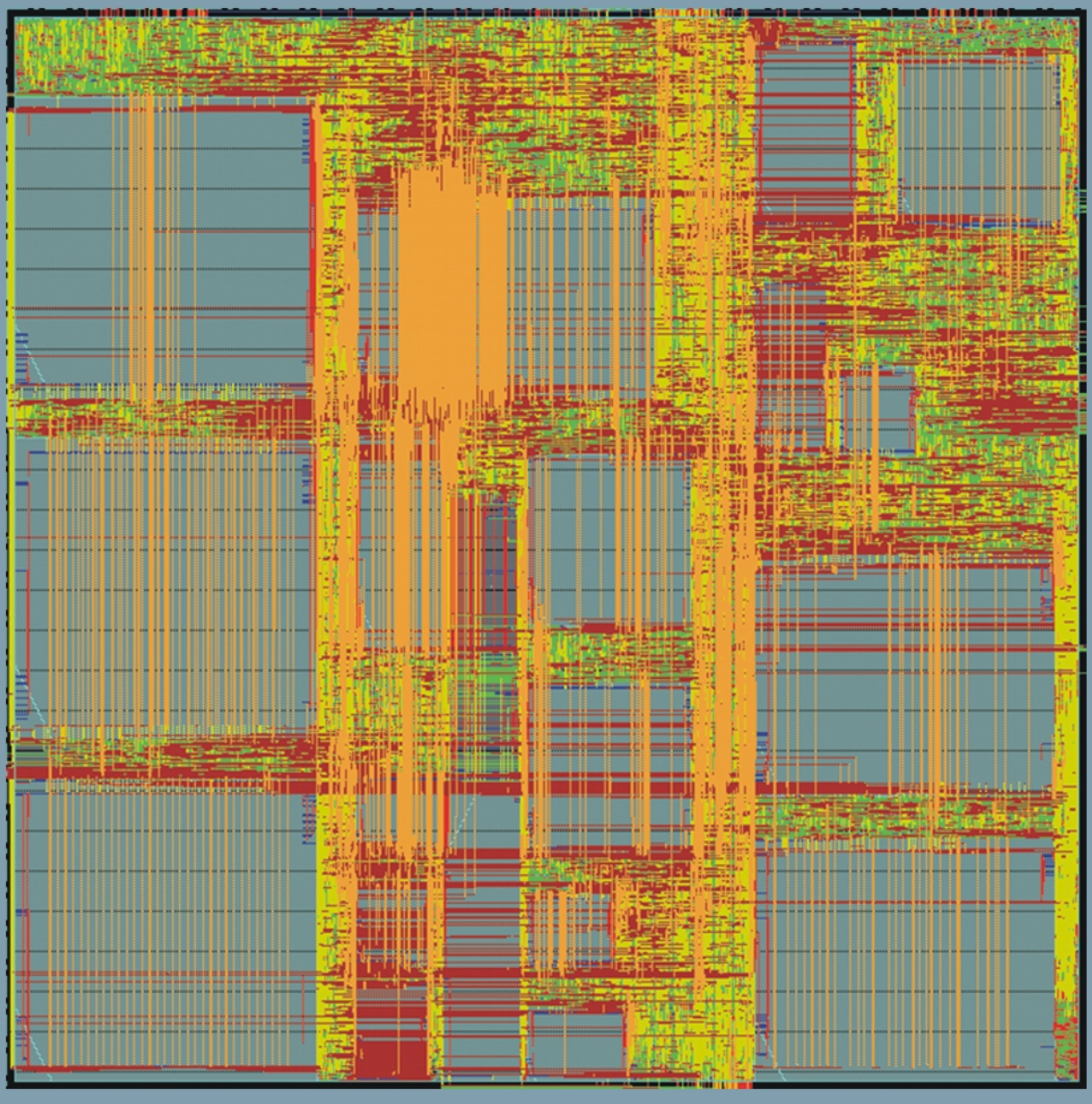
Low Power Design of Block-Based Video Codecs



Kristof Denolf

Low Power Design of Block-Based Video Codecs

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven,
op gezag van de Rector Magnificus,
prof.dr.ir. C.J. van Duijn, voor een commissie
aangewezen door het College voor Promoties in
het openbaar te verdedigen op
donderdag 7 juni 2007 om 16.00 uur

door

Kristof André Jan Denolf

geboren te Kortrijk, België

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. H. Corporaal

Copromotor:
dr.ir. D. Verkest

This work was carried out at IMEC.

Cover design: Kristof Denolf

Printed by: ACCO cvba

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Denolf, Kristof A.J.

Low power design of block-based video codecs / by Kristof André Jan Denolf. - Eindhoven : Technische Universiteit Eindhoven, 2007.
Proefschrift. - ISBN 978-90-386-2033-6
NUR 959
Trefw.: beeldcodering / elektronica ; ontwerpen / real-time computers ; toepassingen /digitale beeldverwerking.
Subject headings: video coding / low-power electronics / data flow graphs / real-time systems.

To Ellen, Noor and Rune

Acknowledgements

All that I can give you
Is forever yours to keep
Wake up everyday with a dream
And happy everafter in your eyes

'Happy Everafter In Your Eyes' – Ben Harper

This PhD thesis is not an individual work, it has grown with the support and care from colleagues, friends and family. Even those who are not mentioned by their name, know they deserve my gratitude. Thank you:

Henk Corporaal, my promotor, and Diederik Verkest, my co-promotor, for your continued guidance and dedication, though the focus of my work shifted significantly.

Marco Bekooij, for introducing me to dataflow models of computation and helping me to add shared memory concepts.

Johan Cockx, for being my first-line aid who always gives valuable feedback.

Christophe De Vleeschouwer, for teaching me the fundamentals of video coding and for being my mentor at the start of this work.

Members of the core and promotion committee: Francky Catthoor, Henk Corporaal, Peter de With, Diederik Verkest and Kees Vissers, and of the promotion committee only: Marco Bekooij and Twan Basten for the useful comments and suggestions on the PhD text.

Jean-Yves Mignolet, for making the layout of the MPEG-4 part 2 encoder, included on the cover of this book.

Jan Bormans and Gauthier Lafruit, for recognizing the PhD candidate in me.

Colleagues joining me in the Xilinx project: Adrian Chirila-Rus, Mark Palusz-

kiewicz, Paul Schumacher, Bob Turney, Bart Vanhoof and Kees Vissers, for facing me with the problems of making a real design down to an FPGA prototype.

Colleagues, at IMEC or working in the same field: Adrian, Bart, Bart, Bert, Carolina, Christophe, Eddy, Guan, Iole, Jiangbo, Klaas, Eric, Erik, Sergio, Sven, Tong, Wilfried, Wolfgang, Yann and Xin, for always making time to set up a discussion that sometimes totally confused me but yet, put my work in perspective and for making it a pleasure to work.

Sarah Goetgeluck, for proof-reading the complete manuscript.

My parents, sister, close friends and family-in-law, for your interest in my work and the joy you bring.

Ellen, Noor and Rune, for giving me a home of love and understanding.

Kristof Denolf
Brugge, April 2007.

Abstract

Low Power Design of Block-Based Video Codecs

The improving display resolution of new video appliances continuously increases the throughput requirements of video codecs and further complicates the challenges encountered during their cost-efficient design. In contrast, the energy and heat dissipation limitations of mobile appliances create the demand for low-power implementations.

This PhD proposes a memory and communication centric design methodology to reach an energy efficient dedicated implementation. The high level steps of this design flow combine memory optimizations and algorithmic tuning on a sequential executable description. Then, a partitioning exploration introduces parallelism using a cyclo-static dataflow model. To maintain the effect of the high-level optimizations, also implementation specific aspects of communication channels, like using a kind of shared buffers, are expressed without extending the model of computation. Consequently, all analysis potential at design time is preserved. Towards dedicated hardware, these channels are implemented as a restricted, but sufficient set of communication primitives. They allow exploiting the principle of separation of communication and computation and lead to an automated RTL test and development strategy enabling rigorous functional testing. In this way, the design time is reduced.

The introduced methodology is applied to the design of a high-performance MPEG-4 video encoder. The fully dedicated video pipeline exploits the inherent functional parallelism of the compression algorithm. It has a tailored memory hierarchy; uses burst accesses to external memory and supports real-time processing of 30 4CIF frames per second. The effect of the high-level optimizations on the power-efficiency is demonstrated through power simulations. The core consumes only 71 mW in a 180 nm, 1.62V UMC technology. This energy efficiency, achieved without using a low-power CMOS library, is equivalent to the state of the art for high resolution video encoders.

Contents

Chapter 1

Introduction

I remember when I lost my mind
There was something so pleasant about that place
Even your emotions had an echo
In so much space

And when you're out there
Without care
Yeah, I was out of touch
But it wasn't because I didn't know enough
I just knew too much

'Crazy' – Gnarls Barkley

In the multimedia communication era, users have the equipment to share rich content like never before. Modern mobile devices support not only voice, but also text, audio, video, graphics, etc. Several technologies drive this rich media exchange. Firstly, its distribution is supported by improving storage and communication technology. Data storage capacity continues to grow with larger (external) hard disks and successors of the DVD. Broadband technology and next wireless generations not only increase the bandwidth of both wired and wireless networks dramatically, also the quality of transmission is enhanced for various types of data. Secondly, compression techniques, essential to reducing the large size of raw multimedia data, achieve improving performances while maintaining a good quality of the audiovisual content. Finally, the development of semiconductor technology still follows Moore's law, doubling the chip capacity and processing power every 18 months. This provides the technology for the realization of more sophisticated compression and communication tech-

niques and makes advanced rich media communication affordable for everyone [26].

Of the different rich media types, digital video typically contains the largest amount of data. Transmitting uncompressed standard definition video (according to the ITU-R Recommendation BT.601-5) requires a bitrate of 216 Mbps [94]. Storing 20 minutes of it requires around 40 GB capacity [17]. Consequently, advanced video compression techniques are needed to represent this visual data in a more concise format. Video compression algorithms are described in a normative way in video coding standards. All of today's International Standard Organization (ISO) MPEG-x and International Telecommunication Union-Telecommunication standardization section (ITU-T) H.26x video coding algorithms are hybrid block-based Motion Compensation/Discrete Cosine Transform (MC/DCT) systems [17, 94, 98]. They are designed for multiple purposes: broadcasting, streaming, surveillance, storage, etc.

The latest video coding standards achieve much higher compression efficiency at the cost of continuously larger algorithmic complexity. At the same time, video resolutions are improving, leading to higher throughput requirements. In total, the tremendous processing requirements to support real-time video worsen the challenges encountered during their cost-efficient implementation. Hence, the development of a video coding system requires a solid design approach to achieve these requirements. Also mobile multimedia devices, such as cellular videophones and digital cameras, follow these two trends. The energy and heat dissipation limitations of such portable devices create the demand for a low-power design of these multimedia applications.

De Man [105] defines digital system design as the translation of a concept (specification) into a set of realizable components with a detailed interaction amongst them to achieve a useful interaction with their environment. This process involves following a path, called the design flow, in an imaginary space called the design space. Traveling this design space involves many trade-offs among different design goals, like performance (often to meet real-time constraints), power, area, time-to-market, etc. The most important conflict occurs between the flexibility and the efficiency. Flexibility is the ability of a system to adapt to different behavior, i.e. it expresses the degree of programmability to support different applications. Efficiency is the skillfulness in avoiding wasted time and effort/resources. In a digital system, efficiency evaluates the obtained performance against the power dissipation, the operation frequency, the area, etc. A more flexible system typically enables reduced design costs, a shorter time-to-market and a lower Non-Recurring Engineering (NRE) cost at the price of lower efficiency in terms of performance and power.

Among all the computing architectures (see Figure 1.1), application-specific integrated circuits (ASICs) are at one end of the spectrum. Their customiza-

tion to a given application makes them the least flexible solution compared to other architectures. This specialization achieves the most energy efficient and best performing solution but also leads to high design and manufacturing NRE costs. At the other end of the spectrum are general-purpose processors (GPPs). They implement only a limited and fixed set of generic arithmetic and control operations that can be used to implement any arbitrary computation. This level of flexibility brings high-level programming language support with all its benefits for the application developer at the expense of efficiency.

Figure 1.1: Different computing architectures in the flexibility-efficiency trade-off.

The energy efficiency of a real-time implementation depends on the energy spent for a task and the time budget required for this task. Horowitz [58] defines the energy delay product to express both aspects and classifies low power optimization techniques according to their impact on this metric. The best techniques improve both terms of the energy delay product. They include problem reformulation to reduce the complexity and the use of specialized hardware. The next set of low power techniques have impact on a single term of the energy delay product. Introducing parallelism improves the performance with (theoretically) no energy cost. Switching off unused parts of the system (while less instantaneous performance is needed) maintains the throughput while the energy is reduced.

At the start of this PhD work (2002), the large complexity of modern video coding standards and the high resolution requirements still required the energy efficiency benefit of dedicated hardware implementations [91] to meet the low power limitation of mobile devices. Nowadays, there is a clear trend towards preserving programmability because of the richness of the media content. For high volumes, ASIC solutions remain interesting while the energy constraints often leads to heterogenous platforms containing hardware acceleration for specialized functionality like multimedia cores [26]. Combining this observation with the effect of different low power techniques (according to the classification of [58] summarized above) and the development cost to apply them, explains the design philosophy of this PhD: (i) first reduce the problem complexity, (ii) then introduce parallelism and (iii) finally add specialized hardware as it achieves the best energy efficiency. Future work will replace the last step with a mapping to (application specific) processors or reuse the last step to develop hardware accelerators. In line with the design philosophy, we focus on three aspects to achieve the low power consumption of a video codec implementation:

1. Data transfer and storage, known as the dominating cost of modern multimedia applications [22].

2. Explore parallelism based on a cyclo-static dataflow model to achieve sufficiently high throughput and to improve energy efficiency.

3. Reduce development cost of power efficient hardware accelerators through an advanced RTL development and verification approach.

In the following, the basic concepts of video compression are first introduced. Section 1.2 discusses design space related aspects and definitions. The scope, main contributions and outline of this PhD manuscript are presented in Section 1.3.

1.1 Video Compression

The booming of digital video products during the recent years, like in video entertainment with DVD, High Definition (HD) TV, Internet video streaming, digital cameras, high end displays (LCD and Plasma), etc., is enabled by compression/decompression (codec) algorithms that make it possible to store and transmit digital video. The main goal of video compression is to reduce the vast amount of data present in the raw format. The resulting encoded video bitstream uses as few bits as possible while maintaining visual quality. The complementary video decoder converts this compressed form back into a representation of the original video data. The encoder/decoder pair is often described as a codec. If the decoded video sequence is identical to the original, then the coding process is lossless. As typically only a compression ratio of around 3 to 4 is achieved by lossless image compression standards, lossy compression is necessary to achieve higher compression factors. Lossy video compression is based on the principle of removing subjective redundancy: elements of the image or video sequence that can be removed without significantly affecting the users perception of visual quality. Video codecs are based on the mathematical principles of information theory. As a consequence of their complexity, building of practical codec implementations requires making delicate trade-offs that approach being an art form [51].

All standards of the MPEG-x or H.26x families divide a frame of a video sequence into MacroBlocks (MB), corresponding to a 16×16 pixel region of the frame. For video material in the 4:2:0 format (the typical raw input format for these video standards), a macroblock contains 6 blocks of 8×8 pixels (Figure 1.2): 4 luminance and 2 chrominance blocks. An MPEG-x or H.26x video codec is characterized by its block-based processing nature.

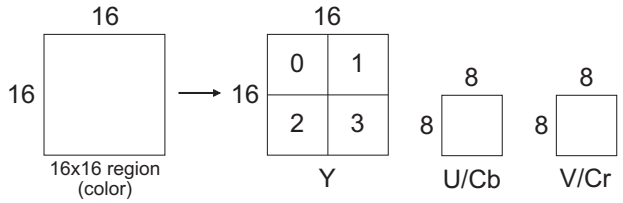


Figure 1.2: 4:2:0 macroblock structure.

1.1.1 A Basic Video Coding Scheme

The main video coding concepts are well described in Chapter 3 of [94], of which this subsection presents a summary. A basic video encoder (Figure 1.3) consists of three main functional units: a temporal model, a spatial model and an entropy encoder. The first processing step, the temporal model, receives the new raw video data as a current frame. To reduce the temporal redundancy, its Motion Compensation (MC) exploits the similarities between the current frame and reference frames (recently coded neighboring, both previous or future, frames) to predict each (macro)block. The Motion Estimation (ME) is the part of the temporal model that searches the closest match in the reference frames for each MB in the current frame. This ME is known as the most performance intensive function of video compression. The output of the temporal model is a residual/error after MC (created by subtracting the prediction from the actual current frame) and a set of model parameters, typically a set of motion vectors describing the relative location of the best match in the reference frames.

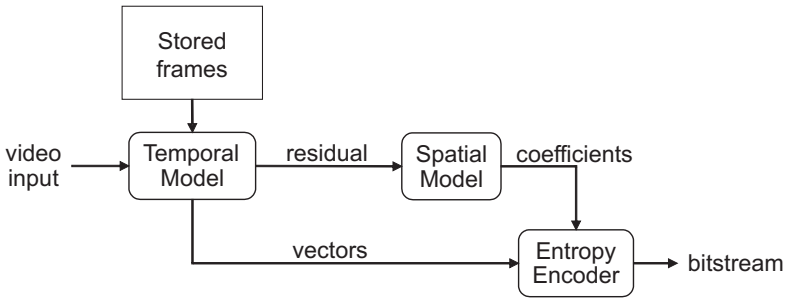


Figure 1.3: Video encoder block diagram [94].

The residual, subdivided into 8×8 or smaller blocks, forms the input to the spatial model which makes use of similarities between neighboring samples in this residual frame to reduce spatial redundancy. The transform part of it, such as the Discrete Cosine Transform (DCT) or a derivative, decorrelates the spatial samples by representing them into the frequency domain as transform

coefficients. The quantization part reduces these coefficient to perceptually important values only. The output of the spatial model is this set of quantized transform coefficients.

The motion vectors (from the temporal model) and the quantized coefficients (from the spatial model) are further compressed by the entropy coder. This step exploits the statistical nature of these vectors or coefficients to produce the final compact video bitstream ready for storage or transmission. Such a compressed sequence also contains header information containing the characteristics of the sequence or parameters of the frame.

The process of redundancy removal is reversed in the video decoder that, while parsing the compressed bitstream, recovers the coefficients and motion vectors using an entropy decoder. The prediction of motion compensation is combined with the decoded residual to reconstruct the original video.

A video codec has the choice between 3 modes to compress an individual frame: I, P or B. Intra frames (I) are encoded independently: no reference to any other frame is made (i.e. no MC). Inter coded frames exploit the temporal correlation. They can be coded as Predicted (P) frames using MC with only previous frames as reference (forward prediction) or as bi-directional predicted frames (B) frames from both past frames as well as frames slated to appear after the current frame.

1.1.2 Video Coding Standards

Standardization work in the video coding domain (Table 1.1) started around 1990. The ITU-T H.26x family has grown over 3 generations: H.261, H.263 and H.264. H.261 was targeted for two-way video conferencing applications over ISDN networks. The H.263 standard was released for transmission of video-telephone signals at low bit rates.

The ISO MPEG-x family, MPEG-1 and MPEG-2, are currently the most widely introduced video compression standards. MPEG-1's initial driving application was storage and retrieval of moving pictures and audio on digital media such as video CDs, but is widely accepted for distributing video over the Internet. MPEG-2 was developed for the compression of digital TV signals and supports both progressive and interlaced video. MPEG-4 increased error robustness to support wireless networks, included better support for low bitrate applications and provided additional object-based functionalities. MPEG-4 has found application in Internet streaming, wireless video, and digital consumer video cameras as well as in mobile phones and mobile palm computers.

One of the most important developments in video coding in the last years has

been the joint definition of the H.264/AVC standard by both ITU-T and the ISO/IEC. This most recent video standard, with official name MPEG-4 part 10 and ITU-T H.264, covers all application domains of the previous ones, while providing superior compression efficiency and additional functionalities over a broad range of bit rates and applications. More specifically, H.264/AVC employs: a more precise motion compensation prediction, variable size MC blocks, multiple frames of the past, context-based entropy coding, an in-loop deblocking filter and advanced rate-distortion optimization [84]. The joint effort currently continues in the development of Amendment 3 for MPEG-4 part 10 adding scalability features to AVC. This new video standard, better known as SVC, is expected to be fixed around April 2007.

Table 1.1: Basic features of international MPEG and ITU-T video coding standards.

Feature	H.261 (1990)	MPEG-1 (1993)	MPEG-2 (1995)	H.263 (1995)	MPEG-4 part 2 (2000)	H.264/AVC (2003)
Frame type	I,P	I,P,B	I,P,B	I,P,B	I,P,B	I,P,B
MC block size	16×16	16×16	16×16, 16×8	16×16, 8×8	16×16, 8×8	16×16, 16×8, 8×16, 8×8, 8×4, 4×8, 4×4
MC accuracy	1 pel	$\frac{1}{2}$ pel	$\frac{1}{2}$ pel	$\frac{1}{2}$ pel	$\frac{1}{2}$ pel	$\frac{1}{4}$ pel
Transform	8×8 DCT	8×8 DCT	8×8 DCT	8×8 DCT	8×8 DCT	8×8, 4×4 Integer DCT
Entropy coding	VLC	VLC	VLC	VLC, SAC	VLC	UVLC, CAVLC, CABAC
Deblocking filter	In-loop	None	Post	In-loop (Annex J)	Post	In-loop
Prediction Mode	Frame	Frame	Frame, field	Frame	Frame, filed	Frame, field
Resolutions	176×144, 352×288	352×240, 352×288	all up to HDTV	128×96, 352×288	all up to HDTV and above	all up to HDTV

The combination of temporal block-based motion compensated prediction and block-based DCT coding provides the key elements of the MPEG and ITU-T video coding standards described above. For this reason, the MPEG and ITU-T coding algorithms are usually referred to as hybrid block-based MC/DCT algorithms [17, 26, 94]. Even the standards under development (like the scal-

able and multi-view video codec) build on top of this basic scheme. Every more recent generation adds new or refined video tools to improve the compression performance or to extend the supported functionality. Needless to say, most of the advanced techniques increase the complexity at both the encoder and decoder side. Consequently, with increasing video resolutions, a real-time implementation of these algorithms requires a solid design flow to achieve a cost-efficient solution.

