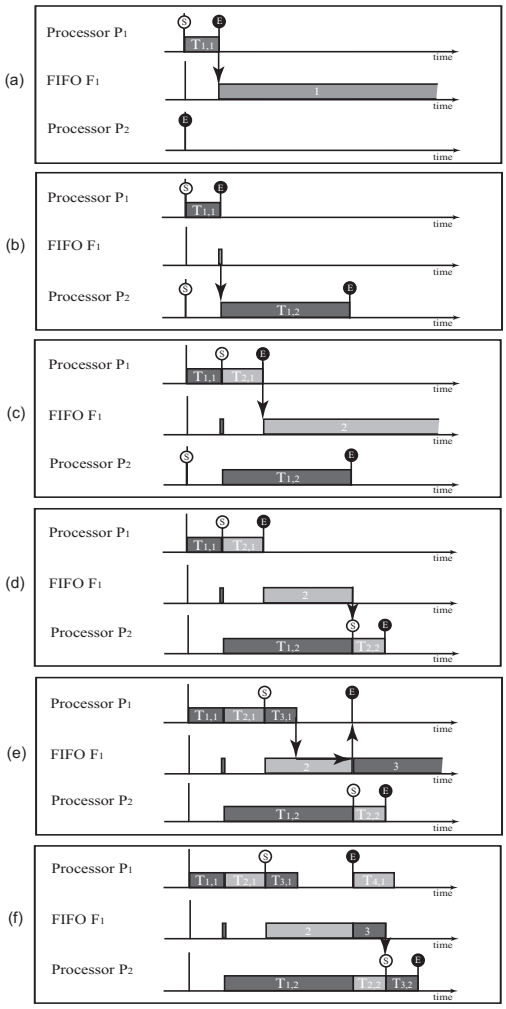


Figure 5.8: Visualisation of the internal simulation process in the PAS. The figure shows the execution states and buffer levels of two PUs and one FIFO, respectively. Three MBs are processed on two individual PUs. For each MB, one task is executed on P_1 and one on P_2. After partially decoding each MB, processor P_1 writes the results to FIFO F_1. For simplicity, the FIFO's maximum size is set to 1 macroblock. Processor P_2 reads the MBs from this buffer and computes the remaining decoding tasks. A detailed explanation is given in the text.

limitations in the buffer communication (e.g., write stalls due to insufficient buffer sizes) can be considered using the FAFs and MAFs. For example, the parsing task requires a certain amount of free memory for writing its results into an output buffer. The decision space can be explored by specifying how much memory is required and how the data transfer influences the runtime using MAFs (e.g., "What happens in the case of insufficient buffer?" or "How does the memory access latency influence the decoder's runtime?").

Figure 5.8e visualises a case where insufficient buffer is available and a write stall occurs. Since FIFO F_1 has a maximum size of 1, processor P_1 cannot write the results of decoding MB 3 immediately. It has to wait until processor P_2 has read MB_2 and freed the occupied memory in FIFO F_1. A write stall occurs in this case. After finishing MB_2, processor P_2 reads MB_3 from FIFO F_1 and decodes it. After MB_3 could be written to FIFO F_1, processor P_1 continues with the next MB.

For determining write stalls between tasks, the PAS internally computes the amount of data which is exchanged between dependent tasks. It uses the user-defined communication FIFOs and connects the tasks to these FIFOs. For each task, how much data is read/written to a FIFO is specified. This information can be extracted from the decoder's source code (e.g. "the deblocking task reads X bytes"). For each FIFO, the PAS maintains a list of all read and write operations that occur throughout the execution of VCA. This enables the PAS to automatically compute the FIFO levels for each point in time throughout the VCA's execution. The PAS delays a task if insufficient memory for writing the task's results is available.

Using the MAFs, the time a task requires for reading/writing to a FIFO can be specified. It facilitates modelling of different memory properties such as read and write duration, latency and access time. The PAS delays the writing tasks automatically based on the MAFs.

5.4 Summary

In this chapter, we have introduced general aspects and design goals for VCA partitioning. The flexibility and descriptive clarity for modelling VCAs to different hardware partitionings and to exploit available knowledge (e.g. from hardware profilings, algorithm knowledge, etc.) are considered as the prime requirements of such a partitioning approach. We have introduced the PAS concept for describing a VCA in an abstract way and for mapping the VCA onto a virtual hardware platform. During a specification step, information on the VCA structure, the virtual hardware platform and the partitioning is defined. The characterisation step introduces information about runtime complexity, transfer behaviour and physical constraints. The last stage, the simulation, combines all this information to estimate the runtime as well as the memory transfer behaviour of our application.

An implementation of the PAS has been provided. It enables detailed analysis of implementation aspects of our proposed concept such as concurrent simulation. We have provided functionality for fast generation of VCA descriptions from DDPs. Furthermore, mechanisms for assigning tasks to PUs and FIFOs have been introduced that enable the system developer to describe complex platforms and VCAs with low effort.

CHAPTER 6

Concept verification and design space exploration results

In this chapter, we use the PAS methodoloy for modelling various real-world partitioning scenarios of H.264 decoders. We verify the PAS methodology and demonstrate its usage in the following way. First, we use the PAS for modelling a single-core decoder running on one core of an existing dual-core reference architecture (Section 6.1). Second, we derive function traces of a single-core decoder running on this architecture in Section 6.2. By calibrating the PAS model according to these hardware profiles, we can model the hardware characteristics of our architecture within the PAS. In Section 6.3, we verify the usage of the PAS using a dual-core H.264 decoder running on this reference architecture. By comparing the runtime behaviour of the reference dual-core decoder system with the PAS simulation of the same VCS, the accuracy of the PAS is estimated. Third, we demonstrate how the PAS can be applied for fast design space explorations in Section 6.4.

6.1 Specification of a dual-core video coding system

For this evaluation, the same reference architecture as described in Chapter 4.3.1 has been used (Figure 4.4). The decoding in this VCS works as follows: The ARM processor receives the compressed video data as an MPEG-2 transport stream. It extracts the compressed H.264 video data and writes it to the external mDDR memory via DMA transfer. One CHILI processor uses DMA transfers for fetching the H.264 video data into the core's faster 64 kB local data memory and starts decoding it. This PU executes all parsing and entropy decoding tasks. The results of this process are stored into a shared memory (SRAM). The second CHILI processor reads the data from the SRAM and executes the reconstruction tasks before it writes the results into the external DDR memory. This reconstruction includes all pixel-manipulation tasks such as prediction and deblocking. The decoded pixel information is transferred via DMA to the DCCs framebuffer memories. These framebuffer memories are located on the external mDDR memory.

System specification and characterisation of tasks

In the first step, we profile a single-core decoder and generate DDPs obtained as explained in Chapter 4.3. Based on the DDPs we can specify the tasks of our VCS in an automatic way. During the DDP-based task specification, complexity information of each task for the individual PUs is obtained as well. After the assignment of a task to a PU is known (i.e. the task has been assigned to a PU of our architecture), the complexity information from the specific DDP obtained on this PU serves as the processing duration of the task during the PAS simulation.

The second step consists of defining the available PUs and FIFOs of our system. One ARM processor for transport stream multiplexing and two CHILI processors for executing the computationally intensive video processing tasks are defined. Furthermore, communication FIFOs (mDDR, SRAM and local processor memories) for data exchange between the tasks are specified according to the reference architecture.

Before starting the exploration of new designs, we have to calibrate the PAS to match the characteristics of our reference hardware. This procedure is explained in the next section.

6.2 Characterisation of virtual hardware

After specifying the structure of our VCS, we calibrate the VCS according to the target hardware. This step includes characterisation of the PUs and memory transfer behaviour. Characterisation of the PUs is done by specifying the clock rate of the processors within our VCS. In this case, a clock rate of 300 MHz of our reference system has been used.