1.2 Design Space

Abstraction is the key to master the complexity of digital system design [105]. By raising the level of abstraction, the number of objects in the design decreases and this allows the designer and tools to focus on the critical aspects and explore a larger region of the design space without being overwhelmed by unnecessary details [48]. A design flow is a methodological approach consisting of a set of design steps in a specific ordering that guides the designer to traverse several abstraction levels down to the final implementation [29].

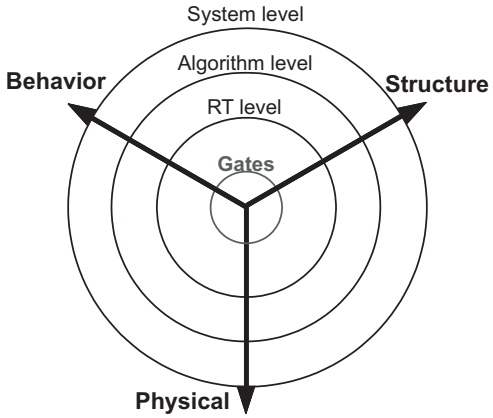


Figure 1.4: The Y-chart.

The Y-chart [47] is a general representation of the design space capturing three basic views/models of the system: behavior, structure and physical implementation (Figure 1.4). The behavior model consists of a set of abstract functional entities with their control and data dependencies to describe the desired functionality. The structural model contains a set of components and their connectivity. The physical implementation is the layout of the system: a blueprint or a mask containing the dimensions and position of all components (placement) and their interconnections (routing).

The three axes of the Y-chart correspond with these views. Each axis spans

four levels characterized by the type of its objects and measuring the degree of abstraction: system-level, algorithm, Register Transfer (RT) and gates. The center represents a fully detailed description of the system. Moving away from the center, corresponds to a higher level of abstraction where the objects are composed of lower level ones. Typically, a higher abstraction level offers a larger amount of opportunities [62, 75, 58](e.g. optimization freedom, design choices, etc) and has a lower modeling cost.

A design flow is a path traversing, in a step-wise manner, over different points in the Y-chart to gradually refine the system specification. Each point uses models of which the accuracy and amount of implementation detail depends on its position in the Y-chart. These aspect models help the designer to estimate the consequences of successive design decisions for certain aspects of the product, without really working out the design. In this way, a bigger search space can be covered [29]. Two types of models are used in this work: a Model of Computation (MoC) and a Programming Model (PM). A basic MoC is a mathematical formalism to reason about the parallelism of an application specification. In this text, the term model of computation is used for a more extended version where the concept of time is added [18] with annotations. A PM is a conceptualization of the final architecture in which the application is written.

Model of Computation is a mathematical formalism that describes the interaction between components in a system. Models of computation provide useful high-level abstractions for concepts like time and concurrency. It allows the analytical derivation of properties like liveness, throughput, latency or the use of resources like time-slices, buffer capacities and bandwidths.

Programming Model is the conceptualization of the machine that the programmer uses while coding applications. Each programming model specifies how parts of the program running in parallel communicate information to each other and what synchronization operations are available to coordinate their activities. Applications are written in a programming model [30]. Typically a compiler is available to translate a program, written in this programming model, into machine code.

The separation between function (behavior) and architecture (structure) inherent to the Y-chart is one of the important aspects of the orthogonalization of concerns methodology [65]. Another pillar is the separation of computation and communication. This allows the design process to consist of two analogous parts [48]: (i) mapping the computational parts on processors or designing dedicated accelerators for them and (ii) mapping the communication onto the system busses or network or designing dedicated communication

channels. Additionally, this separation allows the mapping or design of separate computational or communication components of the system.

Given the typically data-dominated nature of multimedia applications (and video applications in particular), special care needs to be taken of its data transfer and storage [22] as this is the main cost factor [46, 68, 108].

Consequently, the design flow proposed in Chapter 2 has a memory and communication focus. It starts from (C code) behavioral description at system level (upper left of Figure 1.4) and ends with a structure at RT level. From there on, synthesis and place and route tools complete the path to the final implementation on FPGA or ASIC. At every design step, the appropriate MoC and PM is selected.

1.3 About this Thesis

The real-time and low power implementation of multimedia applications spans a very broad range of design techniques, platform choices, optimization opportunities, etc. By setting the focus of this PhD work in the next subsection, a first selection of architecture characteristics is fixed and the size of the exploration space is reduced. Within this scope, the main contributions are highlighted and the structure of this manuscript is introduced.

1.3.1 Scope

The proposed design approach concentrates on block-based image and video processing, more specifically on the hybrid block-based motion compensation/DCT video compression systems. All widely accepted video coding standard of ISO MPEG and ITU-T, but also Microsoft's Windows Media Video 9 and the Chinese Audio-Video Standard exploit this basic compression scheme. Mainly its block-based processing aspect is of importance as it allows introducing data locality and an efficient use of dedicated communication channels.

To achieve the high computational power with low energy consumption for real-time video compression, the parallelism of a multiple processor implementation is needed. The partitioning step focuses on functional parallelism at the coarse-grain level: each concurrent task implements a different subset of the statements (i.e. a different component) of the application. Inter-iteration dependencies do not hinder the use of functional parallelism, but naturally lead to a pipelined execution. Consequently, it is often called pipelining. By using functional parallelism, heterogeneous, task-specific processors can be exploited. To distinguish the exploited coarse-grain functional parallelism from

fine-grained forms such as instruction level parallelism, software pipelining or data level parallelism, it is called task-level pipelining in this text.

The low energy requirements resulting from the limited heat dissipation and battery lifetime of mobile devices drive the need for a dedicated implementation. This involves custom hardware processor design for the different components of the video pipeline. Each task-specific processor needs only to implement the functionality of a single task, thus leading to a smaller, more efficient processor that is easier to design and synthesize.

1.3.2 Summary of the Contributions

The main contributions of the proposed design methodology for the low power implementation of block-based video codes have a different angle. Contributions 1,2,4 are more generic/academic. Contributions 3,6,7 are applied. Contribution 5 contains both aspects.

1. A design flow covering the abstraction levels from sequential (C) specification to RT-level [42, 38] tailored to block-based video processing. To deal with the data dominance, the established DTSE methodology is exploited and extended at the high-level with algorithmic tuning [41] and completed at the lower levels with a partitioning exploration matched towards RTL development. See Chapter 2.

2. The use of a Cyclo-Static DataFlow (CSDF) model of computation in such a way that also implementation oriented characteristics, like the use of shared memory concepts in the communication channels, are taken into account. This special interpretation of the CSDF model avoids extensions to the model of computation and consequently keeps its analysis potential at design time [37]. This allows targeting a system with a self-timed execution and the calculation of the buffers capacities for a given schedule using the model. See Chapter 3.

3. Driving the development of the SPRINT tool [28] through the definition and use of the communication primitives (see contribution 4) in a real design and by integrating a link to the RTL development and verification environment. See Chapter 4.

4. A limited but sufficient set of Communication Primitives (CPs) bridging the gap between the use of a CSDF model of computation and its implementation on a FPGA or an ASIC. The clear definition of this set of CPs allows exploiting the principle of separation of computation and communication. This does not only enable the separate and upfront implementation of the CPs but also the isolated (RTL) development and

verifications of the different components (parallel tasks) of the video compression system. See Section 3.5.

5. A verification strategy and RTL development approach with an automated environment [27, 38] to reduce the design time. The verification first uses a parallel timed SystemC model to check the introduced concurrency and correctness of the buffer sizes calculated based on the CSDF model. The RTL development framework automatically inserts communication channels and supports simulation and fast prototyping (on FPGA) verification environments to enable rigorous testing. Seamless switching between both environments on a frame basis combines the speed of fast-prototyping with the signal visibility of simulation. A significant design time reduction (over a factor 2) is measured on the MPEG-4 design to which the RTL development and verification approach (described above) was strictly applied on the encoder but not on the decoder. See Chapter 4.

6. A proof-of-concept by applying the proposed design flow on the development of an MPEG-4 part 2 Simple Profile video codec. The encoder supports up to 4CIF (704×576) at 30 frame per second and is mapped on FPGA [39] and synthesized on an ASIC, 180 μm 1.62 V UMC technology [40] to verfiy its low power dissipation. The decoder is an FPGA implementation and sustains up to XSGA (1280×1024) at 30 frames per second [97]. Both encoder and decoder provide multi-stream support. Real-time operation is guaranteed as long as the sum of the streams does not exceed the maximum throughput (mentioned above). See Chapter 5.

7. A texture coding module at the encoder using heuristics to predict blocks that after quantization will consist of only zeros or with only a first row or a first column with non-zero values. This technique reduces the complexity of this texture coding [41]. At the decoder side, texture decoding is simplified by combining motion vector information with the coded block status of a block. In this way, accesses to the reconstructed frame memory are reduced [36] with 20 % for a video sequence of average complexity (e.g. Foreman). See Chapter 6.

In addition to these contributions, every chapter presents an overview of related work and if appropriate a comparison.

1.3.3 Outline

The next chapter introduces the proposed design flow for low-power block-based video processing implementations. The design flow takes special care of

the data transfer and storage cost, as this is known to be the dominant cost of multimedia applications. For each of the 6 steps a detailed description of the design activity and the applicable tools is given. This design flow chapter is the center point of the complete text. Subsequent chapters focus on a single aspect or step of the methodology or apply it to a real-life example.

Chapter 3 studies the use of a model of computation to support the parallelization step of the design flow. It interprets a CSDF model in a special way to include implementation specific details, like the use of shared memory alike communication channels. As no extensions are required for this, the full analysis capabilities at design time are maintained.

To support the HDL translation step, Chapter 4 proposes a verification strategy and RTL development approach. The combination of a parallel SystemC transaction level model with a dual verification environment at RT level shortens the design time.

Chapter 5 applies the design flow to the implementation of an MPEG-4 part 2 Simple Profile video codec. It uses the CSDF modeling to support the buffer capacity calculation in the parallelization step and benefits from the RTL development and verification approach during the hardware design. The resulting low power implementation is a video pipeline exploiting the inherent functional parallelism of the compression algorithm.

One of the algorithmic optimizations applied during the MPEG-4 part 2 Simple Profile video codec design, the intelligent block processing, is described in more detail in Chapter 6. It simplifies the texture processing complexity by predicting blocks full of zero values and combines this coding status with the motion vectors to also avoid motion compensation and storing operations.

Finally, the conclusion and discussion of future work complete this manuscript.

Chapter 2

Design Flow

Half of learning how to play
Is learning what not to play

'Up up up up up up' – Ani Difranco

The high complexity of present-day multimedia codecs and wireless communications stresses the need for a systematic approach to reach a cost efficient implementation. Their design flow covers different abstract layers to gradually evolve towards the final realization. Direct translation from C to RTL-level is error-prone and lacks a modular verification environment. Defining the system first at higher abstraction level permits high-level optimizations, typically yielding larger gains. In addition, this approach shortens the design time: it favors design reuse, allows structured verification and fast prototyping, and enables the usage of synthesis techniques [45].

The main cost factor in this work is energy efficiency. Horowitz [58] defines the energy delay product to measure this energy efficiency, expressing both the consumed energy and the achieved performance. Combining the impact on this metric of low power optimization techniques with the development effort to apply them, explains the followed design philosophy (see Chapter 1). This is reflected in the major phases of the proposed design flow in Figure 2.1: (i) a sequential phase for problem simplification and (ii) a parallel phase first introducing concurrency and then focusing on HW development. More specifically, the design flow starts from a system specification (typically provided by an algorithm group or standardization body like MPEG) that is gradually refined into the final implementation: a netlist with a (set of) executables.

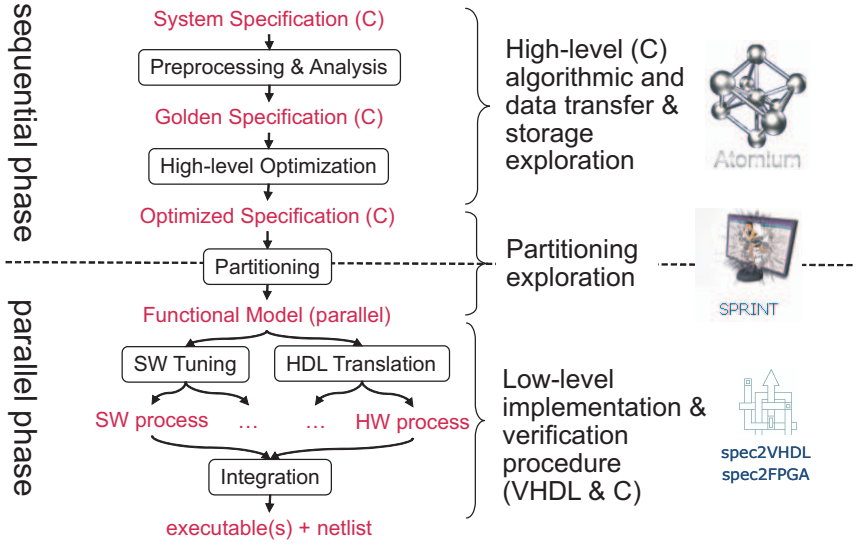


Figure 2.1: Different design stages.

The sequential phase preprocesses the initial specification, analyzes it and applies high-level optimizations. The parallel phase divides the application into parallel processes and maps them on a processor or translates them to RTL. Each design step is discussed in more detail in Sections 2.1 and 2.2.

The first phase uses a higher level of abstraction where the system is represented using a behaviorial model, described as a sequential (C) program, and aims at reducing the memory and computational complexity. During the second phase, concurrency is introduced. Using different Models of Computation (MoC) helps the designer to reason about the parallelism in the system while a Programming Model (PM) provides the means to describe it. A limited but sufficient set of (formally defined) communication primitives (Section 3.5) realizes this PM. The CPs are implemented as dedicated channels on the final HW core. Consequently, the PM directly reflects the final architecture of the system.

The direct relation between PM and architecture is in line with the custom HW focus of this work. A more flexible implementation, on a multiprocessor system for instance, requires also a platform selection path to define an appropriate and optimized architecture for the chosen application class [74]. Such future research could extend the proposed design flow to match the known Y-chart of platform based design.

The data transfer and storage has a dominant impact on the realization efficiency of multimedia applications [22] as the access speed, size and power consumption of the available storage components typically form a severe bot-

tleneck in the system, especially in an embedded context [46, 68, 108]. Additionally, the performance gap between datapath and memory is growing every year [57]. Both phases of the proposed design flow address this memory related cost factor: the high-level optimizations reduce data transfer rates and introduce data locality and in this way, also prepare for the parallelization and for an efficient use of the communication primitives in the second phase.

All quantitative principles of computer design of Hennesy and Patterson [57] are present in the proposed design flow. The 'principle of locality' directly relates to its memory and communication focus (as explained in the previous paragraph). 'Take advantage of Parallelism' is done at coarser level during the partitioning step and at the level of an individual processor in the SW tuning or HW translation steps. Finally, 'make the common case fast' is a principle valid for the complete design flow. It relates to reducing both the computational and memory complexity, i.e. problem simplification, of the bottleneck blocks of the application in the high-level optimization step, to achieving a balanced split in the partitioning step and to processor specifically optimizing the important kernels (either through SW tuning or special HW translation).

The gradual refinement of the system specification as executable behavioral models, described in a well defined PM, yields a reference throughout the design. Combined with a testbench, this behavioral model enables profound verification in all steps. Additionally exploiting the principle of separation of communication and computation in the parallel phase, allows structure verification through a combination of simulation and fast prototyping (Chapter 4).

Chapter 5 demonstrates the proposed design flow on the development of a fully dedicated MPEG-4 part 2 Simple Profile video codec. The following section describe both phases with their tool support in detail.

2.1 Sequential Design Phase: Steps and Tools

The goal of the sequential phase is to transform the initial system specification (reference code coming from a standardization body or from another source), into a functional specification consisting of different functional modules with localized data processing and data communication. The optimizations applied in this first design phase are performed on the sequential program (often C code) at the higher design-level, offering the best opportunity for the largest complexity reductions [58, 62, 75]. Two design steps are present: (i) preprocessing and analysis and (ii) high-level optimizations. Both steps are detailed in the next subsections.

The ATOMIUM tool framework [1] is used intensively in this phase to validate

and guide the decisions. Through instrumentation of an ANSI C program, this tool suite provides functionality for advanced data transfer and storage analysis, code pruning and optimization [20]. These measures, of both platform independent and platform dependent nature, are combined with profiling analysis gained from simulations on a generic computer platform [67] to add an indication of the computational complexity. The combination of instrumentation and profiling allows checking the data-dominance of a function: when the cycle consumption share of a function surpasses its memory access share, the function is computation-dominated.

The effect of the high-level optimizations of the sequential design phase is continuously evaluated using these tools.

2.1.1 Preprocessing and Analysis

The availability of an executable specification at the start of the design process avoids the difficult task to implement a system from scratch. However, such reference software specification often contains functionality supporting different application profiles. This results in an oversized C code distributed over many files.

The preprocessing step restricts this reference code to the required functionality given a particular application profile and restructures it in a golden specification prepared for an initial complexity analysis. ATOMIUM/Pruning automates the error-prone and tedious step of stripping unused functionality. It removes functions and their calls based on the instrumentation data of a testbench representative for the desired functionality. This implies careful selection of the set of input stimuli, which has to exercise all the required functionality. Consequently, during this first step, the testbench triggering all required video tools, resolutions, framerates, etc. is fixed for the rest of the design.

An appropriate complexity analysis is an important and reoccurring part in the implementation of complex multimedia algorithms. It guides the high-level optimization process and can assist the definition of a new multimedia processing architecture [93]. The temporal platform used to compile and run the golden specification (the result of the preprocessing) as basis for such evaluation, typically differs from the target implementation architecture. Consequently, platform independent complexity metrics, like the number of basic operations, providing accurate information for the implementation are required, covering both the computational and the data-transfer aspects [59, 93].

The platform independent memory measures provided by ATOMIUM/Analysis and the profiling numbers obtained on a generic computer platform are collected for some typical cases of the selected testbench. Their distribution over

the main functional modules of the algorithm is complemented with the division of the memory accesses over 4 categories of array sizes, reflecting the inherent data reuse chain present in video coding, i.e. frame - set of macroblocks - macroblock - block - subblock - set of samples:

1. FrameSet: storing a single frame or multiple frames
2. Slice: holding more than a single block (8×8 pixels) and less than a frame
3. Block: containing between 9 and 64 values (i.e. able to store a single block)
4. SampleSet: for small arrays of maximally 8 values

Note that at this stage, the bitwidth of the data in the arrays is not yet taken into account. The data locality of the application can be assessed by evaluating this distribution. An application has a poor data locality if most accesses are made to frames or slices.

The combination of complexity analysis tools also helps evaluating the contribution of the different video coding tools to the total compression performance in relation to their complexity share [96]. Such co-evaluation is essential input for the algorithmic tuning of the next design step (Subsection 2.1.2) that, next to the simplification of the algorithms of the video tools, also selects a relevant subset of them, based on their complexity compression trade off.

The outcomes of this design step are a golden specification, a testbench definition, and an initial complexity evaluation pointing out the first candidates for optimization.

2.1.2 High-Level Optimizations

To cope with the dominant memory cost, the Data Transfer and Storage Exploration (DTSE) methodology [22] has been developed. It was initially oriented towards systems with a customizable memory organization, and later extended for embedded cores and general purpose architectures. DTSE focuses on real-time multi-dimensional signal processing applications characterized by: their real-time constraints, a control flow with many nested loops, many multi-dimensional arrays and long, complex pieces of code. This methodology has been successfully demonstrated on different application areas. Image and video coding examples include: application specific implementations where the designer has full control [77, 107], embedded systems with an application specific

memory architecture [79], and software implementations relying on the conventional memory architecture (general purpose cache) where improved cache hit ratios result in a speed up [78].

The 7 steps in the atomium script [20] of the DTSE methodology gradually include more platform characteristics and hence evolve from platform independent to platform dependent optimizations:

1. Preprocessing and pruning: isolates the (DTSE) relevant functionality to allow a global exploration of the important optimization alternatives, irrespective of the initial specification coding style.

2. Global dataflow transformations: reduce redundant transfers and energy/speed bottlenecks leading to a different internal data-flow code organization without changing the functional behavior.

3. Global loop transformations: introduce access locality and regularity globally across the entire program by transforming the control-flow and loop structure of the application code.

4. Data reuse optimization: adds reuse copy candidates to explore the memory hierarchy, selects an appropriate data reuse tree and assigns this memory hierarchy to different layers.

5. Storage/bandwidth optimization: exploits parallelism in the data transfers to meet real-time constraints while optimizing cost (e.g. power).

6. Memory allocation and assignment: derives a memory organization avoiding multi-port memory use by generating a memory partitioning.

7. Memory inplace optimization: decides an optimal layout for the (multi-dimensional) arrays within each memory for reduced memory footprint and for cache memories to minimize miss rate.

The first step of this atomium script is taken up in the preprocessing and analysis (Subsection 2.1.1) of the proposed design flow (see Figure 2.3). The currently discussed high-level optimization design step is detailed in Figure 2.2. It selects and adapts a major portion of the remaining part of the atomium script to connect to the parallel phase deriving the dedicated (video) architecture. It also adds algorithmic optimizations, trading a limited amount of the algorithmic performance (such as compression efficiency) for a large computational and memory complexity reduction, in such a way that the main DTSE principles are respected: (1) reducing the required amount of processing, (2) introducing data locality and enabling data reuse, (3) minimizing the data transfers (especially to large memories) and (4) limiting the memory footprint. As a result, the algorithmic tuning is complementary to and an

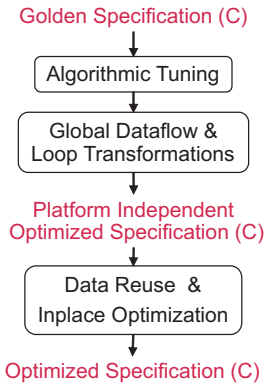


Figure 2.2: Different steps of the high-level optimizations.

enabling factor for the memory optimizations (DTSE). An example of an algorithmic optimization breaking a dependency hindering loop transformation is the modification of the rate control in the MPEG-4 video encoder design of Chapter 5. This extension of DTSE with algorithmic tuning is a first part of contribution 1.