For characterizing the memory access behaviour of our VCS, we have to calibrate the memory transfer times of the PAS to fit the transfer times of our reference architecture. For obtaining these transfer times from the DDP of the single-core H.264 decoder, we assign all tasks of the VCA to a single CHILI processor and adjusted the memory model until both, the PAS simulated single-core decoder and the single-core decoder profiled on a CHILI processor have approximately the same execution runtime. Remember the equation provided in Section 5.3.4 for describing the memory write access behaviour as a linear function of the amount of transferred data, the transfer rate and the latency introduced by initiating a memory transfer:

$$duration_w(T_i) = \frac{T_i.size_{out}}{transfer\_rate(T_i.fifo_{out}, T_i.pid)} + latency(T_i.fifo_{out}) \tag{6.1}$$

For specifying a memory model for our concept verification, we have assumed that 4 bytes (32 bits) per clock cycle can be transferred over the 32-bit data bus of our system (i.e. a transfer rate of 4 bytes per clock cycle) and that a fixed latency for initiating the memory transfer is required. We have specified the duration of memory read and write transfers using the following equations:

$$duration_r(T_i) = \frac{T_i.size_{in}}{4} + latency(T_i.fifo_{in}) \tag{6.2}$$

$$duration_w(T_i) = \frac{T_i.size_{out}}{4} + latency(T_i.fifo_{out}) \tag{6.3}$$

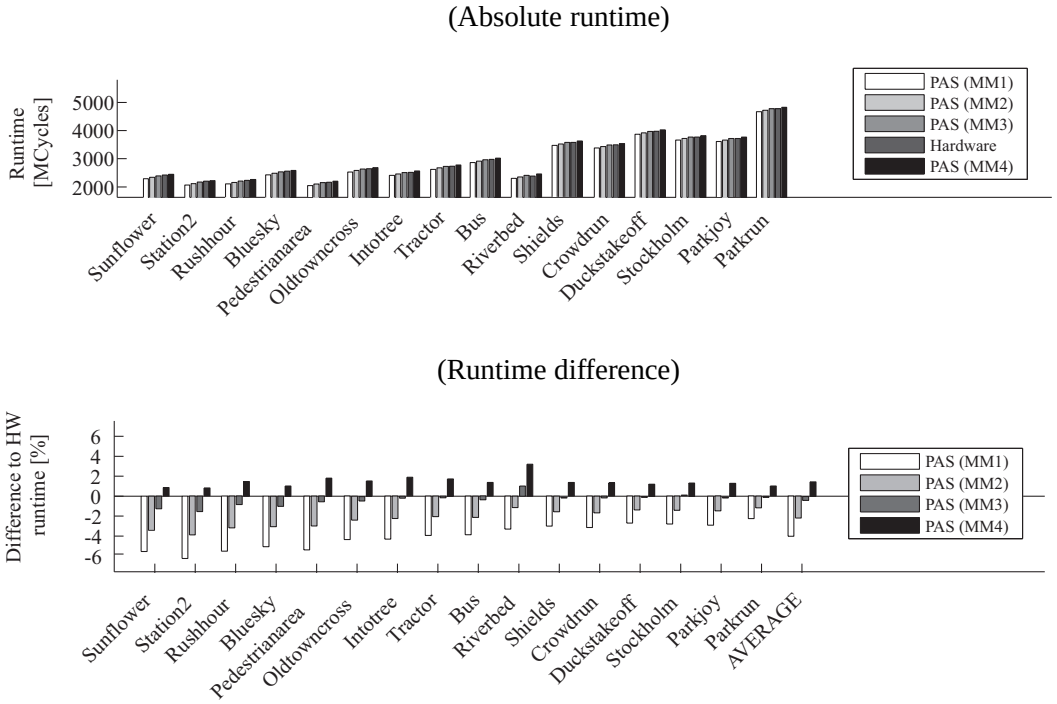


Figure 6.1: Calibration of the PAS memory models: The PAS settings are calibrated according to the single-core profilings. The absolute runtime and the relative runtime difference between the HW single-core implementation and the PAS results after calibration are provided.

For specifying a realistic latency, four different memory models (MM1-4) with different latencies have been compared against the hardware profiling results. Latencies $latency(T_i.fifo_{in/out})$ of 4, 7, 9 and 16 cycles for memory models MM1, MM2, MM3 and MM4 have been set, respectively. It should be noted that the memory model can be extended if more information on the hardware is available.

Figure 6.1 shows the comparison between the simulated and the measured runtime after the calibration. The absolute runtime obtained for each model is visualised for 16 test sequences. Furthermore, the runtime difference between the PAS simulated single-core decoder runtime and the measured runtime are provided. Of all four MMs, MM1 describes the memory access behaviour with the lowest transfer time for moving data between the PUs and the FIFOs and MM4 the memory model with the slowest memory transfer. We can observe that MM3 shows the best aproximation with a maximal runtime difference of less than one percent between measured and estimated runtime. On average, MM3 has runtime differences of less than one percent. The highest differences can be observed for the “Station2” sequence with close to two percent.

The information from DDPs and PAS simulation enable us to compare the measured and the simulated runtime behaviour in more detail. Figures 6.2 and 6.3 show the relative and the absolute difference between measured and simulated runtime over time. This enables us to estimate the cumulative error that is introduced by our model. In this detailed analysis we can see that MM_3 performs well and shows the smallest cumulative error of the four MMs. In

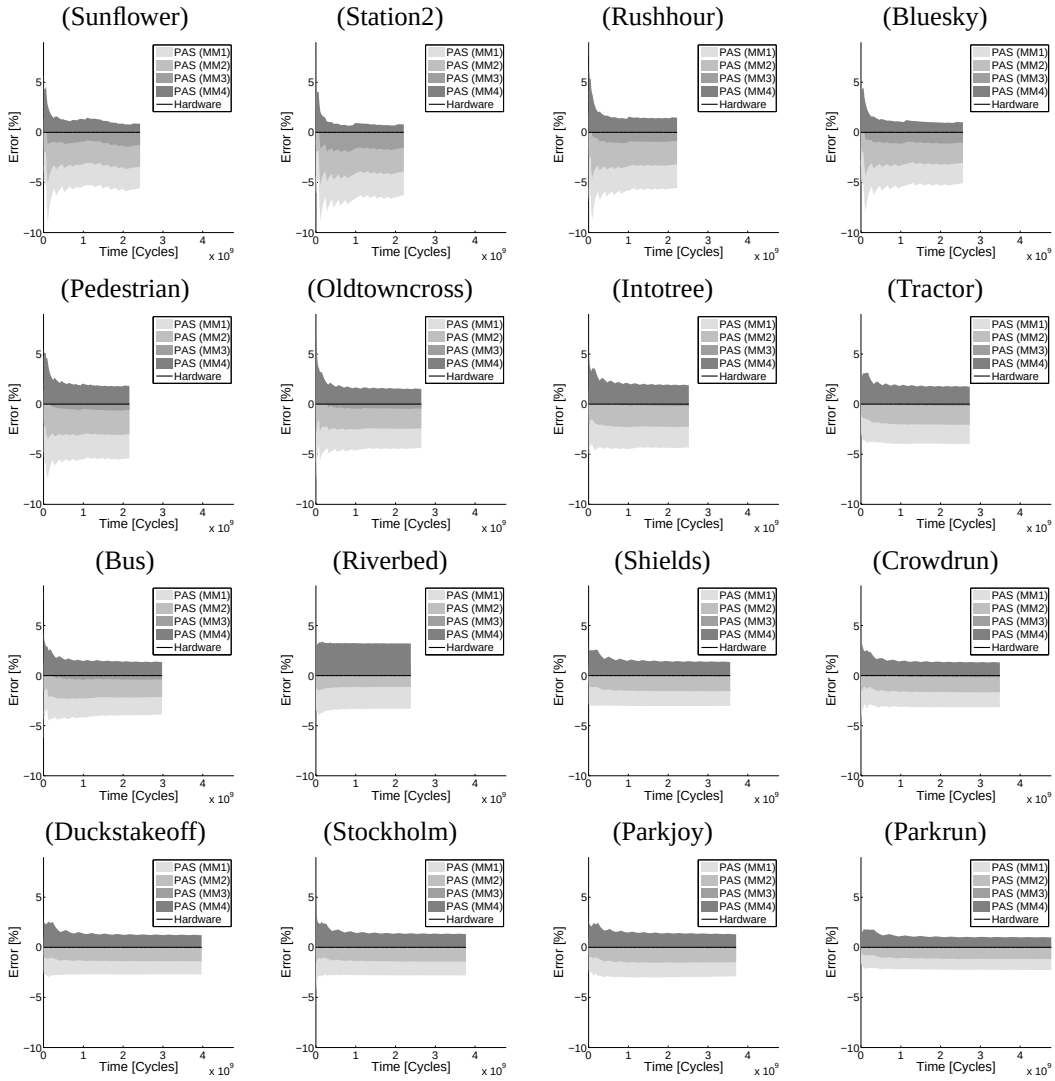


Figure 6.2: Calibration of the PAS memory model: Relative runtime difference in percent between estimated and measured runtime over time for 4 different memory models.

Figure 6.2, it can be observed that for all MMs and test streams the cumulative error stays nearly constant after an initialisation phase of approximately 5×10^8 cycles runtime. This indicates that using linear MMs within our PAS is sufficient for describing the runtime behaviour of our reference single-core decoder. In the following, we use MM_3 for characterising the memory access behaviour in our VCS.