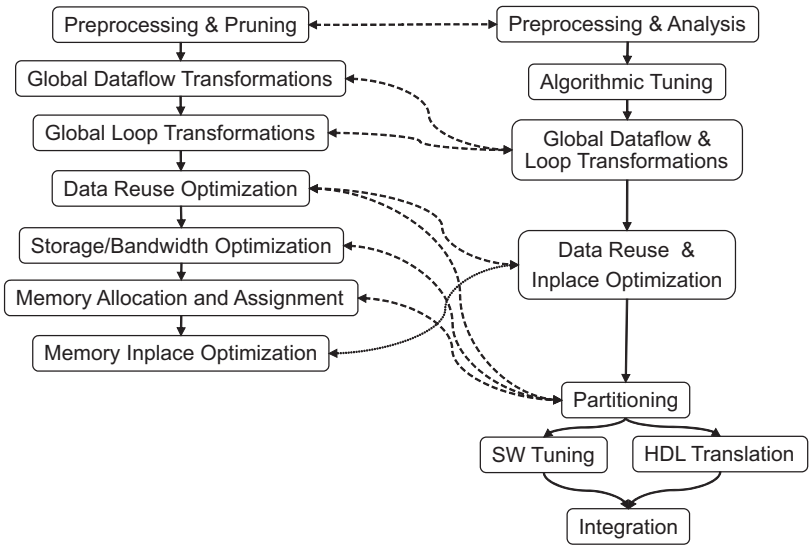


Figure 2.3: Relation between the atomium script on the proposed design flow.

The relation between the proposed design flow and the atomium script is illustrated in Figure 2.3. The platform independent steps of the atomium script (global dataflow and loop transformations) are applied unaltered during the high-level optimizations. They are used in a broad sense so that not only the memory complexity is tackled, but where possible, also the com-

putational complexity is reduced. From its more platform dependent part, only the data reuse exploration and inplace optimizations are retained. Real-time constraints are not yet taken up as the application first needs to be partitioned in parallel processes to achieve the high throughput requirements. Storage/bandwidth optimizations, memory hierarchy layer assignment, and memory allocation and assignment are addressed during this partitioning and during the SW tuning or RTL translation. The identification of the type and the number of dataflow edges (i.e. communication channels) between processes, and the calculation of the buffer sizes of these edges is made during partitioning. Such a dataflow edge is then allocated to a (HW) communication primitive. A further refinement is possible during the mapping when the schedule of each process gets fixed. This coupling of DTSE to a partitioning and mapping phase is the second part of contribution 1. In addition, data layout optimizations are applied to enable burst transfers between frame-sized arrays and the reuse copies. An example of such data layout adaptation is the tiling of a raster scan frame memory in a block-based organization (see also Chapter 5).

All these high-level optimizations produce a (memory) optimized specification with localized data processing and a tailored memory hierarchy. They reduce the size of the data elements of the communication channels and the amount of data exchange over these channels. In this way, they prepare for a throughput and power efficient implementation. Additionally, the manual rewriting in this step simplifies and cleans the code.

Next to the combination of ATOMIUM/Analysis with a profiling tool for feedback on the effect while applying the high-level optimizations, the ATOMIUM tool framework also provides support for some of the optimization steps in the atomium script:

- ATOMIUM/MH: optimizes data reuse and memory hierarchy usage.
- ATOMIUM/MA: defines an optimal memory architecture.
- ATOMIUM/MC: optimizes the amount of storage required.

2.2 Parallel Design Phase: Steps and Tools

The second phase selects a suited partitioning of the application and calculates the corresponding buffer sizes using a Cyclo-Static DataFlow (CSDF) MoC (Chapter 3). Each resulting process is then mapped on a processor or translated to HDL separately. The last design step gradually composes these functional components to build up the final system.

2.2.1 Partitioning

To achieve the required high throughput rates, the partitioning introduces concurrency by deriving a suited split of the application in parallel processes that, together with the memory hierarchy, define the system architecture. Figure 2.4 illustrates this partitioning exploration using the SPRINT tool [2]. From the localized dataflow of the optimized specification, the designer extracts a CSDF [19] graph. This CSDF graphs exploits the special channels of Chapter 3 to be able to retain the effect of the high-level optimizations (e.g. data reuse) leading to an efficient implementation. The selected PM is mainly based on a message passing system and is defined as a limited set of communication primitives (Chapter 3). The division of the application over the different actors, together with the calculation of the buffer sizes of the edges between them (see Chapter 3), leads to a parallel system with self-timed execution.

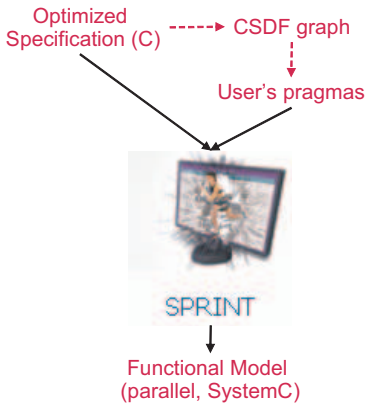


Figure 2.4: The SPRINT tool automatically generates a parallel SystemC model from sequential C code based on user's pragmas indicating the partitioning.

The partitioning exploration is supported by the SPRINT tool that, based on the requested process boundaries, automatically translates a sequential description (the optimized specification) into a functionally equivalent concurrent SystemC model consisting of tasks and communication channels. Although the designer defines the task boundaries, the communication channels are automatically detected by the tool and mapped on the set of communication primitives of the chosen PM. By including externally provided cycle estimations, the un-timed model is refined to a timed SystemC model. Additionally, the SPRINT tool provides a link to the RTL development environment of Chapter 4 with the option to generate a specification file, a stimuli model or a prototype test.

The actor/process boundaries of the CSDF graph are described as a set of user pragmas and fed together with the optimized specification to SPRINT (Figure

2.4). Before moving to the (work intensive) RTL description, the automatically generated (parallel) functional model allows an evaluation of the concurrency of the proposed partitioning: (i) verify the functional correctness and (ii) assess the parallelism between different processes (directly impacting the final achieved throughput). Both aspects depend on correctly calculating the buffer sizes of the CSDF edges. Chapter 4 explains the functional verification in detail.

During the partitioning, the code of the optimized specification is refined to simplify the exploration. The code is reorganized so that: (i) a C function corresponds to a functional module and (ii) function variables are grouped according to their communication type (detailed by using comments).

Once the partitioning is fixed, the timing specifications for the RTL design of the resulting processes are derived from their repetition rates in the CSDF graph. These timing specifications are expressed as target response times: the product of the available number of cycles with operation frequency for a process. When the operation frequency is fixed, it defines the cycle budget or when the number of cycles is given for a process, it defines the required operation frequency.

2.2.2 RTL development and SW refinement

During this design step, each process of the partitioned system is implemented separately, either as a HW component (RTL development) or mapped on a suited processor (SW refinement). As the main focus is to create a fully custom system, most of the processes are translated to an HDL description.

Enabling SPRINT's link to the RTL development and verification creates a specification file (SPEC) for the process of interest. This SPEC file is used by the SPEC2VHDL and SPEC2FPGA scripts to automatically create an environment easing the HW description step of the functional component (Chapter 4). Clearly separating the communication, described as CPs in the PM using a HDL language, and the computation allows the HW design and functional verification of an isolated component. Inserting probes in the functional model generates the necessary input stimuli and the expected output. In this way, each individual functional component is tested extensively, by combining simulation and emulations on a prototyping platform for functional verification to minimize the debug cycle during the integration design step. The development environment supports both test mediums, allowing the designer to concentrate on the functional description of the component and its implementation.

The timing constraints for the functional component under development result from the previous step. It is now the task of the designer to find a suited architecture and schedule so that for a minimum HW cost, the required throughput

is obtained. If the current functionality is too complex or too large to implement, and hence possible leads to a critical path or a large area consumption, a re-iteration over the partitioning, further splitting the current process, can be applied.

2.2.3 Integration

Using the same RTL development and verification, the different functional components are gradually combined and tested until they form the complete HW part of the system. Finally they are mapped together with the selected processor(s) for the remaining SW on the final target platform.

2.3 Design Environment

To assure a successful application of the design flow on a practical example, a design environment needs to be set up, supporting the verification of the design. The next subsections describe some elements of such design environment.

2.3.1 Testbench

At the start of the design, a testbench is defined triggering all video tools of the selected application profile. To ensure a maximum code coverage, video sequences of different spatial and temporal resolution, and with different motion characteristics are included. This test set is used throughout the design for functional verification and is automated by scripts. An example testbench is given in Section 5.1.

2.3.2 Programming Model Libraries

The principle of separation of computation and communication allows the implementation and debugging of the communication primitives of the PM (section 3.5) at the different levels of the design flow in parallel with the application. In this way, they are available in time for the application design.

2.3.3 Fast Prototyping Board

In parallel with the design of the video application, the fast prototyping board is prepared to allow its usage during the RTL development and the integration

steps. Chapter 4 combines simulation with fast prototyping for functional verification.

2.4 Related Work

The design experiences of [72] on image/video processing indicate the required elements in rigorous design methods for the cost efficient hardware implementation of complex embedded systems: higher abstraction levels and extended functional verification. We establish a design flow tailored to block-based video processing that addresses these aspects. The flow covers the abstraction levels from sequential (C) specification to RT-level. It combines and extends known approaches and techniques to obtain an extremely low-power implementation. More specifically, it builds on the mature DTSE methodology to deal with the data dominance of video processing.

Design flow related literature is extensive. A good overview of embedded system design is given in [29]. Many papers only discuss an aspect or a subpart of the design flow. In contrast, we present a complete design flow, building on top of existing knowledge. To illustrate the parts of the design space covered by the proposed flow, Figure 2.5 couples the Y-chart [47] to the different levels in Transaction Level Modeling (TLM) [43], ordered as a design pyramid. The sequential phase refines the system specification from the system to algorithmic level of the Y-chart using an algorithmic (ALG) TLM model, i.e. sequential C code. The parallel phase makes the transition to the RT-level of the Y-chart. During its partitioning step, an untimed communicating processes (CoPr) and a timed communicating processes (CoPr + T) TLM model generated by SPRINT is used. The RTL development step makes the translation to RTL. The Programmer's View (PV) and Cycle Accurate (CA) levels are not addressed by the proposed design flow.

The Data Transfer and Storage Exploration (DTSE, [22, 85]) presents a set of loop and dataflow transformations and memory organization tasks to improve the data locality of an application. In this way, the dominating memory cost factor of multimedia processing is tackled at the high level. We combine them during the sequential phase with algorithmic optimizations that comply with the DTSE rules.

The DTSE optimization steps are organized to connect well to the partitioning phase. This parallelization uses a Cyclo-Static Dataflow (CSDF, [19]) graph. This model of computation is used in a non-trivial way (Chapter 3) to express the special communication channels resulting from the DTSE optimizations.

The selected and clearly defined set of communication primitives is a key element of the proposed design flow. It allows exploiting the principle of

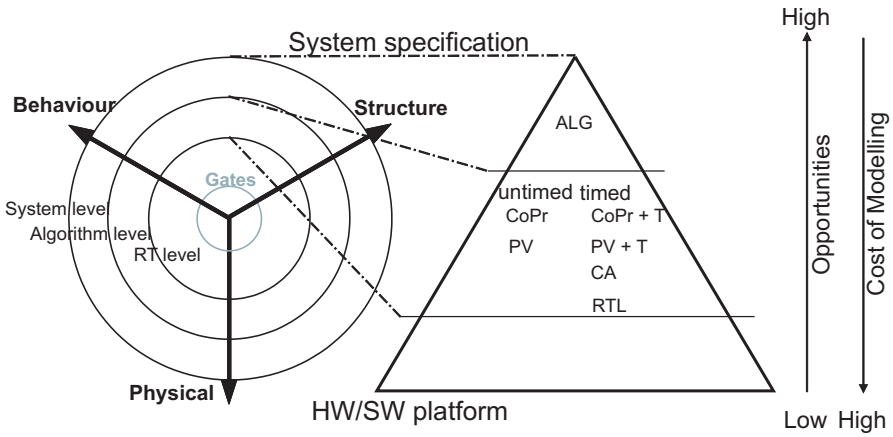


Figure 2.5: The Gajski-Kuhn Y-chart linked to the different abstraction levels of transaction level modeling presented in a design pyramid.

separation of communication and computation [65] and enables the automated RTL development and verification strategy of Chapter 4 combining simulation with fast prototyping.

2.5 Conclusion

The design of video cores supporting complex algorithms at a high-throughput in a cost-efficient way requires translating a high-level specification (typically C) into the final implementation. Still today, this development means overcoming the gap between system definition and modeling to the final realization. A solid design flow is required to overcome this gap and to gradually reach the implementation meeting all design specifications.

We proposed a stepwise design and test methodology, oriented to energy efficient, block-based video processing, that supports the designer towards a correct system description at the RTL level (contribution 1). The design flow covers different levels of abstraction of the design space to master the complexity of the design. The typical data dominance of multimedia systems motivate its memory and communication focus.

The energy efficiency of the implementation is measured as an energy delay product. The design steps are ordered with respect to their impact on this metric and their development effort. This results in a design flow with 2 major phases: (i) a sequential phase for problem simplification reusing DTSE principles and (ii) a parallel phase introducing concurrency and focusing on the support for HW development.

In the first phase of the design flow, the reference code is transformed through memory and algorithmic optimizations into a block-based video application with localized processing and dataflow. Memory footprint and frame memory accesses are minimized. These optimizations enable an efficient use of the fixed set of communication primitives. The second design phase builds a dataflow model of the application to support its partitioning in a set of concurrent processes to achieve the required throughput. Each process is mapped on a suited processor or translated to an HDL description. Finally, the complete system is integrated.

Chapter 3

Cyclo-Static Dataflow and Implementation Modeling

You see I've been somewhere
Not far away but such a different space
Just to sit in one place
And just when I thought I could take no more
The bells started to ring
And my soul to sing
And I know that what is
Just is

'Just Is' – Lamb

The increasing complexity and concurrency in digital multi-processor systems used to build modern multimedia codecs or wireless communications require a design flow covering different abstract layers to gradually evolve towards a final, efficient implementation. Describing the system first at higher abstraction, using a Model of Computation (MoC), helps the designer with design space exploration: it permits reasoning about the system and enables the usage of synthesis techniques.

Dataflow MoCs have proven to be useful for describing multimedia processing applications [100] as they enable a natural visual representation exposing the parallelism and allowing an evaluation of the temporal behavior. Cyclo-Static DataFlow (CSDF) [19] is particularly interesting because this variant is one of the most expressive dataflow models while still being fully analyzable at design time (e.g. consistency checks, dead-lock analysis).

However, dataflow MoC also have restrictions: they assume infinite buffer capacity, they only consider point to point communication, they have no direct means to describe shared memory, etc. On the other hand, an implementation on a multi-processor platform has optimized communication channels, often based on shared buffers, to improve the efficiency. Examples are a sliding window for data reuse or a circular buffer with multiple consumers. Also, due to implementation restrictions, buffer sizes are limited. As it is not always clear how the behavior of such channels can be expressed in a CSDF model, the designer could judge it as an unsuited MoC, thus losing its analysis potential.

This chapter studies how such implementation aspects can be represented in a CSDF model within its current definition. The main contribution is the modeling of special behavior on channels which is required to obtain an efficient implementation, like data re-use or shared buffers. The proposal of a short-hand notation for these special channels provides an intuitive expression of shared memory related aspects in CSDF without requiring extensions of the MoC. As a result, the enriched CSDF graph remains fully analyzable at design time and allows reasoning about the temporal behavior. The capabilities of the approach are demonstrated in Chapter 5 by describing a power-efficient custom implementation of an MPEG-4 part 2 video encoder using these special channels.

The special channels and the limited buffer sizes are modeled in CSDF by representing them by two edges, one forward edge assuring the synchronization and one backward edge monitoring the free buffer space. Conditions are formulated on those two edges to assure functional correctness of the modeled application (i.e. no overwriting of live data) and these conditions are verified for every special channel. A basic technique for the buffer capacity calculation through life-time analysis is presented.

In a custom hardware design, the special channels are supported by a limited set of Communication Primitives (CPs) consisting of 2 groups: synchronizing and non-synchronizing elements. Based on the principle of redundant synchronization [100], allowing to remove the synchronization on a path between 2 tasks if another path still keeps these tasks synchronized, all special channels can be realized using non-synchronizing CPs combined with a path of synchronizing CPs.

This chapter is organized as follows. After summarizing dataflow theory and introducing the basics of CSDF in the next section, the modeling of an implementation is discussed in Section 3.2. The description of special channels in CSDF is explained in Section 3.3. In Section 3.4, an approach for the buffer length calculations is presented. The library with only a limited set of communication primitives supporting the use of the special channels in the HW is described in Section 3.5.

3.1 Dataflow Models

In the application specific domain, specialized MoCs like dataflow models aid in identifying and exploring the parallelism, and in the manual or automatic derivation of optimized implementations [100]. The choice of the model of computation is a trade-off between its expressiveness and its analysis potential. Figure 3.1 arranges different dataflow process network MoCs [45] according to this trade-off. The data streaming nature of video processing and the highly concurrent target architecture match well with a Kahn Process Network (KPN) representation that is on the one end of the classification. Unfortunately, this model of computation does not support taking decisions on the scheduling and buffer sizes at design time. At the other side of the classification, (homogeneous) synchronous dataflow is a refinement of KPN that has the analysis potential for scheduling and buffer sizing at design time, but it restricts the expressiveness excessively [32]. The presented classification only covers a subset of the existing MoCs. Many other types, for instance supporting more dynamism, are described in [45].

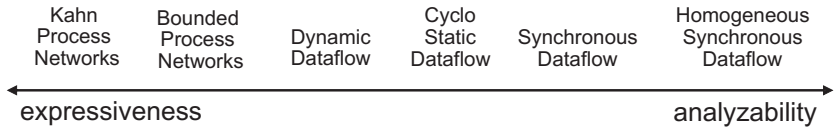


Figure 3.1: Classification of different MoCs.

In this work, a dataflow model is chosen as it combines the readability of block diagrams and signal flow charts with a clear semantics for system design and analysis tools [100]. More specifically, the cyclo static variant is selected as it offers the best expressiveness while still being fully analyzable at design time. The following subsections (3.1.1, 3.1.2 and 3.1.4) are based on [10, 19, 100].

3.1.1 Definitions of Dataflow Theory

A comprehensive introduction to dataflow modeling is included in [18, 100]. This subsection gives a summary to introduce the dataflow definitions and terminology. In dataflow, the application is described as a directed graph $G = (V, E, \delta, \gamma, \pi, \rho)$ that consists of a finite set of actors V and a set of directed edges, $E = \{(v_x, v_y) | v_x, v_y \in V\}$. Figure 3.2 shows an arbitrary multigraph, a generalization of a directed graph in which two or more edges can exist between the same actors. The actors correspond to the subtasks of the application, transforming input data into output data. They are by definition atomic (i.e. indivisible). The directed edges (arcs) represent channels carrying data between the communicating actors. The edges act as First-In-

First-Out (FIFO) queues with a theoretically unlimited depth. A token is a synchronizing communication object. It can be used to represent containers or just to model synchronization. Containers are fixed size data structures. The initial token placement $\delta : E \rightarrow \mathbb{N}$ represents delays on the edges. They can be interpreted as dependencies across iterations of the graph.

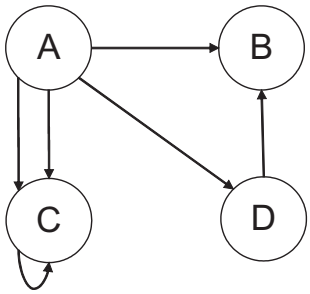


Figure 3.2: A directed multigraph.

The actor execution is data-driven: it is enabled to fire as soon as sufficient tokens are available on all inputs (i.e its firing-rule, a boolean expression in the number and/or the value of tokens turns true). An actor consumes tokens from its input edges in one atomic action at the start of the firing and writes tokens on its output edges in one atomic action at the end of the firing. The number of consumed tokens per firing is given by $\gamma : E \rightarrow \mathbb{N}$. Actor A therefore consumes $\gamma(e_u) = c_A^u$ tokens per firing from edge $e_u = (v_y, A)$. The number of produced tokens per firing is given by $\pi : E \rightarrow \mathbb{N}$. Actor A therefore produces $\pi(e_u) = p_A^u$ tokens per firing on edge $e_u = (A, v_y)$. Figure 3.3 shows a single actor with production and consumption rules on its edges. Multiple simultaneous firings of the same actor are known as auto-concurrency in dataflow theory. The Response Time (RT) $\rho(A)$ of an actor A is the elapsed time between its enabling and the end of the firing, with $\rho : V \rightarrow \mathbb{N}$.

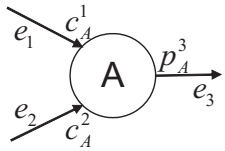


Figure 3.3: An actor A in a dataflow graph with 3 edges can fire a soon as there are at least c_A^1 tokens on edge e_1 and at least c_A^2 tokens on edge e_2. It produces p_A^3 tokens on edge e_3 at the end of the firing.

The data-driven operation of a dataflow graph allows synchronization between the actors: an actor can not be executed prior to the arrival of its input tokens. When a solution exists for the balance equations (see Section 3.1.4) of a dataflow graph, it is said to be consistent, i.e. it can run without a continuous

increase or decrease of tokens on its edges. A dataflow graph is called non-terminating or live if it can run forever, i.e. all parts can run infinitely often. On the contrary, it has a deadlock if it has a maximal execution of finite length. A dataflow graph is bounded, when an implementation of this graph is feasible using a limited amount of memory. Consistency is a necessary and sufficient condition for the boundedness of a live dataflow graph. More details on liveness and boundedness of dataflow graphs is given in [49].