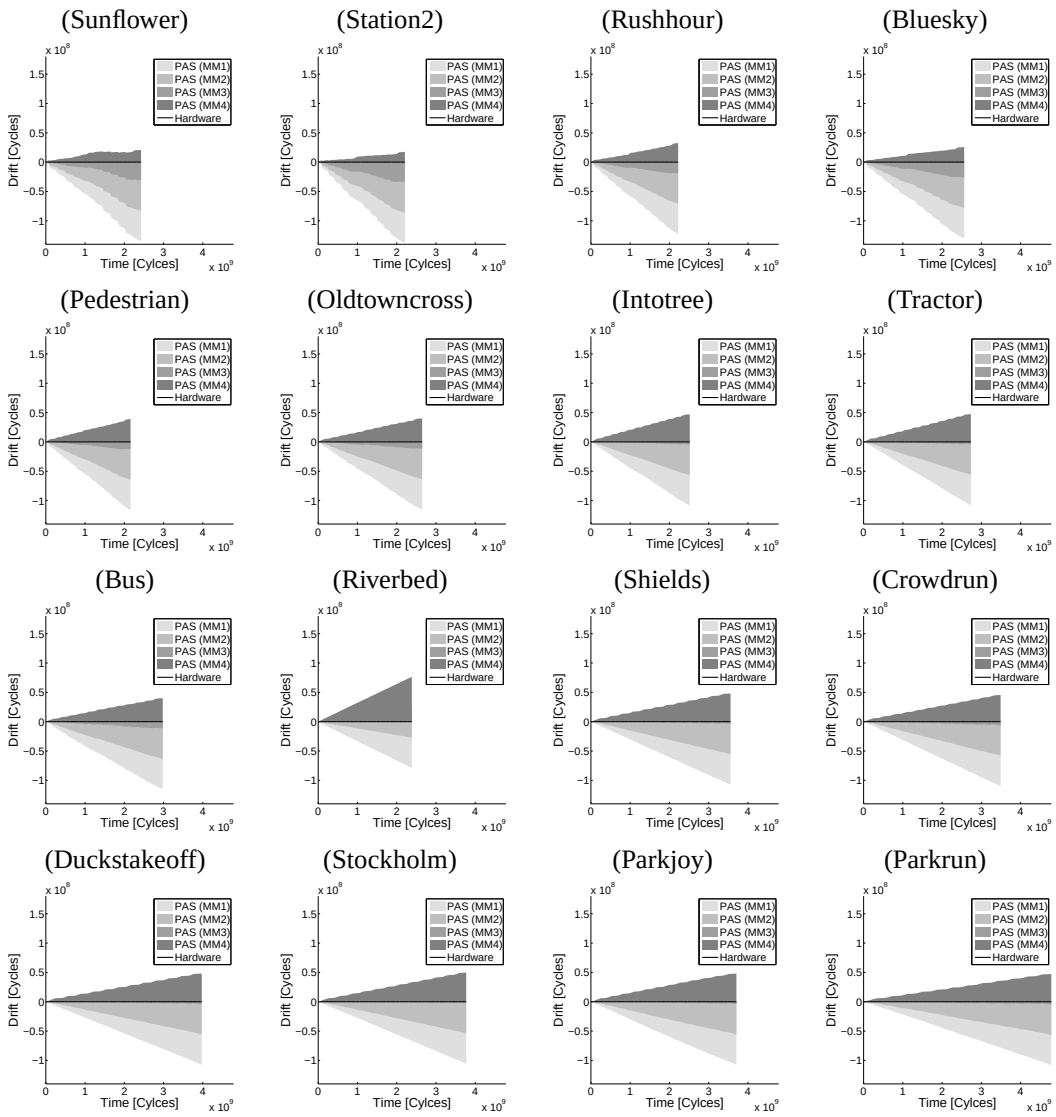


Figure 6.3: Calibration of the PAS memory model: Absolute runtime difference in clock cycles between estimated and measured runtime over time for 4 different memory models.

6.3 Verification using a functional dual-core decoder splitting

After calibrating the PAS, we have evaluated the difference between a measured and a PAS-predicted dual-core decoder runtime behaviour. The decoding process of 16 test sequences was simulated using the PAS. This enabled us to determine the accuracy of the PAS simulator for the simulated VCS for these test sequences. Figure 6.4 provides a comparison between the HW-profiled and the PAS-simulated results. In this figure, the overall runtime is visualised for (i) the measured single-core decoder runtime, (ii) the runtime measured for the dual-core decoder

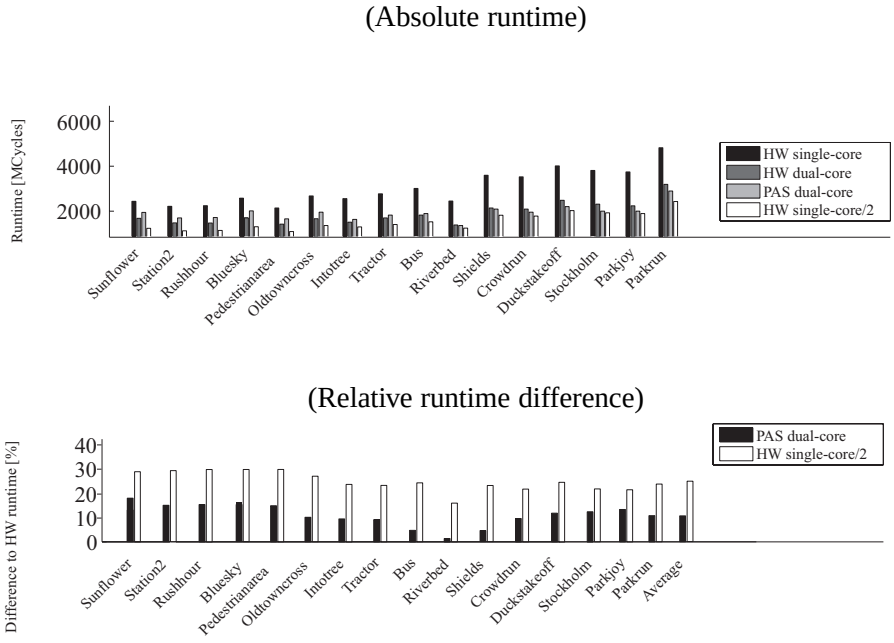


Figure 6.4: Verification of the PAS simulator: The figure provides the absolute runtime of the single-core HW implementation, the dual-core HW implementation, the PAS estimation and a simple estimation technique. This simple estimation technique divides the runtime by the number of cores (e.g. for two cores by a factor of 2). Furthermore, the relative runtime difference between the dual-core HW implementation and PAS estimated runtime as well as the simple estimation technique is provided. In both figures, the PAS estimation clearly outperforms the simple estimation technique.

implementation, (iii) the estimated runtime derived from the PAS simulation and (iv) a runtime estimation based on a simple estimation technique. This estimation technique assumes that the workload can be divided equally amongst all available PUs and divides the runtime measured for the single-core implementation by the number of available PUs.

An average relative error of around 11.5 percent and a maximal relative error of around 18.5 percent can be observed for the PAS over the 16 test sequences. This clearly outperforms the simple estimation technique that assumes that both PUs can "divide" the tasks equally amongst them and reduce the runtime by 50%. This assumption is not capable of addressing differences in the balancing between the individual PUs. This can be observed in Figure 6.4 in the large relative runtime difference of on average 25 percent for all test sequences.

It can be noted that for sequences with higher bitrates such as "Parkrun", "Parkjoy" and "Stockholm", the PAS provides better estimations between 2 and 13.5 percent than for sequences with lower bitrates where higher relative differences of up to 18.5 percent can be observed. This indicates that the PAS model describes the dynamic behaviour of our decoding system less accurately for low bitrate scenarios.

Figures 6.5 and 6.6 provide the relative and absolute difference between measured and es-

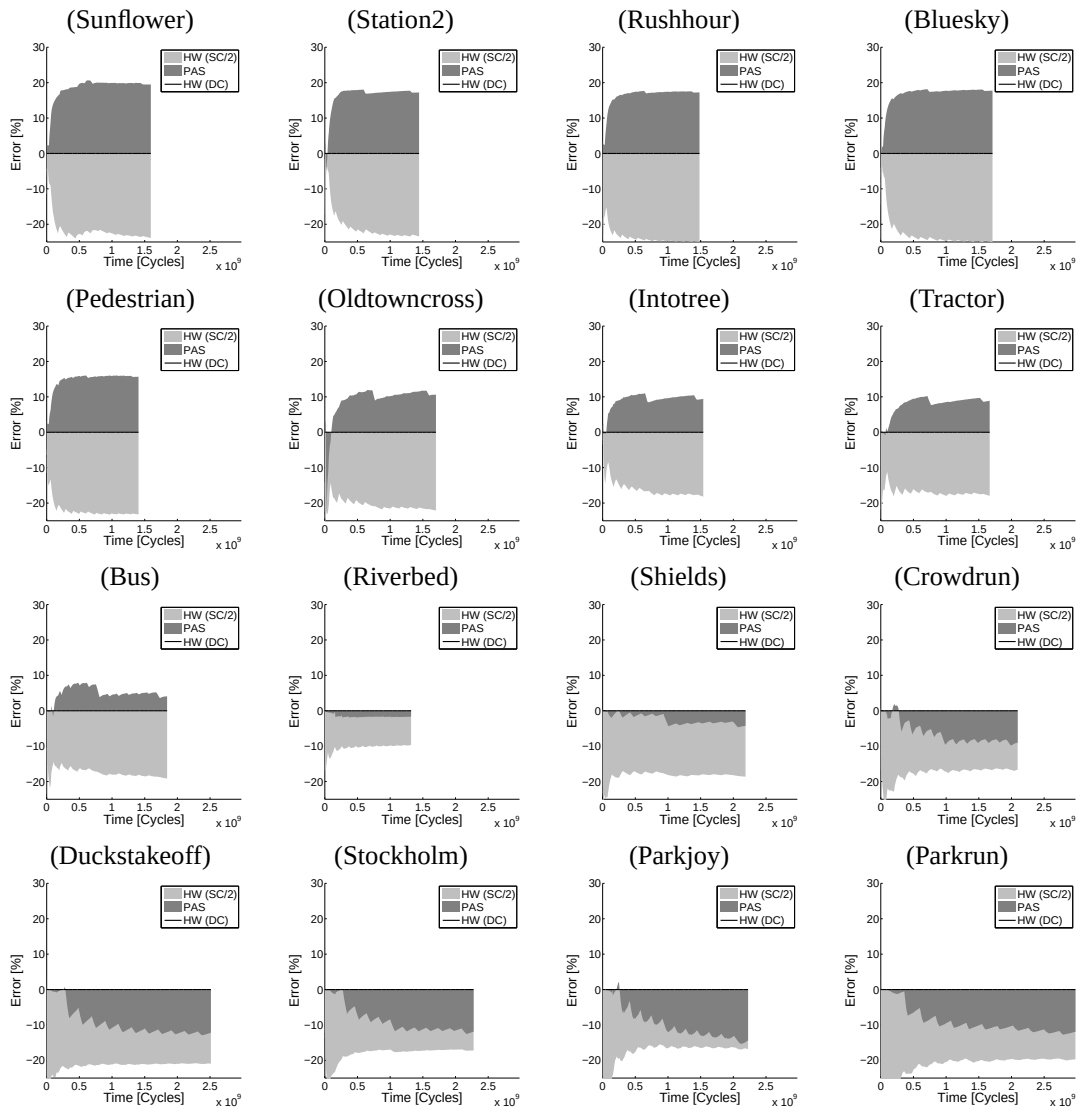


Figure 6.5: Verification of the PAS simulator: The figure shows the relative runtime difference between the runtime HW(DC) measured on a dual-core system, the runtime estimations by the PAS and a simple runtime estimation HW(SC/2) that divides the runtime measured on a single-core system by the number of available cores.

timated runtime, respectively. For each MB, we have measured the time when the decoding of this MB has been finished and compared it with the time estimated by the PAS. We can see in Figure 6.5 that the simple estimation technique, in general, is too optimistic and underestimates the runtime by typically more than 20%. The PAS tends to slightly underestimate the runtime for high-bitrate sequences but provides good estimations for moderate and high-bitrate sequences with a relative error of not larger than 13.5%. For sequences with very low bitrates such as

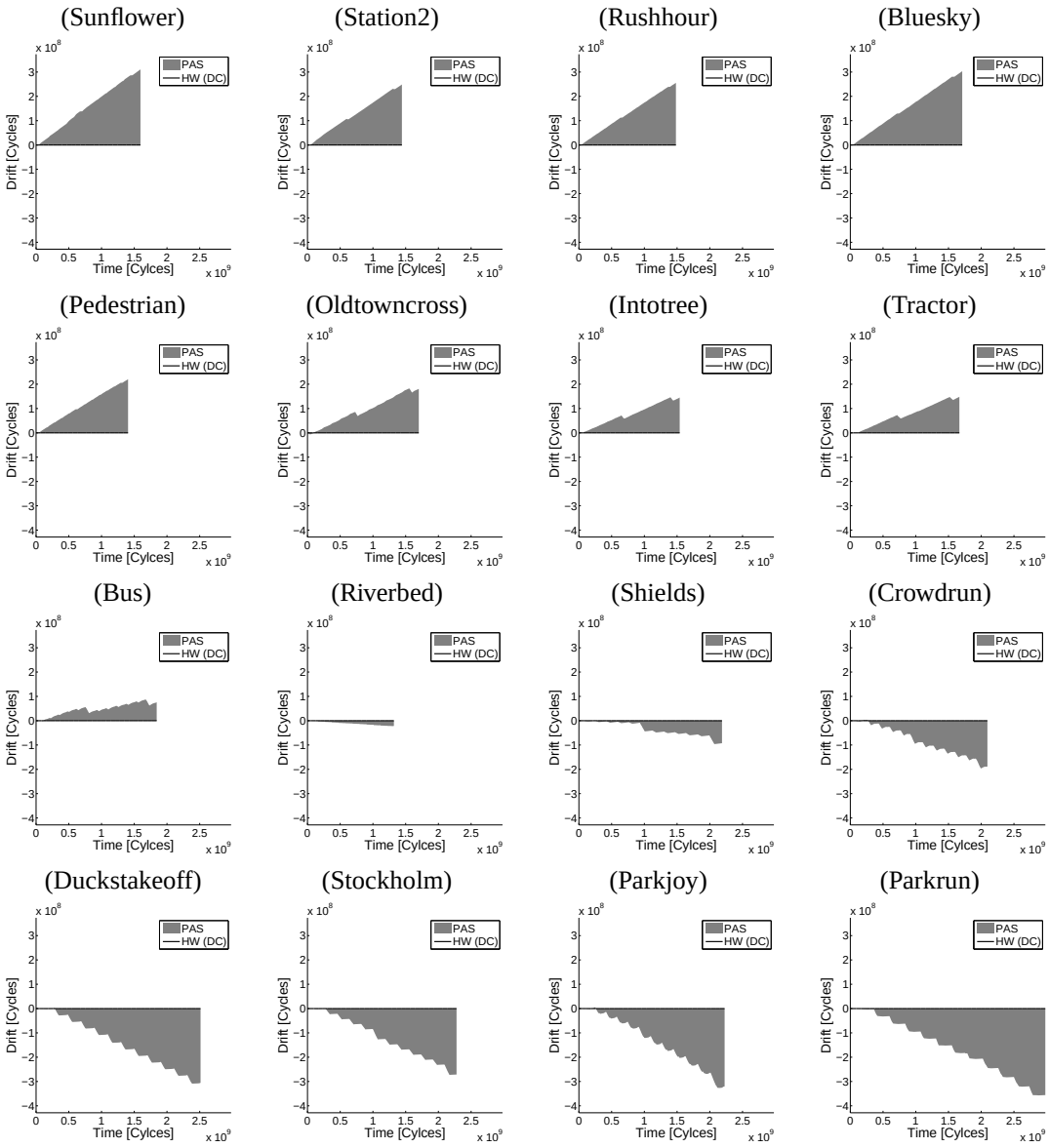


Figure 6.6: Verification of the PAS simulator: The figure shows the absolute runtime difference between the runtime HW(DC) measured on a dual-core system and the runtime estimated by the PAS.

"Sunflower" and "Pedestrian", the PAS tends to overestimate the runtime by up to 18.5%, which means that it still outperforms the simple estimation technique for these streams. It can be concluded that on all test sequences the PAS outperforms the simple estimation technique in terms of accuracy.

Looking at the absolute runtime differences between PAS and hardware-measured runtimes in Figure 6.6, we can observe a strong "stair" effect for high-bitrate sequences such as the

"Crowdrun", the "Dukstakeoff", "Stockholm", "Parkjoy" and "Parkrun" sequences. Between the consecutive frames, strong increases of the cumulative absolute runtime error occur. The pattern indicates that during specific phases of the decoding process the estimation works less effectively which results in a stronger drift during these phases. In this case, it seems that the runtime prediction works less well for B-frames, which coincide with the locations of the increased drifts. This effect between two consecutive frames cannot be observed for the other sequences with lower bitrates, which indicates that the model cannot yet estimate all coding options that are typically used during B-frames when coding high-bitrate video sequences. Furthermore, for sequences with moderate bitrates such as "Tractor" and "Oldtowncross", stronger changes of the absolute runtime error can be observed at frames where a GOP ends and the decoding process that takes place at the end of GOPs could be further refined. This provides room for further increasing the PAS accuracy. However, for this work an average relative error of 7 percent is precise enough to start design space explorations. This is explained in the following section.