A schedule of a dataflow graph specifies: (i) the assignment of the actors in the graph to processors, (ii) the execution ordering of these actors on each processor and (iii) the actor firing times, so that all data dependency constraints are met. Each of these 3 task may be performed either at run-time (a dynamic strategy) or at compile time (a static strategy) [100]. If the scheduling approach performs all 3 scheduling tasks at compile time, the result is called a fully-static schedule. In this work, a relaxed variant is applied, called self-timed execution: the assignment and ordering are done at compile time but the determination of the start times are done at run time. Such schedule can be easily implemented (in hardware). Often a schedule is executed repetitively on a stream of input data, called iterations of a periodic schedule.

For a DSP-application, both the liveness and consistency of the graph are required to get a proper execution. A forever running execution can be obtained by repeating one iteration of a periodic schedule [70]. To keep the number of tokens on the edges limited, the number of tokens produced on an edge during one period of this schedule must equal the number of tokens consumed from it. This condition is expressed by the balance equations (see also Section 3.1.4). The number of actor firings in one period can be derived from this consistency requirement. The existence of a deadlock-free schedule for one iteration [70] is a sufficient condition for a graph to be live. Any such schedule is called a valid static schedule of the graph.

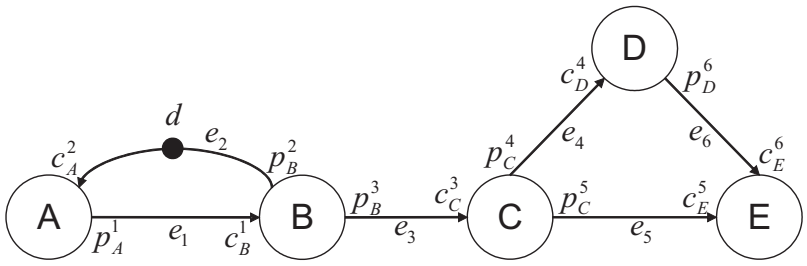


Figure 3.4: An arbitrary dataflow graph with 1 chain (B, C) and 2 clusters (A, B) and (C, D, E). Cluster (A, B) is also a cycle.

The following definitions, illustrated in Figure 3.4 are used in the rest of text:

- *path*: a path of length k from an actor v_x to an actor v_y in a graph

$G = (V, E)$ is a sequence $< v_0, v_1, v_2, \ldots, v_k >$ of actors so that $v_x = v_0$ and $v_y = v_k$ and $(v_{i-1}, v_i) \in E$ or $(v_i, v_{i-1}) \in E$ for $i = 1, 2, \ldots, k$. The length of the path is the number of edges in the path.

- *open path*: is a path $< v_0, v_1, v_2, \ldots, v_k >$ with $v_0 \neq v_k$
- *closed path*: is a path $< v_0, v_1, v_2, \ldots, v_k >$ with $v_0 = v_k$
- *chain*: open path of edges and actors independent of the orientation of the edges for which only one path exists between any pair of actors in the chain.
- *cluster*: a graph in which every pair of actors is interconnected by at least two distinct paths, independent of the orientation of the edges. Consequently, a chain can never be part of a cluster.
- *cycle*: closed path of edges and actors in which each actor occurs only once. Every edge e_i leaves actor v_{i-1} and enters actor v_i.

3.1.2 Different Dataflow Model Types

Depending on how the consumption and production together with the firing rules are specified, different classes of graphs are distinguished: homogeneous synchronous dataflow, synchronous dataflow, cyclo-static dataflow, and dynamic dataflow [10, 19].

3.1.2.1 Homogeneous Synchronous Dataflow

In Homogeneous Synchronous Data Flow (HSDF) or single-rate dataflow [100] the consumption and production on each edge is a single token. An actor can fire if there is at least one token on all its incoming edges. An application that can be described using a HSDF model is guaranteed to be deterministic, i.e. the output-data only depends on the input data and not on the order in which the individual tasks are executed nor on their execution or communication times [10]. Consequently, a static schedule can easily be constructed by tools at design time. Figure 3.5 instantiates the arbitrary dataflow graph of Figure 3.4 as a HSDF graph by setting all production and consumption amounts to one.

Most signal processing applications contain tasks that operate at different rates. They are more easily described in Synchronous Dataflow (SDF).

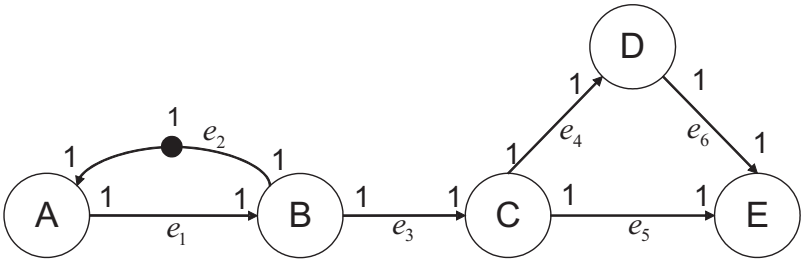


Figure 3.5: An example HSDF graph.

3.1.2.2 Synchronous Dataflow

In a synchronous [70] or multi-rate dataflow description, consumption and production are constant integers known at design/compile time. In the graph, they are noted at the corresponding side of the edge. When an actor P fires, it produces p_P^u tokens on edge e_u. Actor C consumes c_C^u tokens from this edge each time it is invoked. The firing rule of an actor only becomes true if each input contains at least the consumption amount of that input.

The consistency of an SDF graph depends on the relative production and consumption rules on each of the edges. Its consistency condition requires solving its balance equations. Note that every SDF graph can be converted in an equivalent HSDF graph [100]. Hence, every application fitting in a SDF model is deterministic. Figure 3.6 instantiates the arbitrary dataflow graph of Figure 3.4 as an SDF graph by setting all production and consumption amounts to values greater than or equal to one in such a way that the balance equations are respected.

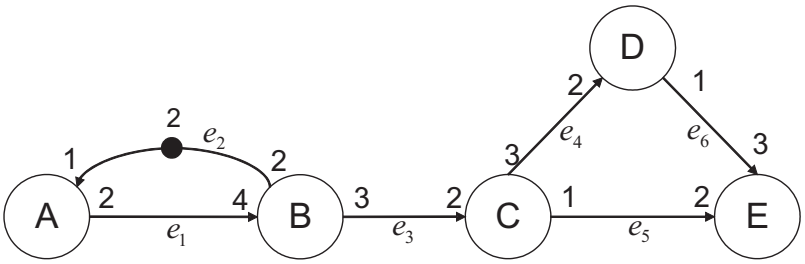


Figure 3.6: An example SDF graph.

The main limitation of the SDF model is the assumption that every subtask (actor) behaves identically for each firing: consuming and producing a fixed number of tokens. To allow the description of state-dependent behavior (i.e. cyclic behavior), the cyclo-static dataflow model is introduced [19].

3.1.2.3 Cyclo-Static Dataflow

Cyclo-Static DataFlow modeling was first proposed by Bilsen [19] as an extension of SDF. In CSDF, each actor A has an execution sequence of length L_A, called the actor period. Consequently the production and consumption are also sequences of constant integers noted on the corresponding side of the edge e_u as $\{p_P^u(0), p_P^u(1), \ldots, p_P^u(L_P - 1)\}$ for the producing actor P and $\{c_C^u(0), c_C^u(1), \ldots, c_C^u(L_C - 1)\}$ for the consumer. During its firing i, actor P produces $p_P^u(i \bmod L_P)$ tokens on edge e_u. Similarly, firing j of actor C consumes $c_C^u(j \bmod L_C)$ tokens from the same edge. Again, the firing rule of an actor A becomes true for invocation j if all inputs contain at least $c_A(j \bmod L_A)$ tokens. Note that the first firing is labeled as invocation 0 and that firing sequences are read from left to right.

Also a CSDF graph is deterministic. Its consistency can be evaluated through the balance equations and a valid static schedule can be found [19] at compile time. Reference [89] shows that a CSDF graph can be transformed in a SDF graph.

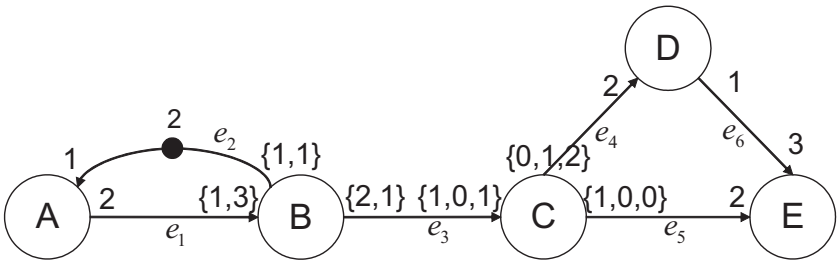


Figure 3.7: An example CSDF graph.

Figure 3.7 instantiates the arbitrary dataflow graph of Figure 3.4 as a CSDF graph. Actor C has a state-dependent multiplexer behavior: only the first firing of each actor period sends tokens to actor E while the other two firings send tokens to actor D.

If the application also contains subtasks with data dependent behavior a Dynamic Dataflow (DDF) model is required.

3.1.2.4 Dynamic Dataflow

In general DDF, all types of conditionals are supported while the graph remains fixed. The firing-rule can be any Boolean expression of token amounts or values, and the consumption and production may be unknown at compile-time because they depend on the actual value of tokens. Because of the partial knowledge at compile-time, DDF needs a run-time scheduling mechanism to

determine in which order nodes can be fired. Moreover, it is not always possible to derive whether the graph is deterministic, consistent or deadlock-free at compile-time. As a result, a precise upperbound calculation of the required buffer sizes is hindered.

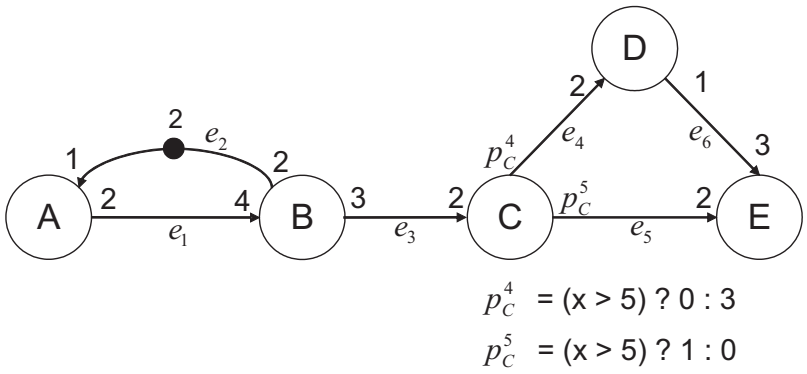


Figure 3.8: An example DDF graph. x is an input value read from edge e_3.

Figure 3.8 instantiates the arbitrary dataflow graph of Figure 3.4 as a DDF graph. Actor C has a data-dependent multiplexer behavior: if x, a value read or calculated from e_3, is larger than 5, one token is sent to actor E, otherwise 3 tokens are sent to actor D.

3.1.3 Temporal Monotonic Behavior

The data-driven operation of a dataflow graph allows its execution in a self-timed manner: actors start as soon as they are enabled. Additionally, the FIFO ordering of the tokens assures they can not overtake each other. The FIFO ordering of the tokens is automatically respected on the edges of a dataflow graph as these edges act as queues. In the actors, the FIFO ordering is guaranteed if auto-concurrency is excluded by a self-cycle with a single token, forcing sequential firing of this actor or by making the response time of the actors constant.

These two properties are a sufficient condition for the definition in [92, 110, 111] of the monotonic execution of a CSDF graph G as follows: a CSDF graph G executes monotonically in time if no decrease in response time or start time of any firing k of any actor A can lead to a later enabling of any firing l of any other actor. It is shown that a dataflow graph with self-timed execution, that maintains the FIFO ordering of the tokens, possesses this important property of monotonic behavior in time. As a result, a decrease in response time can only lead to earlier token production and consequently to an equal or earlier actor enabling. Overall, this could possibly lead to a higher throughput.

In this work, the focus is on cyclo-static dataflow [19] as it is deterministic and allows checking conditions such as deadlocks and bounded memory execution at compile or design time. For DDF, this is not always possible, making DDF less interesting to dimension a real-time implementation under worst case assumptions. Additionally, if dynamic dataflow concepts are required to model a multi-media application, this is often only needed for a part of the graph and can sometimes be reduced to CSDF by considering worst-case scenarios [103].

After introducing the elements and properties of CSDF in the next subsection, it will be shown that a consistent relation exists between CSDF model and implementation. As a result, containers will not arrive later in an implementation with self-timed execution than the corresponding tokens in the CSDF model. If worst-case response times are used while building this schedule, the worst-case throughput is known and guaranteed.

3.1.4 Basics of Cyclo-Static Dataflow

This subsection briefly explains how the consistency and liveness of a CSDF graph is evaluated. More details are given in [19, 100]. The following notations are used in this text:

- L_A actor period or cycle length of the sequences of actor A
- $\{p_A^u(0), p_A^u(1), \ldots, p_A^u(L_A-1)\}$ production sequence of actor A
- $\{c_A^u(0), c_A^u(1), \ldots, c_A^u(L_A-1)\}$ consumption sequence of actor A
- $p_A^u(i)$ number of tokens produced on edge e_u by actor A during its firing i

$$p_A^u(i) = \begin{cases} \text{element } i \text{ in the production sequence} & \text{if } 0 \leq i \leq L_A - 1 \\ p_A^u(i \bmod L_A) & \text{if } i \geq L_A \end{cases}$$

- $c_A^u(j)$ number of tokens consumed from edge e_u by actor A during its firing j

$$c_A^u(j) = \begin{cases} \text{element } j \text{ in the consumption sequence} & \text{if } 0 \leq j \leq L_A - 1 \\ c_A^u(j \bmod L_A) & \text{if } j \geq L_A \end{cases}$$

- $P_A^u(k)$ number of tokens produced on edge e_u by actor A after k firings

$$P_A^u(k) = \sum_{i=0}^{k-1} p_A^u(i) \tag{3.1}$$

- $C_A^u(l)$ number of tokens consumed from edge e_u by actor A after l firings

$$C_A^u(l) = \sum_{j=0}^{l-1} c_A^u(j) \tag{3.2}$$

- q_A^b basic repetition rate of actor A (see Equation (3.6))

A CSDF graph G is compactly represented by its *topology matrix* Γ containing one column for each actor and one row for each edge. Its $(i,j)^{th}$ entry corresponds to the total number of tokens produced/consumed by the actor with number j on the edge with number i during one actor period of j. If the actor with number j produces tokens, the entry is positive while for a consuming actor, the entry is negative. The *actor period matrix* L contains one row with the actor periods (cycle lengths). Its j^{th} entry holds the actor period of the actor with number j.

A *period balance vector* r is a positive solution of the balance equations

$$\Gamma \cdot r^T = 0 \tag{3.3}$$

Such a period balance vector only exists if

$$\text{rank}(\Gamma) = N_G - 1 \tag{3.4}$$

with N_G the number of actors in the CSDF graph. A *repetition vector* q translates the number of actor periods of r to a number of actor firings. It is the product of the period balance vector with the actor period matrix

$$q = r \cdot L_{diag} \tag{3.5}$$

with L_{diag} the diagonal version of L. All repetition vectors are a multiple of the *basic repetition vector* q^b, which can be derived from any arbitrary repetition vector q as

$$q^b = \frac{q}{s}, \text{ with } s = \underset{y \in V}{\text{gcd}}(r_y) = \underset{y \in V}{\text{gcd}}(\frac{q_y}{L_y}) \tag{3.6}$$

The existence of a repetition vector (consistency) is a necessary condition for bounded memory execution (boundedness). However this criterion is still not sufficient to guarantee the existence of a valid static schedule due to possible deadlock problems. To check if such a schedule with basic repetition vector q^b actually exists for a consistent (C)SDF graph, [19, 70] propose to construct a single-processor schedule for one iteration, i.e. in which each actor A fires at least q_A^b times.

Example: The topology matrix Γ of the CSDF graph in Figure 3.7 is derived by giving the actor letters a corresponding number. Row 3 corresponds to edge e_3 with actor B producing 3 tokens during $L_B = 2$ and actor C consuming 2 tokens during $L_C = 3$.

$$\Gamma = \begin{pmatrix} 2 & -4 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 & 0 \\ 0 & 3 & -2 & 0 & 0 \\ 0 & 0 & 3 & -2 & 0 \\ 0 & 0 & 1 & 0 & -2 \\ 0 & 0 & 0 & 1 & -3 \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & 2 & 3 & 1 & 1 \end{pmatrix}$$

The rank of $\Gamma = 4$, indicates a solution exists for the balance equation. A valid period balance vector is

$$r = \begin{pmatrix} 8 & 4 & 6 & 9 & 3 \end{pmatrix}$$

The repetition vector $q = r \cdot L_{diag}$ equals

$$q = \begin{pmatrix} 8 & 8 & 18 & 9 & 3 \end{pmatrix}$$

The basic repetition vector q^b equals the repetition vector q as the greatest common divider of the elements of vector r is 1.

3.2 Using CSDF to Model Implementation Aspects

The implementation of an application can be represented as a directed task graph [111] consisting of tasks communicating through FIFO buffers with fixed capacity, called regular channels (Figure 3.9(a)). Only containers, communication units with room for a fixed amount of data, are communicated over these FIFOs. These containers can be free (available for data production) or completed (no longer available for data production and either partially filled or full). Note the difference with a dataflow model where a token can represent a container or just synchronization. Tasks have production and consumption sequences and can only start if sufficient completed containers are present on their input FIFOs and sufficient free containers are available in their output FIFOs. More specifically, executing a task consists of the following steps: (i) acquire: check the availability of the completed input containers and free output containers, (ii) execute the code of the function describing the task

behavior (accessing the data in the container) and (iii) release: signal the completion of the production of the output containers and the finishing of the consumption of the input containers. The elapsed time between the successful acquiring and releasing in a task execution is bounded by the worst-case execution time, known at design time. Finally, it is assumed that at most one instance of a particular task can execute at any time. This is important when the task keeps an internal state with data that is needed during a next execution and to maintain the FIFO ordering of the containers.

In a real implementation, also other communication types than the regular channel are deployed, often to optimize the data transfer. Examples are a sliding window for data reuse or a shared buffer with multiple consuming tasks. Such communication types are called special channels. The next subsections describe how the regular channel and some types of special channels can be expressed with a CSDF graph. Their CSDF representation is essential to be able to use the design time analysis techniques of CSDF.

3.2.1 Blocking Write and Blocking Read

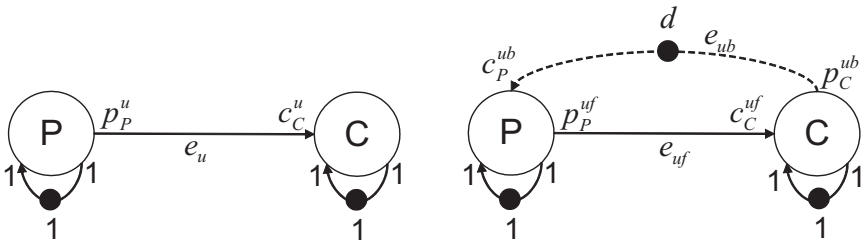
In the modeling of such an implementation task graph as a CSDF graph, a task corresponds to an actor with a response time equal to the task's worst-case execution time. The acquire and release of containers in the implementation are respectively represented by the removal and arrival of tokens on the edges in the CSDF model. While a container is always represented by tokens in the dataflow model, the inverse is not necessarily true, as tokens can also express synchronization only. For example, a self-cycle on each actor models that no two instances of a task can execute simultaneously.

The blocking read behavior of a FIFO queue (i.e. the stalling of the consuming task because the queue is empty) is modeled by the data-driven operation of the actors. Because of the fixed depth of the FIFO queue, it also has a blocking write: the producing task is halted as long as the FIFO is full. This blocking read and blocking write behavior can be represented by a pair of queues in opposite direction [100, 102] in the CSDF graph (Figure 3.9(b)). The tokens on the forward queue e_{uf} (from producer P to consumer C) represent completed containers while the tokens on the feedback queue e_{ub} indicate the free containers. The fixed size of the FIFO buffer (i.e. its depth expressed as the number of containers it can maximally hold) is modeled by the number of initial tokens d on e_{ub} for an initially empty FIFO.

The tight coupling between the tokens and the containers is expressed by requiring that, at the end of the task execution, a producing or consuming task releases all containers acquired at the start of the task invocation.

$$\forall i, j \in \mathbb{N} : p_P^{uf}(i) = c_P^{ub}(i) \text{ and } c_C^{uf}(j) = p_C^{ub}(j) \tag{3.7}$$

Consuming c_C^{uf} tokens from e_{uf} releases the corresponding containers, but only at the end of the firing with the production of the same number of tokens p_C^{ub} on e_{ub}. To produce p_P^{uf} tokens representing completed containers at the end, the same number c_P^{ub} of them is consumed at the start of the firing, expressing the acquiring of the containers. Consequently, the tokens on the two edges represent correctly how the containers are used in the task graph: acquiring at the start of the execution and releasing at the end of the execution.



(a) Regular channel (b) CSDF equivalent

Figure 3.9: The feedback edge e_{ub} limits the size of edge e_u to d.

Note that the presence of self-cycles with one initial token, as shown in Figure 3.9(b), is assumed but not drawn in the following CSDF graphs of this text.

3.2.2 Decoupling Tokens from Containers

The tight coupling of tokens and containers in a regular channel represents the most common interpretation of the behavior of an edge in a dataflow model: a container is released from/to the edge after a single firing. Figure 3.10 illustrates the data reuse in the overlapping regions of the search area data during the motion estimation of a video encoder [94]. Such sliding window behavior can not be modeled with the common CSDF interpretation, since the complete dashed search area is required as firing condition and consequently, it will be released entirely from the edge after the first execution of the motion estimation task.

Similarly, the production of a container over multiple task executions can not be expressed in the common CSDF interpretation as the acquired containers at the start, are released to the consuming task at the end of the same invocation. Finally, edges represent point-to-point communication, hindering the expression of shared containers between multiple tasks.

Figure 3.10: Data reuse in the overlapping regions of the search area data for motion estimation.

Relaxing the requirement in Equation (3.7) allows breaking this tight relation between tokens and containers and enables the modeling of special data communication. During a firing of the producer, the number of produced tokens p_P^{uf} on e_{uf} can differ from the number of consumed tokens c_P^{ub} from e_{ub}. Similarly, a consumer firing can consume a different number of tokens from e_{uf} than the number produced on e_{ub}. The obtained decoupling of tokens and containers and its use to model special channels (described in the next subsection) without extending CSDF is the core of contribution 2. It allows the leverage of the formal reasoning on CSDF models to include the behavior of optimized communication.

In the example of Figure 3.10, this decoupling of tokens and containers allows releasing only the left, non-overlapping part of the search area (p_C^{ub}), while the complete search area was required to enable the execution of the motion estimation (c_C^{uf}), with $p_C^{ub} < c_C^{uf}$. The next subsection discusses the behavior of this special channel and other types (dealing with the other restrictions listed above) in detail.

Bounded memory condition: To maintain bounded memory execution, during one period of the producing task, the sum of acquired containers at the producer should equal the sum of completed containers (first equality of Equation (3.8)). Similarly, during one period of the consumer, the sum of released containers has to equal the sum of consumed completed containers (second equality of Equation (3.8)).

$$\sum_{i=0}^{L_P-1} p_P^{uf}(i) = \sum_{j=0}^{L_P-1} c_P^{ub}(j) \text{ and } \sum_{j=0}^{L_C-1} c_C^{uf}(j) = \sum_{i=0}^{L_C-1} p_C^{ub}(i) \tag{3.8}$$

Using the short notation of Equations (3.1) and (3.2)

$$P_P^{uf}(L_P) = C_P^{ub}(L_P) \text{ and } C_C^{uf}(L_C) = P_C^{ub}(L_C) \tag{3.9}$$

Mutual exclusiveness condition: Additionally, at any moment at the producing task, the sum of completed containers should not be larger than the sum of acquired containers to avoid writing in a non-free container.

$$\forall k \in \mathbb{N}_0 : C_P^{ub}(k) \geq P_P^{uf}(k) \tag{3.10}$$

Data preservation condition: Similarly, at any moment at the consuming task, the sum of released containers should not be larger than the sum of acquired new containers to avoid loss of data.

$$\forall k \in \mathbb{N}_0 : P_C^{ub}(k) \leq C_C^{uf}(k) \tag{3.11}$$

The number of free containers f in the buffer of edge e_u after k firings of P and l firings of C is

$$f = d - C_P^{ub}(k) + P_C^{ub}(l) \tag{3.12}$$

where the fixed size of the buffer of the channel is modeled by the number of initial tokens d on e_{ub} for an initially empty channel.

3.3 Modeling Special Channels

Using the decoupling of tokens and containers, the following subsections present some interesting cases of modeling special behavior on edges of the task graph. For each of these special channels, a CSDF equivalent is given when possible. If the equivalent exists, the special channel becomes a short-hand notation for the CSDF graph.

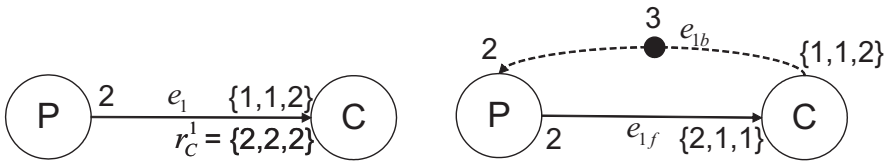
The applied decoupling of tokens and containers is a non-standard interpretation of CSDF. An alternative is a more straightforward representation of the special channels in strict CSDF. This results in more expensive equivalents in terms of required buffer space and token exchange rate. Appendix C presents strict CSDF equivalents for most of the special channels. The extra buffer space of the strict CSDF equivalents reaches 50 % for a specific edge of the MPEG-4 encoder design (see Table 5.14).

3.3.1 Non-Destructive Read

A non-destructive read special channel supports the reuse of data during a next invocation. To achieve this, a consumer can acquire more containers than it will release at the end. In a video codec implementation, data reuse enables copying repetitively accessed data to a smaller and more efficient buffer, and avoids refreshing data in overlapping regions of for instance the motion estimation (see Figure 3.10 in Section 3.2.2).

A simple example is shown in Figure 3.11(a). $r_C^1 = \{2, 2, 2\}$ indicates that C can only start if 2 completed containers are available, while its consumption sequence only releases both of them at the end. The buffer occupancy is illustrated in Figure 3.11(c) with filled and empty squares. For now, the large dots (under new) in this figure can be ignored. Remember that production and consumption sequences are read from left to right. The following iteration period of a static schedule is constructed:

- P acquires 2 containers and releases them to the consuming task at the end of its execution, resulting in 2 completed containers.
- C can execute. It acquires 2 containers ($r_C^1(0) = 2$) but only releases 1 container. The other container remains available on the channel.
- C can no longer execute (it needs 2 completed containers), so P acquires 2 containers and releases them to the consuming task at the end of its invocation, resulting in 3 completed containers on the channel.
- C can execute again. It reuses one container from its first invocation and additionally acquires a second container. At the end, C releases a single container, leaving 2 completed containers on the edge.
- C executes a last time, reusing one container and additionally acquiring a second one. At the end, C releases both containers. This empties the channel and brings the system back in its starting position.



(a) Special channel

(b) CSDF equivalent

Firing	New	Buffer	Reuse
P	••		-
C			1
P	••		-
C	•		1
C			0

(c) Buffer occupancy after firings of P and C

Figure 3.11: Example non-destructive read special channel.

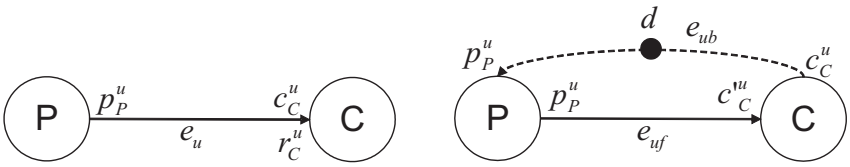
Generally, an edge e_u with non-destructive reads (Figure 3.12(a)) allows a consuming task C to acquire $r_C^u(j)$ containers during its $(j+1)^{th}$ invocation of which only $c_C^u(j)$ containers are released, with

$$\forall j \in \mathbb{N} : r_C^u(j) \geq c_C^u(j) \tag{3.13}$$

This special channel enables data reuse: the same container is accessed over multiple invocations of the same task. Because this container remains available on the special channel, the number of acquired containers $r_C^u(j)$ consists of a number of reused containers and a number of additionally acquired containers. Note that during the first task invocation, all acquired containers are additionally acquired containers.

The number of containers $r(j)$ that is reused from the current invocation j during the next task execution $j+1$ is obtained with Equation (3.14) as the difference between the number of acquired containers and the number of released containers. When the number of acquired containers $r_C^u(j)$ is smaller than the number of reused containers $r(j-1)$ from the previous invocation, this equation calculates $r(j)$ recursively.

$$r(j) = \begin{cases} r_C^u(0) - c_C^u(0) & \text{if } j = 0 \\ r_C^u(j) - c_C^u(j) & \text{if } j > 0 \text{ and } r_C^u(j) > r(j-1) \\ r(j-1) - c_C^u(j) & \text{otherwise.} \end{cases} \tag{3.14}$$



(a) Special channel (b) CSDF equivalent

Figure 3.12: Non-destructive reads between a producer P with period L_P and production sequence $p = \{p_P^u(0), \ldots, p_P^u(L_P-1)\}$ and a consumer C with period L_C and sequences $r = \{r_C^u(0), \ldots, r_C^u(L_C-1)\}$ and $c = \{c_C^u(0), \ldots, c_C^u(L_C-1)\}$ for which $c_C^u(j) \leq r_C^u(j)$.

To avoid an accumulation of containers in the channel that would lead to unbounded memory requirements, the sum of additionally acquired containers during a repetition of the task should equal the number of released containers (bounded memory condition of Equation (3.9)). This requires that the number of reused containers of the last firing of the repetition (q_C) is zero. Consequently, at least all reused containers $r(q_C-2)$ of the one but last firing of the repetition should be acquired, and all acquired containers need to be released:

$$r_C^u(q_C - 1) = {c_C}^u(q_C - 1) \geq r(q_C - 2) \tag{3.15}$$

Proof of Equation (3.15):

In order to prove Equation (3.15), both cases of Equation (3.14) for $j = (q_C - 1) > 0$ are considered while requiring that $r(q_C - 1) = 0$

1. When $r_C^u(q_C - 1) > r(q_C - 2)$ with $r(q_C - 1) = 0$ in Equation (3.14)

$$c_C^u(q_C - 1) = r_C^u(q_C - 1)$$

2. When $r_C^u(q_C - 1) \leq r(q_C - 2)$ with $r(q_C - 1) = 0$ in Equation (3.14)

$$c_C^u(q_C - 1) = r(q_C - 2)$$

Combining this with the Equation (3.13)

$$r_C^u(q_C - 1) \leq c_C^u(q_C - 1) \text{ and } r_C^u(q_C - 1) \geq c_C^u(q_C - 1) \Rightarrow$$

$$r_C^u(q_C - 1) = c_C^u(q_C - 1)$$

Overall

$$r_C^u(q_C - 1) = {c_C}^u(q_C - 1) \geq r(q_C - 2)$$

Q.E.D.

The above condition on the last firing of the repetition also applies to the last firing of the actor period, or

$$r_C^u(L_C - 1) = {c_C}^u(L_C - 1) \geq r(L_C - 2) \tag{3.16}$$

This condition can sometimes be met by setting the actor period appropriately. In video processing for instance, extending the actor period from a row basis to a frame basis allows the correct releasing of all reused containers at the frame border, when no data reuse dependencies exist between frames.

Figure 3.12(b) shows how this data reuse behavior is expressed in CSDF using the decoupling of tokens and containers. Only containers that are no longer reused are released as indicated by the production on the feedback edge e_{ub}

$$p_C^{ub} = c_C^u \tag{3.17}$$

The forward edge e_{uf} assures the correct synchronization between the actors P and C. The number of additionally acquired containers, i.e. the required

number of new completed containers, $c_C^{uf} = c_C'^{u}$ is calculated in Equation (3.18) so that actor C can only start firing j if the sum of reused containers $r(j-1)$ and additionally acquired containers $c_C'^{u}(j-1)$ equals at least $r_C^u(j)$.

$$c_C^{uf} = c_C'^{u}(j) = \begin{cases} r_C^u(0) & \text{if } j = 0 \\ r_C^u(j) - r(j-1) & \text{if } j > 0 \text{ and } r_C^u(j) > r(j-1) \\ 0 & \text{otherwise.} \end{cases} \tag{3.18}$$

At the producing side, this special channel behaves like a regular channel, or

$$\forall i \in \mathbb{N} : p_P^{uf}(i) = c_P^{ub}(i) \tag{3.19}$$

Applying this to the simple example (Figure 3.11) yields the CSDF equivalent of Figure 3.11(b). Using Equation (3.18) c_C^{1f} equals $\{2, 1, 1\}$. p_C^{1b} is the consumption sequence of the special channel: $\{1, 1, 2\}$. The large dots of Figure 3.11(c) indicate completed containers, while the empty squares represent free containers. The number of initial tokens on e_{1b} represents the number of free containers at the start. Again, one iteration period of a static schedule is constructed:

- P consumes 2 tokens from e_{1b}, representing the acquiring of 2 containers at the start of its firing and produces 2 tokens on e_{1f} representing the releasing to the consumer of the 2 completed containers (shown as the 2 large dots).
- C can fire. It consumes 2 tokens from e_{1f} but only produces one token on e_{1b}, modeling the releasing of a single container that becomes free. The other container remains available on the channel.
- C can no longer execute (no tokens on e_{1f}), so P fires and consumes 2 tokens from e_{1b}, representing the acquiring 2 containers at the start of its firing and produces 2 tokens on e_{1f} representing the releasing to the consumer of the 2 completed containers (shown as the 2 large dots). The buffer of the channel is now completely occupied.
- C can fire again and can reuses one container from its first firing. It only consumes a single token from e_{1f} representing the additional acquiring of one container. In total, C can access the data from 2 containers. At the end, C produces one token on e_{1b}, modeling the release of a single container. One container remains available for reuse.
- C fires a last time, reusing one container. It only consumes a single token from e_{1f} representing the additional acquiring of one container. In

total, C can access the data from 2 containers. At the end, C produces two tokens on e_{1b}, modeling the release of both containers. This marks all containers of the channel as free and brings the system back in its starting position.

To verify the consistency and race conflict freeness of a general non-destructive read special channel, the conditions of Subsection 3.2.2 are evaluated. Of this bounded memory, mutual exclusiveness and data preservation conditions (Equations (3.9), (3.10), (3.11)), only those at the consumer side need to be checked. The ones at the producer side are automatically fulfilled as $p_P^{uf} = c_P^{ub}$ (since the producer behaves like a regular channel, see Equation (3.19)).

Proof of the requirements in Equations (3.11) and (3.9):
To check the data preservation condition of Equation (3.11), it is first adapted to the non-destructive read special channel, using $c_C^{uf} = c_C^{'u}(j)$ of Equation (3.18) and $p_C^{ub} = c_C^u$ of Equation (3.19)

$$\forall l \in \mathbb{N}_0 : P_C^{ub}(l) \leq C_C^{uf}(l) \Rightarrow C_C^u(l) \leq C_C^{'u}(l)$$

The reasoning behind the proof is the following: starting from the most recent firing $j = l - 1$, $c_C^{'u}(j)$ is substituted using Equation (3.18) and $r(j)$ is replaced using Equation (3.14). This is done repeatedly for decreasing j until the first firing ($j = 0$) is reached. In order to use Equation (3.18), its 3 cases are considered:

1. $j = 0$, after a single firing ($l = 1$)
   Evaluating Equation (3.2) and (3.18) after a single firing: $C_C^u(1) = c_C^u(0)$ and $C_C^{'u}(1) = c_C^{'u}(0) = r_C^u(0)$
   Because of Equation (3.13): $c_C^u(0) \leq r_C^u(0)$
2. $r_C^u(l-1) > r(l-2)$, $l > 1$
   $$C_C^u(l) \leq C_C^{'u}(l)$$
   Using Equation (3.2): $C_C^{'u}(l) = \sum_{j=0}^{l-1} c_C^{'u}(j) = \sum_{j=0}^{l-2} c_C^{'u}(j) + c_C^{'u}(l-1) = C_C^{'u}(l-1) + c_C^{'u}(l-1)$
   $$C_C^u(l) \leq C_C^{'u}(l-1) + c_C^{'u}(l-1)$$
   Using Equation (3.18): $c_C^{'u}(l-1) = r_C^u(l-1) - r(l-2)$
   $$C_C^u(l) \leq C_C^{'u}(l-1) + r_C^u(l-1) - r(l-2)$$
   Since Equation (3.14) calculates $r(j)$ recursively when $r_C^u(j) \leq r(j-1)$, $r(l-2)$ can be rewritten as: $r(l-2) = r_C^u(l-x) - \sum_{j=2}^{x} c_C^u(l-j)$, with $x \geq 2$ indicating the most recent firing $l-x$ for which $r_C^u(l-x) > r(l-x-1)$ or $r_C^u(j) \leq r(j-1)$ for $l-x < j < l-1$.
   $$C_C^u(l) \leq C_C^{'u}(l-1) + r_C^u(l-1) - r_C^u(l-x) + \sum_{j=2}^{x} c_C^u(l-j)$$

Using Equation (3.18) shows $c_C^{'u}(j) = 0$ when $r_C^u(j) \leq r(j-1)$ for $l-x < j < l-1$ and $x \geq 2$. As a result: $C_C^{'u}(l-1) = \sum_{j=0}^{l-2} c_C^{'u}(j) = \sum_{j=0}^{l-x} c_C^{'u}(j) + \sum_{j=l-x+1}^{l-2} c_C^{'u}(j) = C_C^{'u}(l-x+1)$

$$C_C^u(l) \leq C_C^{'u}(l-x+1) + r_C^u(l-1) - r_C^u(l-x) + \sum_{j=2}^{x} c_C^u(l-j)$$

Using Equation (3.2): $C_C^u(l) = \sum_{j=0}^{l-1} c_C^u(j) = \sum_{j=0}^{l-x-1} c_C^u(j) + \sum_{j=l-x}^{l-2} c_C^u(j) + c_C^u(l-1) = C_C^u(l-x) + \sum_{j=l-x}^{l-2} c_C^u(j) + c_C^u(l-1)$

$$C_C^u(l-x) + \sum_{j=l-x}^{l-2} c_C^u(j) + c_C^u(l-1) \leq C_C^{'u}(l-x+1) + r_C^u(l-1) - r_C^u(l-x) + \sum_{j=2}^{x} c_C^u(l-j)$$

Because $\sum_{j=l-x}^{l-2} c_C^u(j) = \sum_{j=2}^{x} c_C^u(l-j)$

$$C_C^u(l-x) + c_C^u(l-1) \leq C_C^{'u}(l-x+1) + r_C^u(l-1) - r_C^u(l-x)$$

With $C_C^{'u}(l-x+1) = C_C^{'u}(l-x) + c_C^{'u}(l-x)$ and using equation 3.18 for firing $l-x$ for which $r_C^u(l-x) > r(l-x-1)$: $c_C^{'u}(l-x) = r_C^u(l-x) - r(l-x-1)$

$$C_C^u(l-x) + c_C^u(l-1) \leq C_C^{'u}(l-x) + r_C^u(l-x) - r(l-x-1) + r_C^u(l-1) - r_C^u(l-x)$$

$$C_C^u(l-x) + c_C^u(l-1) \leq C_C^{'u}(l-x) + r_C^u(l-1) - r(l-x-1) \tag{3.20}$$

Again, since Equation (3.14) calculates $r(j)$ recursively when $r_C^u(j) \leq r(j-1)$, $r(l-x-1)$ can be rewritten as: $r(l-x-1) = r_C^u(l-y) - \sum_{j=x+1}^{y} c_C^u(l-j)$, with $y \geq x+1$ indicating the most recent firing $l-y$ for which $r_C^u(l-y) > r(l-y-1)$ or $r_C^u(j) \leq r(j-1)$ for $l-y < j < l-x-1$.

$$C_C^u(l-x) + c_C^u(l-1) \leq C_C^{'u}(l-x) + r_C^u(l-1) - r_C^u(l-y) + \sum_{j=x+1}^{y} c_C^u(l-j)$$

Again, using Equation (3.18) shows $c_C^{'u}(j) = 0$ when $r_C^u(j) \leq r(j-1)$ for $l-y < j < l-x-1$ and $y \geq x+1$. As a result: $C_C^{'u}(l-x) = C_C^{'u}(l-y+1)$

$$C_C^u(l-x) + c_C^u(l-1) \leq C_C^{'u}(l-y+1) + r_C^u(l-1) - r_C^u(l-y) + \sum_{j=x+1}^{y} c_C^u(l-j)$$

Because $C_C^u(l-x) = C_C^u(l-y) + \sum_{j=l-y}^{l-x-1} c_C^u(j)$ and $\sum_{j=l-y}^{l-x-1} c_C^u(j) = \sum_{j=x+1}^{y} c_C^u(l-j)$

$$C_C^u(l-y) + c_C^u(l-1) \leq C_C^{'u}(l-y+1) + r_C^u(l-1) - r_C^u(l-y)$$

Again, with $C_C^{'u}(l-y+1) = C_C^{'u}(l-y) + c_C^{'u}(l-y)$ and using equation 3.18 for firing $l-y$ for which $r_C^u(l-y) > r(l-y-1)$: $c_C^{'u}(l-y) = r_C^u(l-y) - r(l-y-1)$

$$C_C^u(l-y) + c_C^u(l-1) \leq C_C^{'u}(l-y) + r_C^u(l-1) - r(l-y-1) \tag{3.21}$$

Since Equation (3.21) equals Equation (3.20) when y is replaced by x, $l-y-1$ will equal zero after a certain amount of iterations. Setting $l-y-1=0$

$$C_C^u(1)+c_C^u(l-1)\leq C_C^{'u}(1)+r_C^u(l-1)-r(0)$$

$$c_C^u(0)+c_C^u(l-1)\leq c_C^{'u}(0)+r_C^u(l-1)-r(0)$$

With $r(0)=r_C^u(0)-c_C^u(0)$ (see Equation (3.14))

$$c_C^u(0)+c_C^u(l-1)\leq r_C^u(0)+r_C^u(l-1)-(r_C^u(0)-c_C^u(0))$$

$$c_C^u(l-1)\leq r_C^u(l-1)$$

This is true because of Equation (3.13)

3. $r_C^u(l-1)\leq r(l-2)$

$$C_C^u(l)\leq C_C^{'u}(l)$$

When $x\geq 2$ indicates the most recent firing $l-x$ for which $r_C^u(l-x)>r(l-x-1)$ or $r_C^u(j)\leq r(j-1)$ for $l-x<j<l-1$, then according to Equation (3.18) $c_C^{'u}(j)=0$ for $l-x<j<l-1$ making $C_C^{'u}(l)=C_C^{'u}(l-x+1)$

$$C_C^u(l)\leq C_C^{'u}(l-x+1)$$

Using Equation (3.2): $C_C^{'u}(l-x+1)=\sum_{j=0}^{l-x}c_C^{'u}(j)=\sum_{j=0}^{l-x-1}c_C^{'u}(j)+c_C^{'u}(l-x)=C_C^{'u}(l-x)+c_C^{'u}(l-x)$

$$C_C^u(l)\leq C_C^{'u}(l-x)+c_C^{'u}(l-x)$$

Using Equation (3.18) $c_C^{'u}(l-x)=r_C^u(l-x)-r(l-x-1)$

$$C_C^u(l)\leq C_C^{'u}(l-x)+r_C^u(l-x)-r(l-x-1)$$

Since Equation (3.14) calculates $r(j)$ recursively when $r_C^u(j)\leq r(j-1)$, $r(l-1)$ can be rewritten as: $r(l-1)=r_C^u(l-x)-\sum_{j=2}^{x}c_C^u(l-j)$, with $r_C^u(j)\leq r(j-1)$ for $l-x<j<l-1$. This makes $r_C^u(l-x)=r(l-1)+\sum_{j=1}^{x}c_C^u(l-j)$

$$C_C^u(l)\leq C_C^{'u}(l-x)+r(l-1)+\sum_{j=1}^{x}c_C^u(l-j)-r(l-x-1)$$

Using Equation (3.2): $C_C^u(l)=\sum_{j=0}^{l-1}c_C^u(j)=\sum_{j=0}^{l-x-1}c_C^u(j)+\sum_{j=l-x}^{l-1}c_C^u(j)=C_C^u(l-x)+\sum_{j=l-x}^{l-1}c_C^u(j)$

$$C_C^u(l-x)+\sum_{j=l-x}^{l-1}c_C^u(j)\leq C_C^{'u}(l-x)+r(l-1)+\sum_{j=1}^{x}c_C^u(l-j)-r(l-x-1)$$