6.4 Design space exploration

In this section, exploration of design space is demonstrated by introducing a functional partitioning of our single-core reference H.264 decoder and evaluating the bottlenecks in the resulting system (Section 6.4.1). In Section 6.4.2, this VCS is partitioned further using the multi-column partitioning scheme introduced in Section 3.5.3. For descriptive clarity, we use only four test sequences ("Bus", "Shields", "Stockholm" and "Parkrun") for demonstrating the steps of the design space exploration.

6.4.1 Functional partitioning

Typically, we start exploring a system from a single-core decoding system. This system runs on a single CHILI processor and with a performance of $5 - 10$ frames per second (fps) according to our profilings and the calibrated PAS single-core simulation. We start the exploration of our decoder development by moving the computationally-complex parsing and entropy decoding tasks onto a second CHILI PU. The remaining tasks are executed on the first PU. The two PUs are connected by a buffer that can hold one line of MBs (i.e. 80 MBs for a horizontal resolution of 1280 pixels and 16 pixels width per MB). We use the PAS for estimating the performance of the two-core system. Figure 6.7a shows the frame rates resulting from the individual decoder partitionings. For evaluating the results, the optimal frame rate increase is also visualised. In this context, optimal means that the performance of a system scales linearly with the number of cores (i.e. two processors would result in doubling the frame rate).

The functional partitioning into parser and reconstructor (P+R) significantly reduces the runtime complexity of all 4 sequences. It can be noted that sequences with higher bitrates such as the "Stockholm" and the "Parkrun" sequences benefit more from this functional splitting. Figure 6.7b and 6.7c show the usage for the parsing and the reconstruction cores and the complexity for the parsing and the complete decoding process in cycles per second, respectively. The high usage of 100% of the reconstructor processor compared to the parser for all 4 test sequences shows that the reconstructor represents the bottleneck in our system. Low-bitrate sequences

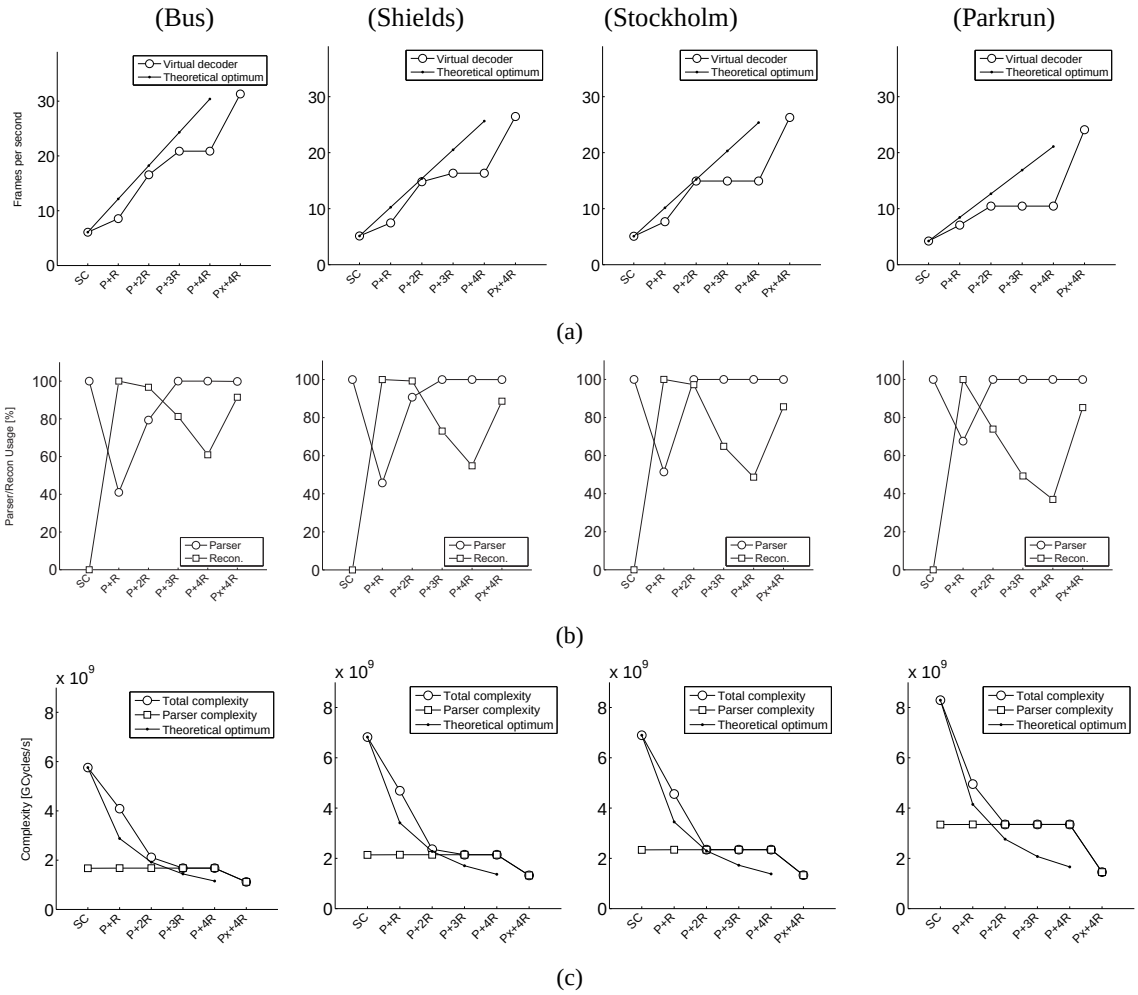


Figure 6.7: Runtime of the simulated decoder partitioning approaches. (a) The frames per second achieved for various approaches are provided: the single-core decoder (SC), the functional split decoder into one parser and one reconstructor core (P+R), one parser and data-parallel partitionings of the reconstructor onto up to four cores (P+2R, P+3R, P+4R) and an extended CHILI processor with improved parsing combined with a data-parallel partitioning of the reconstructor onto four cores (Px+4R). (b) The usage of the cores running the parsing and the reconstruction tasks. The average usage for all available reconstruction cores is shown. (c) The absolute runtime complexity for real-time decoding of the 4 sequences in clock cycles per second.

such as “Bus” and “Shields” only have a moderate parser usage of approximately 40 percent while a stronger parser usage of more than 60 percent can be observed for the high-bitrate sequence “Parkrun”.

The slow reconstructor and the limited buffering between the two cores result in write stalls at the parser side. We have various options for using the parser’s computational resources more efficiently. Firstly, we can improve the reconstructor performance by moving it onto multiple

processing units. Secondly, we can move additional tasks from the reconstructor (e.g., the filter strength calculation from the deblocking) to the parsing core. Thirdly, we could increase the buffer sizes between parser and reconstructor. All three options can easily be evaluated with the PAS. In the next section, we follow the first option, which is the most interesting approach in the context of multi-core architecture design.

6.4.2 Data-parallel partitioning

Functional partitioning of the decoder reveals that the reconstructor is the bottleneck in our current system. By using a data-parallel decoding approach this part of the decoding process can be computed on multiple processing units. In this work, we use the multi-column approach introduced in Section 3.5.3. Each frame is partitioned into vertical regions. Each of the regions is assigned to an individual processing unit. The reconstruction tasks for the MBs in this region are performed on this processing unit. This approach requires consideration of the H.264 macroblock dependencies. Each processing unit can start with the decoding when all dependencies to neighbouring regions have been resolved.

Figure 6.7a shows the increase in frame rate that is achieved with each additional reconstructor core. Adding a second reconstructor (P+2R) strongly increases the frame rates for all sequences according to our simulation results. For the “Bus”, “Shields” and “Stockholm” sequences, the frame rate nearly doubles compared to the scenario with one parser and one reconstructor. Figure 6.7b shows the core usage for the parsing core and the reconstruction cores. For scenarios with more than one reconstructor, the average usage of all reconstruction cores is provided. We can observe that for the “Stockholm” and the “Parkrun” sequences, the parsing core becomes the bottleneck in our decoder. For the “Parkrun” sequences, only around 75 percent of the reconstructors’ execution time is effectively used for decoding. The parsing core runs at nearly 100 percent processor usage.

For three reconstructor cores (P+3R), the performance significantly improves for the “Bus” sequence. The performance for the other sequences does not improve significantly due to the high bitrates and more extensive parsing complexity of these sequences and the resulting stalling times caused by the slow parsing. The average reconstructor usage is between 50 and 80 percent. For 4 reconstruction cores (P+4R), this decreases further and only between 40 and 60 percent of the reconstructors’ execution time is used for decoding tasks.

Figure 6.7c shows the complexity for the parsing and the complete decoding process in cycles per second. The parsing complexity in the final configuration requires approximately 1.25 GCycles per second and determines the runtime of our decoding system. Overall, the PAS simulations indicate that more than 2 cores for the reconstruction do not improve the performance of the system significantly. Only for low-bitrate sequences, a performance increase can be observed. The system designer can choose a system design that considers this already (i.e. cost optimisation) or concentrate on the parsing part of the system that is the obvious bottleneck of the current system. For example, the functionality of the parser can be split onto more cores for improved concurrent processing.

6.4.3 Alternative processor for parsing

For evaluating how a faster parser influences such a system, an extended CHILI core for hardware accelerated parsing was evaluated in the P+4R setup (Px+4R). This extended CHILI processor provides hardware-acceleration for the bistream parsing and the entropy decoding. It should be noted that for simulating heterogeneous architectures with different types of processing units, the PAS uses multiple DDPs as input, and the PAS capability to merge DDPs is exploited. In this case, two DDPs were used for deriving complexity profilings. One DDP was generated from a single-core H.264 decoder running on a normal CHILI and one DDP from the same decoder on an extended CHILI PU. Depending on which processor constellation was simulated, one or the other DDP was used for the PAS simulation.

Overall, we can observe a strong reduction in the parsing time in Figure 6.7c and a significant impact on the frame rates (Figure 6.7a). However, the hardware-accelerated parser increases the frame rate more for sequences with high bitrates such as the “Parkrun” sequence. Figure 6.7b shows that a similar core usage between parser and reconstructor cores and hence a good balance is achieved in this system setup.

6.5 Summary

In this section, we have demonstrated how virtual architectures can be simulated within the PAS. We have proposed a virtual model of a VCS using the PAS’s abstract high-level description language. This model has been calibrated using profilings from existing single core decoder implementations. The calibrated model has been verified using an existing hardware implementation and its accuracy has been determined. On average, a relative prediction error of 11.5 percent could be observed for 16 test sequences.

We have used PAS for exploring new functional and data-parallel decoder partitionings and for predicting the runtime behaviour of these parallel designs. First, we distributed a single-core decoder’s parsing and reconstruction functionalities onto two PUs (i.e. functional partitioning). A PAS simulation of this design idendified the execution of the reconstruction tasks as the bottleneck. By introducing additional PUs for data-parallel processing of the reconstruction tasks, this bottleneck was resolved. We demonstrated that adding up to 3 PUs for the computational intensive reconstruction tasks results in a significant performance increase for this design and a frame rate of up to 17 fps can be achieved. Finally, we replaced the parsing PU in this design with a hardware-accelerated PU that is more suitable for entropy decoding tasks and demonstrated that this new design can achieve real-time performance (i.e. 25 fps) for all test sequences and a high average usage of around 85% for all PUs.

CHAPTER 7

Conclusions and future work

7.1 Conclusions

When making design decisions on a parallel video coding system's architecture (i.e. hardware platform and VCA software partitioning), accurate runtime predictions for VCAs provide an essential means to base these decisions on. This thesis has concentrated on runtime prediction techniques for estimating the performance of parallel VCAs at early stages of the system design. The DDP and PAS methodologies introduced in this thesis combine existing profiling techniques and simulation-based runtime prediction to provide means for efficiently modelling parallel VCSs and for estimating their runtime. The solutions provided in this thesis have tackled two important aspects, namely: (i) analysis of the dynamic behaviour of single-core VCAs in the context of parallel system design and (ii) the runtime prediction of virtual multi-core architectures running parallel VCA implementations.

7.1.1 Analysis of VCA runtime behaviour

We have described the strong structural similarities and conceptually similar coding tools that are shared amongst modern hybrid video coding standards such as H.264 and VC-1. Based on the similar hierarchical coding elements and VCL definitions for representing video content, we have proposed the Data-Driven Profiling (DDP) analysis technique for deriving information from single-core VCA implementations. We have shown that the fundamental similarity between hybrid video coding algorithms can be exploited for mapping of VCA runtime profilings onto the hierarchical data structures and functional blocks of a video coding algorithm. This enables detailed analysis of dynamic runtime aspects of a VCA in context of the processed video data. For example, critical aspects such as variations in the processing time of each coding element and individual functional blocks of the VCA can be investigated.

Knowing the complexity of individual parts of a VCA and being able to analyse complexity in relation to the processed data structures within a VCA provides important insights into dynamic runtime aspects of a VCA on a functional as well as a data level. We have demonstrated

three ways of exploiting DDPs for analysing complexity and deriving essential information for parallel system design. First, we have demonstrated how complexity information about the processed VCL coding elements can already highlight potential problems in work balancing for frame- and slice-based data-parallel approaches in an early design stage. For example, we have extracted information on the dynamic runtime complexity of different MB codings (i.e. I/P/B-predicted). Second, we have shown how complexity variations in the FBs of a VCA's video coding elements can be analysed. This provides a starting point for implementing well-balanced functional partitioning techniques. For demonstrating the above contributions, we have exploited runtime profilings of an H.264 decoder for analysing the dynamic runtime variations in the decoder's functional blocks. We have shown that the runtime as well as the runtime variations for the individual H.264 decoder FBs increase with the bitrate. Decoding blocks with a large amount of conditional code such as the entropy decoding and the deblocking are more sensitive to bitrate changes than pixel-based FBs. Third, we have extracted coding information which determines the progam flow of a VCA and exploited this information to determine the processing time of individual image regions. We have shown how this information supports data-parallel partitioning where the decoding tasks for image and video regions are distributed amongst multiple processors and knowledge about the dynamic behaviour of the VCA provides an intuitive means for choosing the best partitioning.

7.1.2 Modelling and simulation of virtual architectures

We have introduced the PAS simulation technique and demonstrated that this technique can estimate the performance of abitrary system configurations where a VCA is distributed onto multiple processors in an accurate way. It enables the modelling and simulation of virtual video coding architectures without the need for implementing the parallel hardware or VCA software. PAS combines DDP and simulation-based runtime estimation to predict the runtime of a virtual architecture.

We have demonstrated that PAS enables the exploration of complex parallel VCS designs in an early stage of the design process and to quickly adapt existing solutions to new applications and system requirements. Based on this PAS concept, a simulator has been implemented. By analysing the PAS results for a range of complex test sequences, it could be shown that the PAS's runtime prediction on average deviates only by around 11.5% from the real implementation's runtime.

PAS addresses two core requirements of system design exploration, namely high flexibility and low time effort due to modelling and simulation. We have demonstrated the flexibility of our technique to describe complex designs and to explore new designs in a time-efficient way. We have provided examples of functional-partitioned as well as data-parallel H.264 decoding approaches and have shown that no low-level algorithm partitioning is required for the design exploration. The ability to quickly adapt a model to a specific VCA partitioning or a HW architecture has been demonstrated. It has be shown that the PAS can highlight the bottlenecks of a parallel H.264 decoder design before partitioning it onto a multi-core platform. Starting from a single-core H.264 decoder with 5 FPS, we have exploited the PAS methodology for designing a strongly parallel real-time H.264 decoder design that can deliver between 25 and 30 FPS.

We believe that the results of this thesis open up new possibilities to explore parallel system designs in an early design phase and provide novel tools to system designers to optimise the complex development processes of parallel video coding solutions. The contributed techniques can address the design challenges of parallel VCS efficiently and reduce the development time and the risk of design errors significantly.

7.2 Open topics for future research

In this thesis we have demonstrated that the PAS provides accurate means for design space exploration and runtime prediction of virtual and parallel VCS. However, there are various areas and open topics that can be addressed in future research:

- In this thesis we have exploited the structural similarities of video coding algorithms for design space explorations of VCAs. The proposed techniques could be generalised further to be applicable for additional block- or pixel-based image processing algorithms. In this context, algorithm simulation at pixel-level could be investigated.
- Another focus of future research could be on the analysis of existing architectures with high core counts. By using our high-level simulator, we could estimate the complex behaviour of such architectures. We could derive accurate models for the PAS that describe the strong interaction between the individual components in such a system in an efficient way. For example, the impacts of memory bandwidth limitations and hierarchical memory structures on the system's performance could be analysed in detail. This could further refine the PAS methodology and result in a more accurate runtime prediction.
- Furthermore, we could extend the PAS to address the emerging area of Reconfigurable Video Coding (RVC). The techniques and methods provided by RVC could allow the PAS to derive the algorithm structure in an automatic way. On one hand, this can reduce the effort of modelling a VCA in the PAS significantly. On the other hand, this opens up a powerful simulation framework to RVC design applications.