Because $\sum_{j=l-x}^{l-1}c_C^u(j)=\sum_{j=1}^{x}c_C^u(l-j)$

$$C_C^u(l-x)\leq C_C^{'u}(l-x)+r(l-1)-r(l-x-1) \tag{3.22}$$

Again, when $y \geq x+1$ indicates the most recent firing $l-y$ for which $r_C^u(l-y) > r(l-y-1)$ or $r_C^u(j) \leq r(j-1)$ for $l-y < j < l-x-1$, then according to Equation (3.18) $c_C^{'u}(j) = 0$ for $l-y < j < l-x-1$ making $C_C^{'u}(l-x) = C_C^{'u}(l-y+1)$

$$C_C^u(l-x) \leq C_C^{'u}(l-y+1) + r(l-1) - r(l-x-1)$$

Again, using Equation (3.2): $C_C^{'u}(l-y+1) = C_C^{'u}(l-y) + c_C^{'u}(l-y)$

$$C_C^u(l-x) \leq C_C^{'u}(l-y) + c_C^{'u}(l-y) + r(l-1) - r(l-x-1)$$

Again, using Equation (3.18) $c_C^{'u}(l-y) = r_C^u(l-y) - r(l-y-1)$

$$C_C^u(l-x) \leq C_C^{'u}(l-y) + r_C^u(l-y) - r(l-y-1) + r(l-1) - r(l-x-1)$$

Again, since Equation (3.14) calculates $r(j)$ recursively when $r_C^u(j) \leq r(j-1)$, $r(l-x-1)$ can be rewritten as: $r(l-x-1) = r_C^u(l-y) - \sum_{j=x-1}^{y} c_C^u(l-j)$, with $r_C^u(j) \leq r(j-1)$ for $l-y < j < l-x-1$. This makes $r_C^u(l-y) = r(l-x-1) + \sum_{j=x-1}^{y} c_C^u(l-j)$

$$C_C^u(l-x) \leq C_C^{'u}(l-y) + r(l-x-1) + \sum_{j=x-1}^{y} c_C^u(l-j) - r(l-y-1) + r(l-1) - r(l-x-1)$$

$$C_C^u(l-x) \leq C_C^{'u}(l-y) + \sum_{j=x-1}^{y} c_C^u(l-j) - r(l-y-1) + r(l-1)$$

Again, using Equation (3.2): $C_C^u(l-x) = C_C^u(l-y) + \sum_{j=l-y}^{l-x-1} c_C^u(j)$

$$C_C^u(l-y) + \sum_{j=l-y}^{l-x-1} c_C^u(j) \leq C_C^{'u}(l-y) + \sum_{j=x-1}^{y} c_C^u(l-j) - r(l-y-1) + r(l-1)$$

Because $\sum_{j=l-y}^{l-x-1} c_C^u(j) = \sum_{j=x-1}^{y} c_C^u(l-j)$

$$C_C^u(l-y) \leq C_C^{'u}(l-y) + r(l-1) - r(l-y-1) \tag{3.23}$$

Since Equation (3.23) equals Equation (3.22) when y is replaced by x, $l-y-1$ equals zero after a certain amount of iterations. Setting $l-y-1 = 0$

$$C_C^u(1) \leq C_C^{'u}(1) + r(l-1) - r(0)$$

$$c_C^u(0) \leq c_C^{'u}(0) + r(l-1) - r(0)$$

With $c_C^{'u}(0) = r_C^u(0)$ (see Equation (3.18))

$$c_C^u(0) \leq r_C^u(0) + r(l-1) - r(0)$$

With $r(0) = r_C^u(0) - c_C^u(0)$ (see Equation (3.14))

$$0 \leq r(l-1)$$

To check the bounded memory condition of Equation (3.9), L_C firings are considered or $l = L_C$.

$$C_C^u(L_C) = C_C'^u(L_C)$$

Because of Equation (3.16), $r_C^u(L_C - 1) \geq r(L_C - 2)$. This matches the second case of the proof above. Substituting l by L_C and replacing the inequality by an equality yields

$$c_C^u(L_C - 1) = r_C^u(L_C - 1)$$

This is true because of Equation (3.16).

Q.E.D.

3.3.2 Partial Update

A partial update special channel supports the production of containers over multiple invocations. This feature is for instance useful when the producing task works on a block-basis while the consuming task works on a macroblock basis and expects the data of the macroblock in a different order than just four consecutive blocks. Note that waiting for all blocks of the macroblock and then reordering them would invoke more memory accesses. This is in contradiction with the goal of the special channels: to represent optimized communication. Figure 3.13 shows a specific video example where the texture updates works on a block-basis and writes its results in a line scan organized frame memory. Using partial updates with a line of pixels as a container, the texture update can produces its output in the correct order in the frame memory. Only when all pixels of a line are produced over multiple invocations, the container is released.

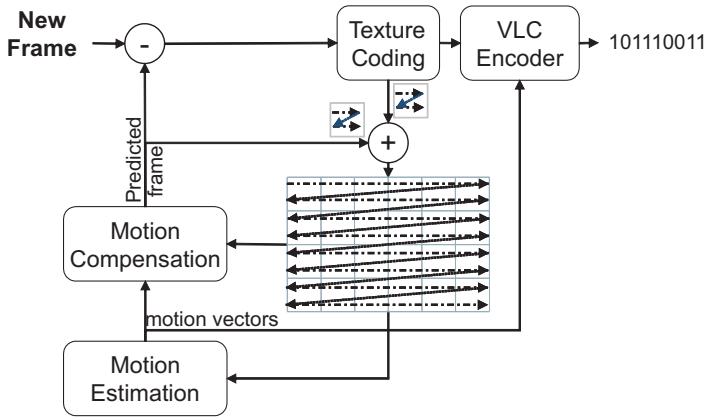
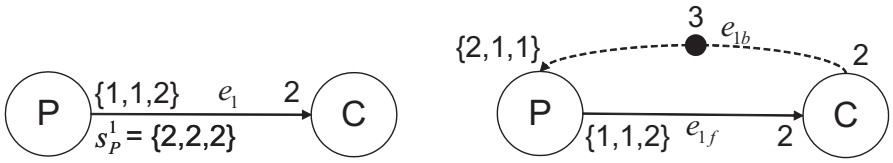


Figure 3.13: The texture update, represented by the addition, uses partial updates to write the reconstructed data in the line scan organized frame memory.

A simple example of a partial update special channel is shown in Figure 3.14(a). $s_P^1 = \{2, 2, 2\}$ indicates that P will write in two containers, while its production sequence $\{1, 1, 2\}$ only releases both of them at the end. The buffer occupancy is illustrated in Figure 3.14(c). The empty squares represent free containers, the full squares indicate completed containers while the diagonally shaded ones are uncompleted containers. For now, the large dots (under new) in this figure can be ignored. The following iteration period of a static schedule is constructed:



(a) Special channel (b) CSDF equivalent

Firing	New	Buffer	Partial
P	•		1
P	••		1
C			-
P	••		0
C			-

(c) CSDF equivalent

Figure 3.14: Example partial update special channel.

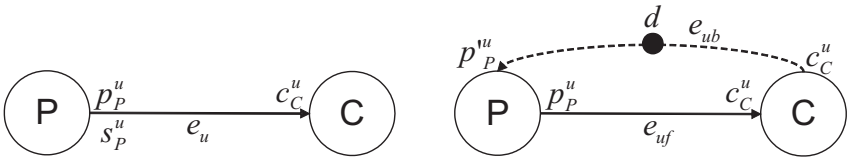
- P acquires 2 containers and releases only one of them to the consuming task at the end of its execution. This results in one completed container and one uncompleted container.
- C can not yet execute, so a second invocation of P starts. P continues with the uncompleted container and additionally acquires a second container as P writes to 2 containers ($s_P^1(1) = 2$). At the end of its execution, P releases only one container to the consuming task. This results in 2 completed containers on the channel and one uncompleted one.
- C can now execute. It acquires 2 containers and releases both at the end, marking them as free containers.
- P has to execute (no completed containers available anymore). P continues with the uncompleted container and additionally acquires a second container. At the end of its execution, P releases both containers to the consuming task. This results in 2 completed containers.

- C can execute again. It acquires 2 containers and releases both at the end, marking them as free containers. This brings the system back in its starting position.

Generally, an edge e_u with partial updates (Figure 3.15(a)) allows the acquiring of $s_P^u(i)$ containers at the producer side during invocation i of which only $p_P^u(i)$ containers are full and released at the end of the task execution, with

$$\forall i \in \mathbb{N} : s_P^u(i) \geq p_P^u(i) \tag{3.24}$$

This enables the production of data in a container over multiple invocations. Because this container remains available on the special channel, the number of acquired containers $s_P^u(i)$ consists of a number of uncompleted containers and a number of additionally acquired containers. Note that during the first task invocation, all acquired containers are additionally acquired containers.



(a) Special channel (b) CSDF equivalent

Figure 3.15: Partial updates between a producer P with period L_P and sequences $p = \{p_P^u(0), \ldots, p_P^u(L_P-1)\}$ and $s = \{s_P^u(0), \ldots, s_P^u(L_P-1)\}$ for which $p_P^u(i) \leq s_P^u(i)$ and a consumer C with period L_C and sequence $c = \{c_C^u(0), \ldots, c_C^u(L_C-1)\}$.

The number of uncompleted containers $s(i)$ in task invocation i that are continued during the next invocation $i+1$ is calculated with Equation (3.25) as the difference between the number of acquired containers and the number of completed containers. When the number of acquired containers $s_P^u(i)$ is smaller than the number of reused containers $s(i-1)$ from the previous invocation, this equation calculates $s(i)$ recursively.

$$s(i) = \begin{cases} s_P^u(0) - p_P^u(0) & \text{if } i = 0 \\ s_P^u(i) - p_P^u(i) & \text{if } i > 0 \text{ and } s_P^u(i) > s(i-1) \\ s(i-1) - p_P^u(i) & \text{otherwise.} \end{cases} \tag{3.25}$$

To avoid the loss of partially produced data, the number of containers acquired during the last invocation has to include the remaining uncompleted ones from the previous executions(s) (calculated with Equation (3.25)) and all of them need to be released

$$s_P^u(n-1) = p_P^u(n-1) \geq s(n-2) \tag{3.26}$$

Similar to the non-destructive read, this condition can sometimes be met by setting the actor period appropriately. If this is not possible, the channel is misused as scratchpad. Such temporal data should be stored in a local buffer of the task. An example of misuse could be a motion estimation task acquiring two containers, one to produce minimum Sum of Absolute Differences (SAD) of the best match and one to store the intermediate result of the SAD calculation during the search. As at the end of the invocation, only the minimum SAD is released and never the intermediate result, the latter one should be stored in a local buffer.

The partial update behavior is represented in Figure 3.15(b) using the decoupling of tokens and containers. Only the completed containers are released to be used by the consumer, as indicated by the production $p_P^{uf} = p_P^u$ on the forward edge e_{uf}. Consequently, this edge e_{uf} synchronizes the producer and the consumer. Equation (3.27) makes sure that the sum of uncompleted containers $s(i-1)$ and additionally acquired containers $p_P^{ub} = p_P^{'u}(i)$ at least equals the number of acquired containers $s_P^u(i)$ for data production during firing i.

$$c_P^{ub} = p_P^{'u} = \begin{cases} s_P^u(0) & \text{if } i = 0 \\ s_P^u(i) - s(i-1) & \text{if } i > 0 \text{ and } s_P^u(j) > s(i-1) \\ 0 & \text{otherwise.} \end{cases} \tag{3.27}$$

At the consuming side, this special channel behaves like a regular channel, or

$$\forall j \in \mathbb{N} : c_C^{uf}(j) = p_C^{ub}(j) \tag{3.28}$$

Applying this to the simple example (Figure 3.14) yields the CSDF equivalent of Figure 3.14(b). Using Equation (3.27) c_P^{1b} equals $\{2, 1, 1\}$. p_P^{1f} is the production sequence of the special channel: $\{1, 1, 2\}$. The large dots of Figure 3.11(c) indicate completed containers, while the empty squares represent the free containers. The number of initial tokens on e_{1b} represent the number of free containers at the start. Again, one iteration period of a static schedule is constructed:

- P consumes 2 tokens from e_{1b} modeling the acquiring of 2 containers and produces only a single token on e_{1f} representing the releasing of a completed container to the consumer (shown as a large dot). One uncompleted container remains available on the channel.
- C can not yet fire, so a second firing of P starts. P continues with the uncompleted container and consumes one token from e_{1b} representing the additional acquire of one container. In total, P can access 2 containers. At the end of its firing, P produces only one token on e_{1f} modeling a completed container. This results in 2 completed containers on the channel and one uncompleted one.

- C can now fire. It consumes 2 tokens from e_{1f} and produces 2 tokens on e_{1b}, marking 2 containers as free.
- P has to fire (no completed containers available any more). P continues with the uncompleted container and consumes one token from e_{1b} representing the additional acquire of one container. In total, P can access 2 containers. At the end of its firing, P produces 2 tokens on e_{1f} modeling the release of both containers to the consumer. This results in 2 completed containers.
- C can execute again. It consumes 2 tokens from e_{1f} and produces 2 tokens on e_{1b}, marking 2 containers as free. This brings the system back in its starting position.

To verify the consistency and race conflict freeness of a general partial update special channel, the conditions of Subsection 3.2.2 are evaluated. Of these bounded memory, mutual exclusiveness and data preservation conditions (Equations (3.9), (3.10), (3.11)), only the ones at the producer need to be checked. The conditions at the consumer are automatically fulfilled as $c_C^{uf} = p_C^{ub}$ (see Equation (3.28)). The proof is similar to the non-destructive read one.

3.3.3 Multiple Consumers

A multiple consumer special channel supports the acquiring of the same completed containers by multiple tasks. In this way, a buffer is shared between them. In a hybrid video encoder, both the motion estimation and the motion compensation need data from the reconstructed frame. In Figure 3.16 the dotted region is the search area read by the motion estimation while the dashed area is fetched by the motion compensation based on the motion vectors.

Generally, an edge e_u with multiple consumers (Figure 3.17(a)) allows N consuming tasks $C1 \dots CN$ to consume the same containers produced by a task P. Each consumer Cy can have its own actor period L_{Cy} as long as a solution exists for their combined balance equations in Equation (3.29) to obey the consistency condition.

$$r_P \cdot P_P^u(L_P) = \begin{cases} r_{C1} \cdot C_{C1}^u(L_{C1}) \\ \vdots \\ r_{CN} \cdot C_{CN}^u(L_{CN}) \end{cases} \tag{3.29}$$

A multiple consumer edge works with a *composed consumption* representing the number of containers that can be released at the consume side on condition

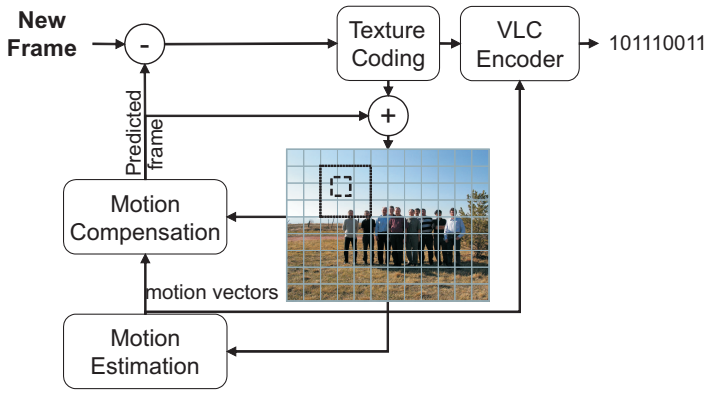
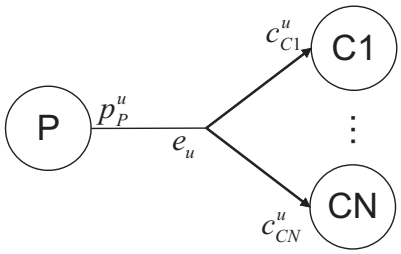
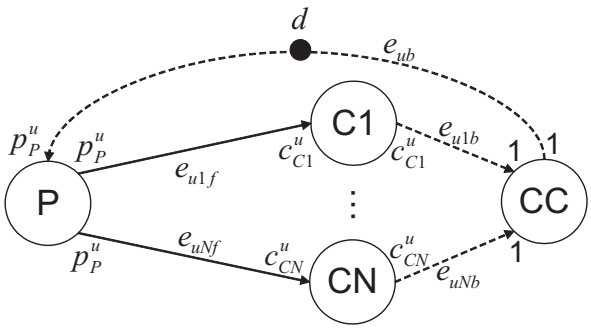


Figure 3.16: The motion estimation reads data in the dotted search area while the motion compensation reads from the dashed area as indicated by the motion vectors.



(a) Special channel

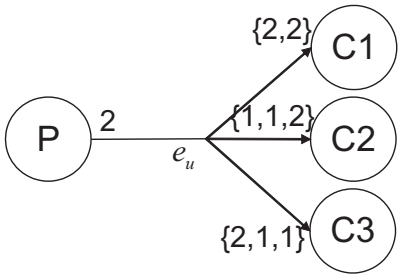


(b) CSDF equivalent

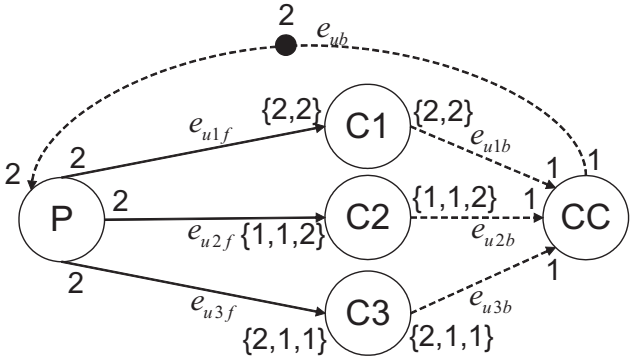
Figure 3.17: Multiple consumers on an edge between a producer P with period L_P and sequence $p = \{p_P^u(0), \ldots, p_P^u(L_P - 1)\}$ and N consumers $C1, \ldots, CN$ with periods $L_{C1}, \ldots, L_{CN}$ and sequences $c1 = \{c_{C1}^u(0), \ldots, c_{C1}^u(L_{C1} - 1)\}, \ldots, cN = \{c_{CN}^u(0), \ldots, c_{CN}^u(L_{CN} - 1)\}$.

that all actors $C1 \dots CN$ have released it. Equation (3.30) calculates the composed consumption $cc^u(j_c)$ after l_y firings of the tasks Cy (with $1 \leq y \leq N$). The index j_c counts the composed consumptions by incrementing j_c whenever a consuming task Cy executes. To make sure all consumers no longer need the container(s), this equation looks for the consuming task with the minimum sum of consumed containers and subtracts the sum of previously composed consumed containers.

$$cc^u(j_c) = \min_{1 \leq y \leq N} \left(C^u_{Cy}(l_y) \right) - C^u_{cc}(j_c), \text{ with } j_c = \left(\sum_{y=1}^{N} l_y \right) - 1 \tag{3.30}$$



(a) Special channel



(b) CSDF equivalent

Figure 3.18: Example multiple consumers special channel.

Figure 3.18(a) presents a simple example of a special channel with 3 consumers. The combined balance equation is easily met as all actors produce or consume 4 containers during one actor period. The calculation of the composed consumption $cc^u(j_c)$ is illustrated in Table 3.1 listing the sum of produced tokens and the sum of consumed tokens of each actor. After an initial firing of P, $C1$ is the first consumer to execute (row 2 of the table). The index of the composed consumption j_c becomes 0, $cc^u(0)$ is also zero as $C2$ and $C3$ still

need the container. Only when $C3$ has also fired once ($j_c = 3$), two containers can be released ($cc^u(j_c)$).

Table 3.1: Calculation of the composed consumption of the example with 3 consumers.

Actor	$P_P^u(k)$	$C_{C1}^u(l_1)$	$C_{C2}^u(l_2)$	$C_{C3}^u(l_3)$	j_c	$C_{cc}^u(j_c+1)$	$cc^u(j_c)$
P	2	0	0	0	-	-	-
C1	2	2	0	0	0	0	0
C2	2	2	1	0	1	0	0
C2	2	2	2	0	2	0	0
C3	2	2	2	2	3	2	2
P	4	2	2	2	-	2	-
C1	4	4	2	2	4	2	0
C2	4	4	4	2	5	2	0
C3	4	4	4	3	6	3	1
C3	4	4	4	4	7	4	1

Such a multiple consumer edge is represented in CSDF using the decoupling of tokens and containers in Figure 3.17(b). On each of the N forward edges e_{uyf} the same number of tokens p_P^u representing the available completed containers is produced during a firing of the producer. The number of tokens consumed from these forward edges can vary for the N consumers, including the consume sequence length, as long as the balance condition of Equation (3.29) is met. The composed consumption is modeled by the CC actor with a zero response time. Only when all consuming actors have released a container, it is made available as free container on the backward edge e_{ub}.

As the size of the container buffer d is shared over all edges, the number of free containers f (in the shared buffer) equals the number of initially free containers d decreased with the number of acquired containers after k firings of the producer and incremented with the number of composed consumed containers after l_c composed consumptions.

$$f = d - C_P^{ub}(k) + C_{CC}^u(l_c) \tag{3.31}$$

Using equation 3.30 and $c_C^{uyf} = p_C^{uyb}$, $C_{CC}^u(l_c)$ can be rewritten and the number of free containers f becomes

$$f = d - C_P^{ub}(k) + \min_{1 \le y \le N}(P_{Cy}^{uyb}(l_y)) \tag{3.32}$$

where the minimum over all edges assures the containers remain available until the last consumer has released them.

The bounded memory, mutual exclusiveness conditions (Equations (3.9), (3.10)) of the special channel are met as for all edges $p_P^{uyf} = c_P^{ub}$, $c_C^{uyf} = p_C^{uyb}$ and the

CC actor has all ones as consumption and production rates. The data preservation condition (Equation (3.11)) is satisfied because the composed consumption can only lead to a later releasing of a container that was still needed by another consuming task.