APPENDIX A

Detailed description of test sequences

This chapter provides a detailed description of the individual test sequences used throughout this thesis. The variety of these sequences in terms of content enables us to test the techniques introduced in this thesis over a wide range of content types and show the potential of our methods to analyse and estimate the decoding behaviour for these sequences.

Sequence 1: Bluesky
In this sequence, a rotating camera records two trees from below. A weakly textured, blue sky is visible between the trees. The borders between the sky and the tree provide most of the video's texture while the trees' leaves appear dark and slightly blurred.

Sequence 2: Bus
The sequence contains slow global camera motion. The vehicles move on the street from the right to the left with moderate speed. Partial and full occlusions between the vehicles occur. The sky at the top-left is low textured in contrast to the moderately textured buildings and vehicles.

Sequence 3: Crowdrun
A crowd runs towards the camera and leaves the scene at the bottom and bottom-left sides of the picture. While the upper half of the frame contains little motion, many local movements caused by the individual runners appear in the bottom half of the frame. The sequence's texture concentrates around the centre (e.g. trees) and the bottom half of the frame (e.g. runners).

Sequence 4: Duckstakeoff
In this sequence, multiple swimming ducks cause complex movements of the water surface. As they take off into the air, additional disturbances at the water surface occur. The fast moving wings of the birds are blurred.

Table A.1: Test sequences 1 to 4.

Sequence 5: Intotree
The sequence shows a house next to a natural area with trees. The camera moves above the scenery and takes a slow turn to the right towards the trees. The trees are highly textured and cause most of the scene's texture. The house and the sky are only weakly textured.

Sequence 6: Oldtowncross
A town is recorded from above. While the camera direction stays constant, a moderate camera transition to the left occurs. Most of the texture is caused by the buildings. The low textured sky stays static during the sequence.

Sequence 7: Parkjoy
A natural and highly textured scenery with a high number of occlusions is recorded. The camera moves to the right and keeps track of the moving people at the other side of the river. A tree in the foreground of the scene moves across the picture and occludes parts of the scenery.

Sequence 8: Parkrun
A slowly moving camera moves horizontally to the right and follows a runner at the other side of the river. The scenery is strongly textured due to fine structured trees and the meadow. The top quarter of each frame is only moderately textured.

Table A.2: Test sequences 5 to 8.

Sequence 9: Pedestrian
The sequence shows a pedestrian area recorded with a static camera. Pedestrians and cyclists cross the scenery and a high number of occlusions occur between them. The camera is focusing on the buildings in the background. The moving objects in the front are out of focus and blurred.

Sequence 10: Riverbed
A riverbed and a complex moving water surface with strong reflections is shown. On the surface fine motions due to the wind and the river motion can be seen. The water surface reflects a grey and untextured sky.

Sequence 11: Rushhour
A static camera in the center of the street records two lanes of passing cars. In one lane, the cars move towards the camera. In the other lane, they move away from the camera. The camera focuses on the closer cars and the background is blurred. The hot and moving air causes optical disturbances.

Sequence 12: Shields
A slowly moving camera tracks a man in an indoor scenery. The man moves to the left and shows fine textured shields hanging on the wall. The motion clearly separates the moving man from the shields in the background.

Table A.3: Test sequences 9 to 12.

Sequence 13: Station2
This sequence shows multiple railway tracks and a moving train. The camera zooms out and the scene appears blurred and out of focus.

Sequence 14: Stockholm
The city of Stockholm is recorded from above. The camera slowly moves from the left to the right revealing new buildings and streets. In contrast to the highly textured buildings and streets, the sky is only weakly textured.

Sequence 15: Sunflower
A randomly moving camera records a bee sitting on a sunflower. While the sunflower appears clear and in focus, the fast moving bee is blurred.

Sequence 16: Tractor
The sequence shows a tractor moving through a field. The camera is focused on the tractor and follows its movements. The field behind the tractor appears out of focus and blurred.

Table A.4: Test sequences 13 to 16.

Bibliography

[Agr09] Priya Agrawal. Hybrid simulation framework for virtual prototyping using OVP, SystemC & SCML: A feasibility study. Master's thesis, Indian Institute of Technology Delhi, 2009.

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of the ACM Symposium on Compiler Optimization*, volume 5, number 7, pages 1–19, 1970.

[BDH+06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[BEJ+11] Shuvra S. Bhattacharyya, Johan Eker, Jörn W. Janneck, Christophe Lucarz, Marco Mattavelli, and Mickael Raulet. Overview of the MPEG reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63:251–263, 2011.

[Beu10] Ralf M. Beuschel. *Video compression systems for low-latency applications*. PhD thesis, University of Ulm, Faculty of Engineering and Computer Science, 2010.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the IEEE International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[BJR+07] Roland A. Burger, Giovanni Jacovoni, Cliff Reader, Xiaming Fu, Xiaodong Yang, and Wang Hui. A survey of digital TV standards china. In *Proceedings of the 2nd International Conference on Communications and Networking in China*, pages 687–696, 2007.

[BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[BWX11] Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of VP8, an open source video codec for the web. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, pages 1–6, 2011.

[CAM09] Lucarz Christophe, Ihab Amer, and Marco Mattavelli. Reconfigurable video coding: Objectives and technologies. In *Proceedings of the IEEE International Conference on Image Processing*, pages 749–752, 2009.

[CHC+05] To-Wei Chen, Yu-Wen Huang, Tung-Chien Chen, Yu-Han Chen, Chuan-Yung Tsai, and Liang-Gee Chen. Architecture design of H.264/AVC decoder with hybrid task pipelining for high definition videos. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 2931–2934, 2005.

[CK94] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, 1994.

[CLN+02] Wander O. Cesario, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, Ahmed A.Jerraya, Lovic Gauthier, and Mario Diaz-Nava. Multiprocessor SoC platforms: A component-based design approach. *IEEE Journal of Design and Test of Computers*, 19(6):52–63, 2002.

[CTGG04] Yen-Kuang Chen, Xinmin Tian, Steven Ge, and Milind Girkar. Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, volume 1, pages 63–72, 2004.

[EJ03] Johan Eker and Jorn Janneck. CAL language report: specification of the CAL actor language. Technical Report UCB/ERL M03/48, University of California at Berkeley, 2003.

[EJL+03] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. In *Proceedings of the IEEE*, volume 91, number 1, pages 127–144, January 2003.

[FG01] Jolon Faichney and Ruben Gonzalez. Video coding for mobile handheld conferencing. *Journal of Multimedia Tools and Applications*, 13(2):165–176, 2001.

[FH03] Lance Fortnow and Steve Homer. A short history of computational complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80:95–133, 2003.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[GS04] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[HJKH03] Michael Horowitz, Anthony Joch, Faouzi Kossentini, and Antti Hallapuro. H.264/AVC baseline profile decoder complexity analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):704–716, 2003.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.

[HS09] Heiko Hubert and Benno Stabernack. Profiling-based hardware/software co-exploration for the design of video coding architectures. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(11):1680 –1691, 2009.

[ISO01] *International Standard ISO/IEC 14496-2, Coding of audio-visual objects – Part 2: Visual*. ISO/IEC, 2001.

[ITU88] *ITU-T Recommendation H.261: Video codec for audiovisual services at p x 384 kbit/s*. ITU-T, November 1988.

[ITU93] *ITU-T Recommendation H.120: Codecs for videoconferencing using primary digital group transmission*. ITU-T, March 1993.

[ITU00] *International Standard ISO/IEC 13818-2, Information technology - generic coding of moving pictures and associated audio information: video*. ITU-T, March 2000.

[ITU05] *ITU-T Recommendation H.263, Infrastructure of audiovisual services - coding of moving video: video coding for low bit rate communication*. ITU-T, January 2005.

[ITU12] *ITU-T Recommendation H.264, Advanced video coding for generic audiovisual services (ITU Rec. H.264 | ISO/IEC 14496-10)*. ITU-T and ISO/IEC, January 2012.

[JBH08] Lee Jae-Beom and Kalva Hari. *The VC-1 and H.264 Video Compression Standards for Broadband Video Services*. Springer, 2008.

[Joi13] Joint Model software for H.264/AVC. *http://iphome.hhi.de/suehring/tml/*. 09/05/2013.

[KF05] Hari Kalva and Borko Furht. Complexity estimation of the H.264 coded video bitstreams. *Computer Journal*, 48(5):504–513, 2005.

[KL07] Hari Kalva and Jae-Beom Lee. The VC-1 video coding standard. *IEEE MultiMedia*, 14(4):88–91, 2007.

[KM96] Peter Voigt Knudsen and Jan Madsen. Pace: A dynamic programming algorithm for hardware/software partitioning. In *Proceedings of the International Workshop on Hardware-Software Co-Design*, pages 85–92, 1996.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[Lee10] Edward A. Lee. Disciplined heterogeneous modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering, Languages, and Systems*, pages 273–287, 2010.

[LHH03] Ville Lappalainen, Antti Hallapuro, and Timo D. Hämäläinen. Complexity of optimized H.26L video decoder implementation. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):717–725, 2003.

[LJL+03] Peter List, Anthony Joch, Jani Lainema, Gisle Bjontegaard, and Marta Karczewicz. Adaptive deblocking filter. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):614–619, 2003.

[LLCW10] Li-Juo Lin, Kuei-Chun Liu, Tse-Min Chen, and Wen-Shan Wang. Data partition analyses for video decoders on PAC Duo platform. In *Proceedings of the IEEE Asia Pacific Conference On Circuits and Systems*, pages 568–571, 2010.

[LM95] Yau-Tsun S. Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, 1995.

[MAJ+09] Cor H. Meenderinck, Arnaldo Azevedo, Ben H.H. Juurlink, Mauricio Alvarez Mesa, and Alex Ramirez. Parallel scalability of video decoders. *Journal of Signal Processing Systems*, 2:173–194, 2009.

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[MM08] Tatsuji Moriyoshi and Shigeki Miura. Real-time H.264 encoder with deblocking filter parallelization. In *Proceedings of the IEEE International Conference on Consumer Electronics*, pages 63–64, 2008.

[MML97] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *Proceedings of the 34th ACM/IEEE Design Automation Conference*, pages 147–152, 1997.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Mathematical Studies, Princeton University Press*, 34:129–153, 1956.

[NCK+96] Lode Nachtergaele, Francky Catthoor, Bhanu Kapoor, Stefan Janssens, and Dennis Moolenaar. Low power storage exploration for h.263 video decoder. In *Proceedings of the 9th Workshop on VLSI Signal Processing*, pages 115–124, 1996.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[PK89] Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(2):159–176, 1989.

[RGM06] A. Rodriguez, Alejandro Gonzalez, and Manuel P. Malumbres. Hierarchical parallelization of an H.264/AVC video encoder. In *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering*, pages 363–368, 2006.

[RM05] Massimo Ravasi and Marco Mattavelli. High abstraction level complexity analysis and memory architecture simulations for multimedia algorithms. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):673–684, 2005.

[SBG08] Florian H. Seitner, Michael Bleyer, and Margrit Gelautz. Development of multicore video decoding platforms based on high-level architecture simulations. In *Proceedings of the Junior Scientist Conference*, pages 71–72, 2008.

[SBSG08] Florian H. Seitner, Michael Bleyer, Ralf M. Schreier, and Margrit Gelautz. Evaluation of data-parallel splitting approaches for H.264 decoding. In *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pages 40–49, 2008.

[SDF06] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for design: a guide to using SystemVerilog for hardware design and modeling*. Springer, 2006.

[SE94] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[SFLB07] Klaus Schöffmann, Markus Fauster, Oliver Lampl, and Laszlo Böszörmenyi. An evaluation of parallelization concepts for baseline-profile compliant H.264/AVC decoders. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, pages 782–791, 2007.

[SHH+04] Sridhar Srinivassan, Pohsiang Hsu, Tom Holcomb, Kunal Mukerjee, Shankar L. Regunathan, Bruce Lin, Jie Liang, Ming-Chieh Lee, and Jordi Ribas-Corbera. Windows Media Video 9: overview and applications. *Journal of Signal Processing: Image Communication*, 19(9):851–875, 2004.

[Sip97] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[SMP06] *SMPTE Standard 421M: VC-1 compressed video bitstream format and decoding process*. SMPTE, August 2006.

[SSBG07] Florian H. Seitner, Ralf M. Schreier, Michael Bleyer, and Margrit Gelautz. A macroblock-level analysis on the dynamic behaviour of an H.264 decoder. In *Proceedings of the IEEE International Symposium on Consumer Electronics*, pages 1–5, 2007.

[SSBG08] Florian H. Seitner, Ralf M. Schreier, Michael Bleyer, and Margrit Gelautz. A high-level simulator for the H.264/AVC decoding process in multi-core systems. In *Proceedings of the SPIE Multimedia on Mobile Devices*, volume 6821, pages 5–16, 2008.

[SSBG09] Florian H. Seitner, Ralf M. Schreier, Michael Bleyer, and Margrit Gelautz. Development of a high-level simulation approach and its application to multi-core video decoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(11):1667–1679, 2009.

[SSBG11] Florian H. Seitner, Ralf M. Schreier, Michael Bleyer, and Margrit Gelautz. Evaluation of data-parallel H.264 decoding approaches for strongly resource-restricted architectures. *Journal on Multimedia Tools and Applications*, 53(2):431–457, 2011.

[ST09] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. *Communications of the ACM*, 52(10):76–84, 2009.

[SW05] Gary J. Sullivan and Thomas Wiegand. Video compression – from concepts to the H.264/AVC standard. In *Proceedings of the IEEE*, pages 18–31, 2005.

[SWC07] Shuwei Sun, Dong Wang, and Shuming Chen. A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition. In *Proceedings of the 3rd International Conference on High Performance Computing and Communications*, pages 577–585, 2007.

[SYT04] Tse-Tsung Shih, Chia-Lin Yang, and Yi-Shin Tung. Workload characterization of the H.264/AVC decoder. In *Proceedings of the 5th IEEE Pacific-Rim Conference on Multimedia*, pages 957–966, 2004.

[TM91] Donald E. Thomas and Philip R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, 1991.

[VAG05] Francisco J. Villa, Manuel E. Acacio, and Jose M. García. Evaluating ia-32 web servers through simics: a practical experience. *Journal of System Architecture*, 51(4):251–264, 2005.

[VHD88] *IEEE Standard VHDL language reference manual*. IEEE Press, 1988.

[vJG03] Erik B. van der Tol, Egbert G.T. Jaspers, and Rob H. Gelderblom. Mapping of H.264 decoding on a multiprocessor architecture. In *Proceedings of the SPIE*, volume 5022, pages 707–718, 2003.

[Wal00] Larry Wall. *Programming Perl*. O'Reilly & Associates, 2000.

[WPH+03] Shih-Hao Wang, Wen-hsiao Peng, Yuwen He, Guan-yi Lin, Chen-yi Lin, Shih-chien Chang, Chung-neng Wang, and Tihao Chiang. A platform-based MPEG-4 Advanced Video Coding (AVC) decoder with block-level pipelining. In *Proceedings of the 2003 Joint Conference of the 4th International Conference on Information, Communications and Signal Processing and the 4th Pacific Rim Conference on Multimedia*, volume 1, pages 51–55, 2003.

[WPH+05] Shih-Hao Wang, Wen-Hsiao Peng, Yuwen He, Guan-Yi Lin, Cheng-Yi Lin, Shih-Chien Chang, Chung-Neng Wang, and Tihao Chiang. A software-hardware co-implementation of MPEG-4 Advanced Video Coding (AVC) decoder with block level pipelining. *Journal of VLSI Signal Processing Systems*, 41(1):93–110, 2005.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: fast and flexible machine simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[WSWW06] Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006.

[WWW04] Ferdinand Wagner, Thomas Wagner, and Peter Wolstenholme. Closing the gap between software modelling and code. In *IEEE International Conference on the Engineering of Computer-Based Systems*, pages 52–59, 2004.

[WZ06] Xin-Fu Wang and De-Bin Zhao. Performance comparison of AVS and H.264/AVC video coding standards. *Journal of Computer Science and Technology*, 21:310–314, 2006.

[Xip13] Xiph.org video test media. *http://media.xiph.org/video/derf/*. 29/09/2013.

[YYS+04] Mohamed-Wassim Youssef, Sungjoo Yoo, Arif Sasongko, Yanick Paviot, and Ahmed A. Jerraya. Debugging HW/SW interface for MPSoC: video encoder system design case study. In *Proceedings of the 41st ACM/IEEE Design Automation Conference*, pages 908–913, 2004.

[ZL06] Zhuo Zhao and Ping Liang. A highly efficient parallel algorithm for H.264 video encoder. In *Proceedings of the 31st IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 489–492, May 2006.