3.3.4 Multiple Producers

A special channel e_u with multiple producers (Figure 3.19) allows N producing tasks $P1 \ldots PN$ to produce containers. A video coding example could be multiple tasks each producing a block (or a set of blocks) in a macroblock. Graphically, this special channel is the symmetrical case of the multiple consumer. From behavioral point of view, two cases need to be considered: (i) each producer acquires and releases its own container(s) and (ii) the producers make partial updates to a shared container. None of these two cases is a symmetrical version of the multiple consumer behavior.

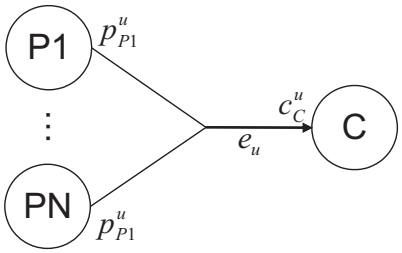


Figure 3.19: The multiple producers special channel with producers $P1, \ldots, PN$ have no CSDF equivalent as the token order depends on the response time.

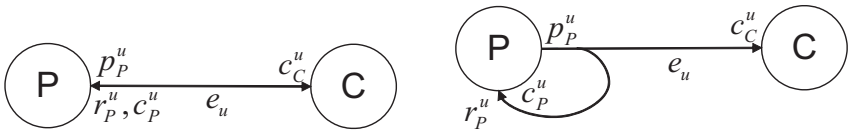
When the producing tasks are working on their own container(s), the order in which the containers are released on the special channel depends on the actual response time of the producers. This non-deterministic behavior of the special channel makes it invalid for representation in CSDF. Note that a specific ordering of the containers can be forced by adding edges between the multiple producers expressing these ordering constraints. In contrast, every consumer of a multiple consumer edge needs to consume all containers of the special channel. The ordering is the strict FIFO ordering on the edge that is maintained (see 3.1.3) by the implicit presence of a self-cycle in all graphs of this section.

Multiple producers with partial updates on a single edge would allow these tasks to write their part of the container. To avoid overwriting data in the shared container, its mutual exclusiveness needs to be guaranteed: spatially or temporally. A possible CSDF equivalent is to use separate edges between the producers and the consumer that only communicate the part of the container that the producer is generating. Separate edges assure the required

mutual exclusiveness. The processing at consumer can become more difficult, as the data needs to be fetched from different channels. Though the advantage towards composing a single container from multiple producers is recognized, this special edge is not further worked out due to its sensitivity to data loss (by overwriting). In contrast, a multiple consumer special edge offers the consumers the freedom to only partially read the containers (while still all of them need to be consumed in FIFO ordering).

3.3.5 Combinations and Generalization

All valid previous special channels can be combined, like an edge with partial updates and non-destructive reads, an edge with partial updates and multiple consumers, etc. An interesting combination is multiple consumers with non-destructive reads as it allows a producing task P to read previously produced containers back (Figure 3.20(a)) by considering the producer also as a consumer on the same special channel (Figure 3.20(b)). In video coding, this occurs for instance when neighboring motion vectors are needed to predict a starting position for the search algorithm.



(a) Special channel

(b) Expressed as multiple consumers with non-destructive reads

Figure 3.20: Special case of the multiple consumers with non-destructive read: a non-destructive read-back at the producer side.

A combination of all valid special channels leads to a general CSDF equivalent for them (see Figure 3.21). All previously presented CSDF equivalents are an instantiation of this general one. When $p_P^{uf}(i) \leq c_P^{ub}(i)$, the producer uses partial updates on the multiple consumer special channel. When $c_{Cy}^{uf}(j) \leq p_{Cy}^{ub}(j)$, consumer Cy uses non-destructive reads. All derived requirements of the special channels (like the combined balance equations, the equality of the number of requested and released containers during the last invocation of the actor period) remain into effect so that the bounded memory, mutual exclusiveness and data preservation conditions (Equations (3.9), (3.10), (3.11)) are fulfilled.

Because CSDF is an extension of SDF, it is interesting to check if the proposed special channels also work for an SDF graph. To meet the restrictions imposed on the non-destructive read and the partial update (expressed respectively in equations 3.15 and 3.26), currently cyclo-static behavior is required. Further

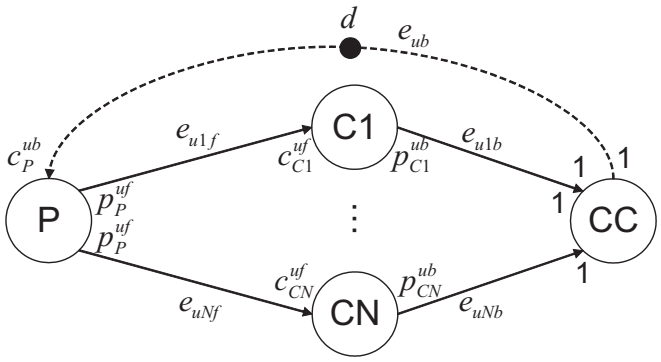


Figure 3.21: General CSDF equivalent of the special edges.

research is needed to investigate the impact of relaxing the constraints and extend these special channels to SDF. The threshold used in the graph chains of [50] look similar to what a non-destructive read would be in SDF. As long as the combined balance equation of the multiple consumer special channel is respected, it could as well be used in a SDF context.

3.3.6 Other Implementation Aspects

All special channels described above represent a synchronizing communication. The implementation of an application can also use non-synchronizing communication, to pass for instance parameters or if synchronization becomes obsolete when tasks never execute concurrently due to ordering constraints. Additionally, memory bandwidth constraints need to be taken into account. The next subsections discuss these implementation aspects. Section 3.6 lists existing (C)SDF work also dealing with these aspects.

3.3.6.1 Global parameters

Global parameters are used in an implementation to pass the most recent settings to a task. Through a global buffer with an updating mechanism, the consuming tasks only see the new parameters when the producer completed the new data in a container. The non-synchronizing behavior of such a communication and its dynamic consumption and production pattern can not be modeled in CSDF. On the other hand, these global parameters do not influence the temporal behavior (since they are a form of non-synchronizing communication) nor need to be considered during the buffer capacity calculation as their size is fixed at design time (depending on the number and the size of the parameters).

Figure 3.22 proposes a notation for such a global buffer. To express the non-synchronizing behavior, the production and consumption amount of the actor is set to zero. To indicate the required buffer space to produce or consume data, partial update and non-destructive read notations are used. Since only the parameters relevant to the consumer need to be written, $s_P^u = r_C^u$. The required buffer size is equal to or a multiple of $s_P^u = r_C^u$.

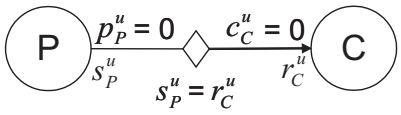


Figure 3.22: Notation of a global buffer.

3.3.6.2 Serialized Actors

In some cases, actors will never fire concurrently due to ordering constraints, either in their schedule or in the graph topology. The schedule ordering constraint can also be represented in the graph by adding an edge to indicate this. In Figure 3.23 actors A, B, C and D can only fire sequentially due to the graph topology. A schedule ordering constraint (for instance a sequential schedule A, B, C, D) of the same graph but without edge e_4 can be represented by adding edge e_4.

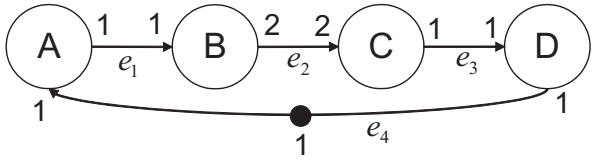


Figure 3.23: Some actors do not fire concurrently due to the schedule or the graph topology.

Using a global buffer allows sharing container space between such serialized actors. This approach is combined in state of the art with lifetime analysis for memory optimized software synthesis [76, 83]. Figure 3.24 uses global buffers to store the data of tokens, while the dotted edges between the actors only represent synchronization. To take into account the implementation modeling aspect where containers are acquired at the start of the invocation but only released at the end, two global buffers are needed.

3.3.6.3 Redundant Synchronization

Reference [100] defines the synchronization of edge e_u in a graph G as redundant if the removal of this edge results in a graph G' that preserves G, i.e.

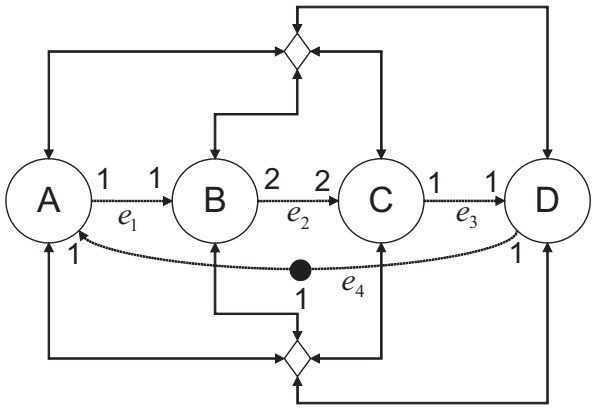
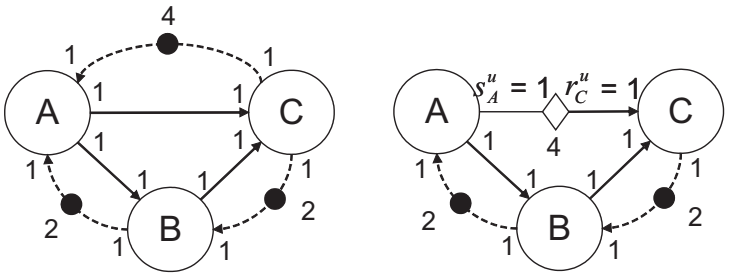


Figure 3.24: Using global buffers between serialized actors allows sharing token space.

all actor dependencies and ordering constraints are unaffected. The order in which such edges are removed is not important. Consequently, all redundant synchronization edges can be removed together [100]. As the redundant edges need no explicit synchronization, they can be implemented as global buffers. Removing redundant synchronization results in a more efficient design as the synchronization cost is reduced.



(a) Graph G with redundant synchronization

(b) Graph G' where the redundant synchronization is removed

Figure 3.25: Actors A and C are synchronized through path (A, C) and path (A, B, C). As both paths have the same delay (expressed as initial tokens), the synchronization of one can be removed.

The principle of redundant synchronization can be applied to a non-destructive read-back at the producer special channel of Figure 3.20(a). The reading back will not influence the presence of containers on this channel, if the following two requirements on c_P^u and r_P^u met:

$$\forall k,l \in \mathbb{N}_0 : \begin{cases} \sum_{i=0}^{k-2} c_P^u(i) \geq \sum_{j=0}^{l-1} c_C^u(j) \\ \sum_{i=0}^{k-2} p_P^u(i) - \sum_{i=0}^{k-2} c_P^u(i) \geq r_P^u(k-1) \end{cases} \tag{3.33}$$

The first requirement expresses that a container that is read back, should not yet have been consumed by C. The second requirement tells that only completed containers of previous firings can be read back. When both requirements are met, the synchronization support on the reading back can be omitted. Equation (3.34) presents a less strict form allowing the producer to read back from the data it writes during the current firing (sum of produced tokens $p_P^u(i)$ includes $(k-1)$). Both conditions are evaluated for a sequential schedule that fires the consuming actor as soon as its firing rule is true. This leads to the shortest availability of the tokens for reading back. Note that the firing of the producer only depends on the available space to produce its tokens.

$$\forall k,l \in \mathbb{N}_0 : \begin{cases} \sum_{i=0}^{k-2} c_P^u(i) \geq \sum_{j=0}^{l-1} c_C^u(j) \\ \sum_{i=0}^{k-1} p_P^u(i) - \sum_{i=0}^{k-2} c_P^u(i) \geq r_P^u(k-1) \end{cases} \tag{3.34}$$

3.3.6.4 Bandwidth Requirements

Another important design aspect is the required bandwith to the memories of the regular and special channels. This bandwidth can be deducted from an analysis of the CSDF model. In case of stream data, where every value of a container is only accessed once during the actor firing, it is related to the RT of the actors. When random access to the data of containers occurs, an analysis of the data dependent access patterns is required. An in depth treatment is a topic of future research and hence outside the scope of this text. It is important to note that the first phase of the design flow reduces the accesses to large memory and favors small and local buffers for high data traffic. In this way, the impact of the often higher latency of larger memories on the final throughput is controlled.

3.4 Buffer Capacity Calculation

The (minimum) buffer capacities d are calculated by manually constructing a (desired) static periodic schedule and combining this with a life-time analysis of the tokens using the worst-case actor response times. The schedule needs to cover at least a complete iteration period in the periodic phase. As a result, it is constructed from the start and also includes the transient phase before reaching the periodic phase. As no dead-lock is allowed in this periodic

schedule to assure the liveness of the graph, the minimum buffer size is found if the number of free tokens f on the feedback edge is zero (i.e. $f = 0$), when the difference between the total number of consumed and produced tokens on this edge reaches a maximum. The buffer capacity d_u of edge e_u is derived from Equation (3.32), the generic case for the all valid special channels, by setting f to zero and considering the life-time analysis from start until one period in steady state (periodic phase) is completed. Assuming the desired schedule reaches the periodic phase after k_{SS} firings of the producer P and $l_{y,SS}$ firings of the consumers Cy

$$d_u = \max_{0 \leq k < k_{SS}+q_P^b; 0 \leq l_y < l_{y,SS}+q_{Cy}^b} (C_P^{ub}(k) - \min_{1 \leq y \leq N}(P_{Cy}^{uyb})(l_y)) \tag{3.35}$$

The throughput of the constructed static schedule relates to μ^{-1}, with μ the iteration period (or total execution time of one period) of this periodic schedule. The temporal monotonic behavior guarantees that moving to a self-timed execution after the buffer sizing yields an implementation with at least this throughput.

Practically, the life-time analysis monitors the number of tokens on the forward and the backward edge of all edges e_u in the CSDF graph G: the forward one for the evaluation of the firing condition, the backward one for the buffer capacity calculation. Consequently, the evaluation $P_P^{uyf}(k) - C_C^{uyf}(l_y)$ on e_{uyf} is made at the end of each firing of its producer or consumer. The evaluation $C_P^{uyb}(k) - P_C^{uyb}(l_y)$ on e_{uyb} is made at the start of each firing of its producer or consumer. The maximum over all e_{uy} during the transient phase and one iteration period in the periodic phase of the desired schedule yields the buffer size d_u. Such buffer capacity calculation is part of contribution 2.

The formula for d_u (Equation (3.35)) and the practical approach presented above only provide a basic buffer sizing technique. For an efficient multiprocessor implementation, four related elements need to be considered in the trade-off: throughput, response times, schedule settings and buffer capacities. Optimization algorithms exploring these trade-offs, such as [52, 102, 111] are outside the scope of this work.

In the context of a custom design, it is assumed that processing resource constraints are relaxed enough to implement every actor as a separate HW accelerator (i.e. a task specific processor). To optimize the introduced concurrency during the partitioning while relaxing the response time requirements for the HW design, the desired schedule is a pipelined and parallel operation. This sets the goal of the buffer length calculation to: find the minimal buffer sizes that satisfy the required throughput while also maximizing the response times (within the throughput constraints). Such ideal parallel operation yields a balanced distribution of the load over all processors, hence this desired sched-

ule is called the balancing principle. Consequently, no serialized actors are allowed (i.e. cycles contain sufficient initial tokens). This requirement leads to an approach where first an acyclic graph is considered for the buffer sizing. Then the feedback edge of the cycle is reinserted to verify it has no impact on the desired schedule. Note that both regular and special channels are considered as a single edge and hence they are not interpreted as cycles. The same is true for the self-cycles.

For the acyclic graph, the worst case actor Response Time (RT) is set to its critical RT, defined as

$$RT_A^{crit} = \frac{\mu}{q_A^b} \tag{3.36}$$

which directly relates to the throughput required in the specification through the iteration period μ of one period of the desired pipelined parallel schedule.

Ade [10, 11] shows that the minimal buffer sizes in an arbitrary SDF graph can be obtained by breaking the graph down in chains and clusters. The edge sizes are then calculated for these basic graph entities, as different chains and clusters do not influence each other. In this way, the buffer capacity calculation is simplified as it is divided over smaller entities.

Practically, the buffer sizing of the CSDF graph under the balancing principle is calculated using the following steps:

1. Reduce the CSDF graph G to an acyclic graph G' by removing the feedback edges in the cycles.
2. Calculate the basic repetition rates and the critical response times of all actors.
3. Break the CSDF graph G' down in basic graph entities: chains and clusters.
4. For each basic graph entity
   - Calculate the repetition rate.
   - Construct the desired schedule from start until every actor has fired at least its repetition rate times during steady state. Monitor the number of tokens on e_{uyf} at the end of the firing of an actor of edge e_u and count the number of tokens on e_{uyb} at the start of the firing of an actor on edge e_u.
5. Add the feedback edge(s). Verify that the desired schedule is not affected.

Example To illustrate the buffer sizing process under the balancing principle, Figure 3.26 presents a CSDF graph using the special channels with one chain (A,B,C) and one cluster (C,D,E).

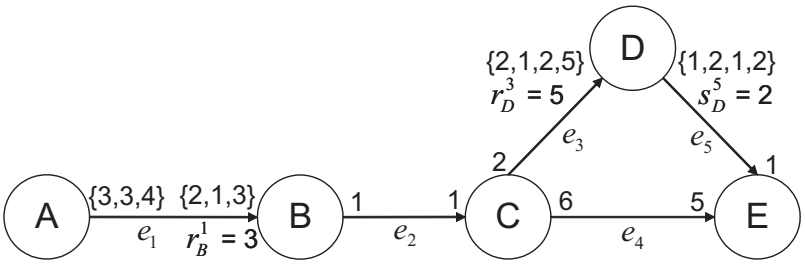


Figure 3.26: Example CSDF graph with one chain and one cluster.

The topology matrix Γ and the actor period matrix L are given by:

$$\Gamma = \begin{pmatrix} 10 & -6 & 0 & 0 & 0 \\ 0 & 3 & -1 & 0 & 0 \\ 0 & 0 & 2 & -10 & 0 \\ 0 & 0 & 6 & 0 & -5 \\ 0 & 0 & 0 & 6 & -1 \end{pmatrix}$$

$$L = \begin{pmatrix} 3 & 3 & 1 & 4 & 1 \end{pmatrix}$$

The rank of $\Gamma = 4$, indicates a solution exists for the balance equation. Solving this balance equation $\Gamma \cdot r^T = 0$ yields

$$r = \begin{pmatrix} 3 & 5 & 15 & 3 & 18 \end{pmatrix}$$

The repetition vector $q = r \cdot L_{diag}$ equals

$$q = \begin{pmatrix} 9 & 15 & 15 & 12 & 18 \end{pmatrix}$$

The basic repetition vector q^b equals the repetition vector q as the greatest common divider of the elements of vector r is 1.

Assuming an iteration period $\mu = 180$, the critical response times RT^{crit} are

$$RT^{crit} = \begin{pmatrix} 20 & 12 & 12 & 15 & 10 \end{pmatrix}$$

After breaking this CSDF graph up in its chain and cluster, the construction of the desired schedule with the life-time analysis is applied to find the required buffer capacities. For each basic entity, the basic repetition rates are

calculated, as at least up to one complete repetition of every actor in the entity needs to be evaluated. Tables 3.3 and 3.2 respectively dimension the edges in the chain and the cluster. The life-time analysis is made on the short-hand notation of the regular or special channel, with 'req' the number of acquired containers at the start of a task execution and 'end' the number of competed containers on the channel at the end of a task execution. The preamble phase is separated from the steady state by an empty line in the tables.

The topology matrix and basic repetition rates of chain (C, D, E) are given by:

$$\Gamma_{cluster(C,D,E)} = \begin{pmatrix} 2 & -10 & 0 \\ 6 & 0 & -5 \\ 0 & 6 & -1 \end{pmatrix}$$

$$q^{b}_{chain(C,D,E)} = \begin{pmatrix} 5 & 4 & 6 \end{pmatrix}$$

Table 3.2: Tracking the required buffer size of the edges in the cluster (C,D,E).

Actor firing	Time (RT)		# on e_3		# on e_4		# on e_5	
	t_{start}	t_{end}	req	end	req	end	req	end
C	0	12	2	2	6	6	-	-
C	12	24	4	4	12	12	-	-
C	24	36	6	6	18	18	-	-
C ‖ D	36	48	8	8	24	24	2	-
C ‖ ...	48	51	10	6	30	-	2	1
... ‖ D ‖ E	51	60	10	8	30	30	3	-
C ‖ ... ‖ ...	60	61	10	-	36	25	3	0
... ‖ ... ‖ -	61	66	10	7	36	-	3	2
... ‖ D ‖ E	66	72	10	9	36	31	4	-
C ‖ ... ‖ ...	72	76	11	-	37	26	4	1
... ‖ ... ‖ E	76	81	11	7	37	-	4	2
... ‖ D ‖ ...	81	84	11	9	37	32	4	-
C ‖ ... ‖ ...	84	86	11	-	38	27	4	1
... ‖ ... ‖ E	86	96	11	6	38	28	4	2
C ‖ D ‖ E	96	106	8	-	34	23	4	1
... ‖ ... ‖ E	106	108	8	8	34	29	4	-
C ‖ ... ‖ ...	108	111	10	6	35	-	4	2
... ‖ D ‖ ...	111	116	10	-	35	24	4	1
... ‖ ... ‖ E	116	120	10	8	35	30	4	-
C ‖ ... ‖ ...	120	126	10	7	36	25	4	2
maximum			11		38		4	

Taking edge e_3 as example, the fifth row of Table 3.2 has start time 48 and end time 51. It contains a new execution of C. At the start time, 2 containers are acquired on top of the 8 completed ones (indicated by end of the previous row), making req 10. At the end time, only the execution of D ends, and 2 of the 8 completed containers are consumed, making end equal to 6.

The topology matrix and the basis repetition rates of chain (A, B, C) are given by:

$$\Gamma_{chain(A,B,C)} = \begin{pmatrix} 10 & -6 & 0 \\ 0 & 3 & -1 \\ 0 & 0 & 2 \end{pmatrix}$$

$$q^{b}_{chain(A,B,C)} = \begin{pmatrix} 9 & 15 & 15 \end{pmatrix}$$

Table 3.3: Tracking the required buffer size of the edges in the chain (A,B,C).

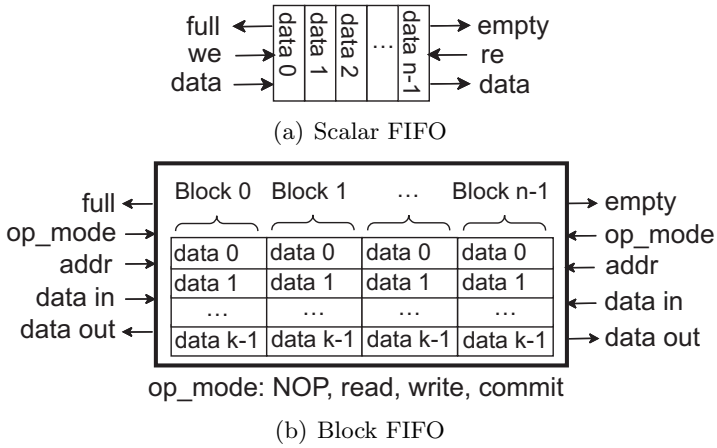
Actor firing				Time (RT)		# on e_1		# on e_2	
				t_{start}	t_{end}	req	end	req	end
A				0	20	3	3	-	-
A	‖	B		20	32	6	1	1	1
...	‖	-	‖ C	32	40	6	4	-	-
A	‖	B	‖ ...	40	44	8	-	2	0
...	‖	...	‖ -	44	52	8	3	2	1
...	‖	B	‖ C	52	60	8	7	2	-
A	‖	...	‖ ...	60	64	10	4	2	1
...	‖	B	‖ C	64	76	10	2	2	1
...	‖	-	‖ C	76	80	10	5	-	-
A	‖	B	‖ ...	80	88	8	-	2	0
...	‖	...	‖ -	88	92	8	4	2	1
...	‖	B	‖ C	92	100	8	7	2	-
A	‖	...	‖ ...	100	104	11	4	2	1
...	‖	B	‖ C	104	116	11	2	2	1
...	‖	-	‖ C	116	120	11	6	-	-
A	‖	B	‖ ...	120	128	9	-	2	0
...	‖	...	‖ -	128	132	9	5	2	1
...	‖	B	‖ C	132	140	9	8	2	-
A	‖	...	‖ ...	140	144	11	5	2	1
...	‖	B	‖ C	144	156	11	3	2	1
...	‖	B	‖ C	156	160	11	6	2	-
A	‖	...	‖ ...	160	168	10	5	2	1
...	‖	B	‖ C	168	180	10	6	2	1
A	‖	B	‖ C	180	192	9	4	2	1
...	‖	B	‖ C	192	200	9	7	2	-
A	‖	...	‖ ...	200	204	10	6	2	1
...	‖	B	‖ C	204	216	10	3	2	1
...	‖	B	‖ C	216	220	10	6	2	1
A	‖	...	‖ ...	220	228	10	4	2	1
...	‖	B	‖ C	228	240	10	7	2	1
A	‖	B	‖ C	240	252	10	4	2	1
...	‖	B	‖ C	252	260	10	7	2	-
A	‖	...	‖ ...	260	264	10	5	2	1
A	‖	B	‖ C	264	276	10	4	2	1
...	‖	B	‖ C	276	280	10	7	2	-
A	‖	...	‖ ...	280	288	11	4	2	1
...	‖	B	‖ C	288	300	11	6	2	1
A	‖	B	‖ C	300	312	9	5	2	1
maximum						11		2	

3.5 Hardware Communication Primitives

The communication between the task specific processors (IP blocks), consisting of both regular channels and the special channels described earlier, is supported by a fixed set of communication primitives (contribution 4). They are built and verified separately to assure the correct operation of this communication library. The obtained separation of communication and computation simplifies the HW design (see Chapter 4). Two main groups are distinguished: synchronizing and non-synchronizing CPs.

3.5.1 Synchronizing Queues

The synchronizing CPs serve two purposes: (i) they signal the presence of a token and (ii) they store the data of the container. In this way, they realize the blocking read and blocking write of a channel with a fixed buffer size. Two types are available: Scalar FIFOs (Figure 3.27(a)) and Block FIFOs (Figure 3.27(b)).



(a) Scalar FIFO

(b) Block FIFO

Figure 3.27: Synchronizing Communication Primitives

3.5.1.1 Scalar FIFO

The scalar FIFO, represented in Figure 3.27(a), passes frequently changing configuration information used for processing data units. The scalar FIFO is implemented as a typical FIFO queue with full and empty blocking behavior. The FIFO control signals allow the acquiring and releasing of its containers that hold scalar data.

As a pop operation on a FIFO releases the container during the next cycle, its depth equals the calculated buffer capacity minus 1.

3.5.1.2 Block FIFO

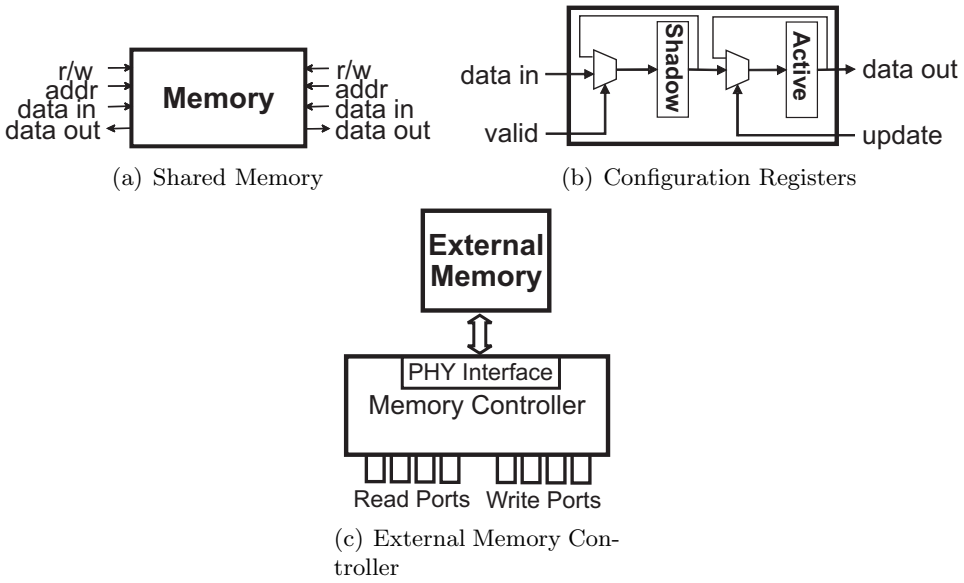
The block FIFO, represented in Figure 3.27(b), passes data units between processes (typically a (macro)block), and is implemented as a first in first out queue of data containers. The data in the active container within the block FIFO can be accessed randomly. The active container is the block that is currently produced/consumed on the production/consumption side. Consequently, the random access capability of the active container requires the extension of the control signal (op_mode) to allow the following operations: (1) NOP, (2) read, (3) write and (4) commit. The commit command indicates that the active block processing has been finished and can be transferred to the consumption side or freed depending on the side.

The Block FIFO offers interesting extra features:

- Random access in a container allowing to produce values in a different order than they are consumed, like the (zigzag) scan order for the (I)DCT.
- An active container can be used as scratch pad for local temporary data.
- Transfer of variable size data as not all data needs to be written.

3.5.2 Non-Synchronizing Elements

To support the implementation of the special channels, the following communication primitives are introduced: shared memory and configuration registers. As they do not provide synchronization support, they can only be used between processes that are already connected (indirectly) through synchronizing CPs. This implies that the synchronization of special channels should always be redundant in the dataflow graph. If this is not the case, i.e. the special channel is the only path between the tasks it connects, an extra channel, for instance passing void containers, needs to be added to the graph, so that the special channel becomes redundant. Note that the choice of shared memory to realize the special channels is an implementation option adding flexibility to allow new types of special channels. They could also be implemented as specialized queues.



(a) Shared Memory

(b) Configuration Registers

(c) External Memory Controller

Figure 3.28: Non-synchronizing Communication Primitives

3.5.2.1 Shared Memory

The shared memory, presented in Figure 3.28(a) is used to share pieces of a data array between two or more processes, and typically holds data that is reused potentially multiple times by them (e.g., the search area of a motion estimation engine). Shared memories are conceptually implemented as multi-port memories, with the number of ports depending on the amount of processing units that are simultaneously accessing it.

Larger shared memories, with as special cases external memory, are typically implemented with a single port. In this case, a memory controller containing an arbiter handles the accesses from multiple processing units (Figure 3.28(c)).

3.5.2.2 Configuration Registers

The configuration registers are used for unsynchronized communication between functional components or between hardware and remaining parts in the software. They typically hold the scalars configuring the application or the parameters that have a slow variation (e.g., frame parameters). The parameters are used for unidirectional communication (read only and write only), or for bi-directional communication as read/write registers. The configuration registers (Figure 3.28(b)) are implemented as shadow registers, giving the freedom to the producer process to write the values whenever he is ready and to the consumer process to update and store the values when needed in

the active element of Figure 3.28(b). This producer and consumer need to be synchronized through another path.

3.5.3 Power Efficiency and Ease of Use

The presented CPs all enable zero-copy communication, i.e. the data does not need to be copied to or from a local buffer prior to sending or receiving. As a result, the number of data accesses to the memories in the CPs remains unaffected by the communication. This is beneficial to the power efficiency and avoids extra memory storage.

The acquiring phase in the task execution, where the presence of the input containers and the space for the output containers is checked, simplifies the HW design, as continuously monitoring the empty or full flags becomes superfluous. This interesting property is mainly enabled by the use of block FIFOs.

As only a limited set of communication primitives is sufficient to support all types of channels, they are standardized. Consequently, they can be implemented separately and upfront to build up a functionally verified communication library. This socketization also improves the IP reuse and aids during the integrating on a fast prototyping board as the interfacing is strictly defined.

3.6 Related Work

Cyclo-Static Dataflow (CSDF, [19]) models of computation match well with the dataflow dominated behavior of video processing. However, modeling implementation specific issues, often related to the use of shared buffers with a fixed size, requires using the CSDF model in a non-trivial way.

Other works confirm the need to express optimized communication by mentioning the use of extensions to (C)SDF to describe image [32] and video [61] applications, but lack a formal description of these extensions. Reference [54] integrates CSDF in a parameterized dataflow, supporting dynamic data production and consumption rates, to expose parallelism at a finer model granularity and to allow reducing memory accesses. To deal with global parameters, reference [88] describes a synchronous piggybacked dataflow model. The modeling of buffer bounds by using a feedback edge is introduced in [100] for inter processor communication graphs (a type of homogenous synchronous dataflow graph) and in [102] to explore the trade-off between throughput and buffer requirements. Our proposed special channels also use a feedback edge to represent the behavior of optimized communication, next to buffer size limitations. In contrast to the listed previous works, they do not need extension of

CSDF as they obey the CSDF definitions and hence the full analysis potential is retained.

A mapping of array-based communication via shared memory onto FIFOs is explained in [64]. Our set of communication primitives realizes the regular and special channels. Similar to [64], the special channels are implemented as shared memory, but they use a controller instead of requiring the existence of a redundant synchronization path (realized by a synchronizing communication primitive). Other papers propose a shared memory implementation of SDF specifications to reduce the buffer requirements based on a lifetime analysis [76, 83]. Our basic buffer capacity approach also uses a kind of life-time analysis to size the buffers of the special channels, but does not yet share buffers between edges.

Papers using an SDF graph during a custom hardware design focus on the automated RTL Code generation of the interfaces and the controller to connect modules, realizing the actors, picked from a hardware library. They either use a centralized controller [60, 63, 87, 112] or distributed control systems [31, 53]. Our limited set of communication primitives strictly defines the supported interface types between HW modules and an automated RTL development and verification environment is build around them in Chapter 4. These communication primitives also support shared buffers next to the FIFO channels of the works above. The self-timed and data-driven operation resulting from the close relation between CSDF model and implementation leads to a distributed control based on the availability and space to consume or produce.

3.7 Conclusion

The CSDF model of computation matches well with the dataflow dominated behavior of multimedia processing, making it a good abstraction means to reason about the parallelism required in an efficient implementation. Among different dataflow models of computation, CSDF is one of the most expressive while preserving the full analysis potential (e.g. consistency checks, dead-lock analysis, etc).

This chapter shows that also implementation specific aspects, like data reuse and shared buffers improving the efficiency or restricting the buffer sizes, can be expressed in an CSDF graph (contribution 2). Such special communication channels often employ a kind of shared circular buffers. They are represented by two edges to correctly model the synchronization and the free buffer space between the communicating tasks. In this way, the graph remains completely analyzable and allows reasoning about its temporal behavior.

With worst case response times and a desired schedule as given, a buffer length

calculation through a life-time analysis of the CSDF model is presented. The obtained relation between the model and the implementation, combined with the temporal monotonic behavior when moving to self-timed execution assures that the throughput of the final implementation is at least the one derived from the iteration period of the desired schedule.

A limited set of communication primitives supports the realization of all types of special channels by exploiting redundant synchronization. They are built as a library for the programming model and can be easily inferred in the HW (contribution 4). Their strict definition enables their separate functional verification.

Chapter 4

Verification Strategy and RTL Development

I was lookin' back
To see if you were lookin' back at me
To see me lookin' back at you

'Safe From Harm' – Massive Attack

Custom hardware implementations of video applications are the most energy efficient solution to support the high throughput requirements within the battery capacity and heat dissipation limitations of a portable device [26, 91]. Their design requires a correct translation of the high-level, optimized C specification to the final implementation at the RTL level. To overcome this gap, a predictable design trajectory is needed guaranteeing the properties of a product, concerning e.g. power, area, real-time and functional correctness. Such predictable design trajectory requires [29]: (i) being able to reason at a high level about a design, in terms of functional and non-functional properties and (ii) being able to realize this design, while preserving all its properties, without time consuming iterations (design closure).

This chapter focuses on the real-time and functional correctness properties of a design and shows how the proposed design flow of Chapter 2 supports their predictability. Towards RTL level, a verification strategy and an automated RTL development approach is introduced. The strict definition of the communication primitives in Chapter 3 and the availability of the optimized C specifications are key elements during this design step. They allow upfront validation of the concurrency and correctness of the system and enable the

isolated development and extensive functional verification of a single functional component during RTL translation, including correct communication modeling. In this way, the debug cycle during system integration is reduced.

The upfront verification uses a high-level concurrent SystemC model, automatically generated from the single threaded optimized C specification by the SPRINT tool. From the same optimized C specification, the RTL verification extracts the input stimuli and the expected output for the design of the task specific processors. The strictly defined set of communication primitives (Section 3.5) provide the clean interface needed in compositional design [90]. They allow exploiting the concept of separation of communication and computation [65] to develop and verify isolated hardware functional components by cutting them out of the system at the borders of their I/O (i.e. the functional block and its communication primitives). The RTL development and simulation environment is automatically generated, allowing the designer to focus on the HDL architecture design of the component.

As the number of stimuli required to completely test a functional module can be significant, the development environment supports simulation as well as testing on a fast prototyping platform. While the high signal visibility of simulation normally produces long simulation times, the fast prototyping platform supports much faster and more extensive testing with the drawback of less signal observability. When an error is encountered on the fast prototyping platform, the designer can isolate its position and return to a restricted simulation with higher observability. In this manner, a low-level building block of the system is first tested extensively to assure a smooth integration and minimize the debug cycle of the complete system.

This chapter first introduces the verification strategy in the next section and then explains in Section 4.2 the benefit of separating the communication from the computation. Section 4.3 builds the high-level transaction level model to evaluate the concurrency and correctness of the parallel tasks. The RTL development environment of Section 4.4 supports the design of a dedicated functional component for every task. Section 4.5 combines simulation and fast prototyping for their functional verification.

4.1 Verification Strategy

In proving the correctness of a design, two aspects are usually distinguished: validation and verification [29]. Validation checks whether the right design is made, i.e. is it doing what it should do? This can be ambiguous and out of the scope of this chapter, since it is often not precisely and completely known how the product under design should behave, in all circumstances, for all possible

uses. Verification checks if the system is designed right, i.e. whether two descriptions at different abstraction levels behave the same.

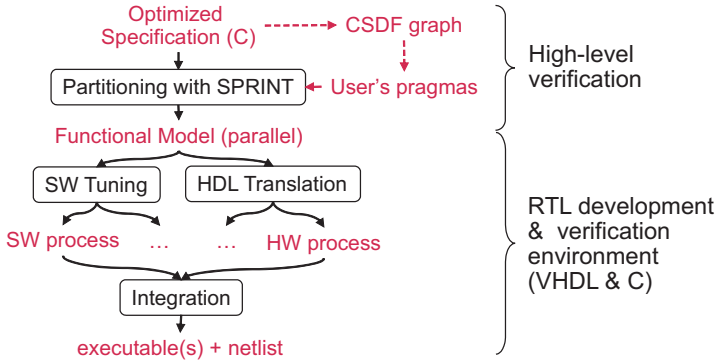


Figure 4.1: A verification approach is applied to all steps in the parallel design phase.

Figure 4.1 extracts the part to which the verification will be applied from the design flow of Chapter 2. Since the algorithmic optimizations simplify functionality (at a controlled cost of a lower compression efficiency), the verification starts from the optimized specification (C). Both the functional correctness and non-functional aspects, mainly throughput and buffer sizes, are considered.

To reason about non-functional properties of the parallel system, like buffer capacities and throughput, the partitioning exploration builds a CSDF model. The consistent relation between this CSDF model and an implementation (discussed in Chapter 3) and the temporal monotonic behavior of a self-timed executing system assure that the throughput derived from the CSDF model, when using Worst-Case Response Times (WCRT), equals the throughput of the final implementation: (i) if the buffer capacities calculated using a desired schedule and the WCRTs are respected in the implementation and (iii) if the execution time of a process is always smaller than or equal to the corresponding WCRT in the CSDF model. Though theoretically superfluous, the high-level verification of Section 4.3 cross-checks the buffer capacities and achieved throughput using a SystemC model. Additionally, as production and consumption sequences in the CSDF model are constructed currently by hand, the correctness of these firing rules can be evaluated.

Towards RTL, the state machines need to correctly implement the firing rules of the actors they implement. As RTL functional verification, a compositional approach is proposed: functional blocks are translated to HDL and verified individually and only when the correctness of their functionality is assured by regression testing, they are composed to build up the complete system. This isolated development is enabled by the data-driven operation of all tasks

in the self-timed executing system that naturally leads to the separation of communication and computation (see Section 4.2). The output data generated by a task only depends on the input required for the task execution and its history (i.e. internal state depending on inputs of previous task invocations). Because the communication is consistent from CSDF model down to final implementation (the regular and special channels are realized by a limited set of CPs), the test stimuli for RTL verification are generated from the optimized specification that is used as golden model. This RTL verification combines the signal visibility of RTL simulation with the testing speed of fast prototyping. The set-up of both environments for the individual block under development is automated (see Section 4.4) and seamless switching between both is supported (see Section 4.5).

Overall, the proposed verification strategy tries to identify errors as early as possible in the design flow. This reduces the source of error when moving the RTL integration phase and hence shortens the debug cycles.

4.2 Separating Communication and Computation

Only a limited set of Communication Primitives (CPs, see Section 3.5) are sufficient to support the special channels of Chapter 3. Their restricted number and their standard definition allows their expression at different abstraction levels and exploiting the principle of separation of communication and computation [65]. The library implementing these CPs is built and verified for the different models upfront to assure its correctness.

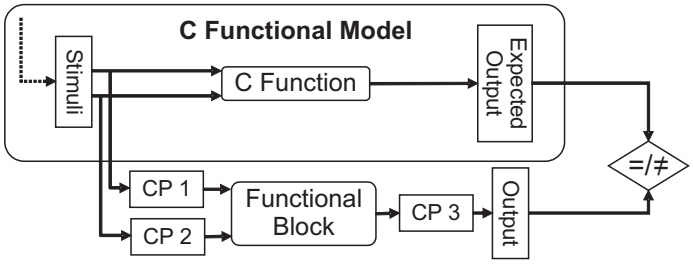


Figure 4.2: The separation of communication and computation enables the functional testing of a functional block by bitwise comparison of its output with the expected output from the functional C model when the same input stimuli are used.

Figure 4.2 shows how a function in the C model is cut out of this C functional model at the borders of its I/O (i.e. the function and its CPs) and corresponds with the same module described as functional block (in SystemC or at RTL level). In this way, functionality in the high-level functional model can be isolated and translated to lower levels, while the component is com-

pletely characterized by the input stimuli and expected output. This creates the possibility to automate the test stimuli generation and the RTL development and functional verification environment for both isolated components and (partially) composed systems (Section 4.4).

4.3 High-Level Verification

The synchronization and the throughput of the parallel system is verified for a first time with a high-level transaction level model: a timed concurrent SystemC model. This model is automatically generated by SPRINT [2] from the sequential optimized C specification, given the task boundaries corresponding to the CSDF graph. Timing annotations containing cycle estimates are used in wait statements in the SystemC model to reflect the response times of the tasks. Theoretically, this verification is merely a cross-checking, as it has been shown that the CSDF model correctly reflects the implementation. Still, this cross-check can reveal errors in the construction of the CSDF model or in the realization of the communication channels, resulting from: (i) wrong production or consumption sequences, (ii) wrong buffer capacity calculations (currently the life-time analysis is made manually) and (iii) wrong interpretation of the redundant synchronization used for the implementation of the special channels.

The buffer capacity calculations of Chapter 3 based on the CSDF graph assume a Response Time (RT) equal to the WCRT to assure the required throughput (i.e. balancing principle) and synchronization on all edges. The SystemC model generated by SPRINT only uses the fixed set of communication primitives and exploits redundant synchronization to realize the special channels, like the final implementation. Additionally, the RTs of the tasks can be set to the Worst-Case Execution Time (WCET) when they are mapped, if this cycle assessment is available at this point in time.

Changing the time annotations of the concurrent SystemC model allows the verification of both the synchronization and the throughput (first part of contribution 5). The synchronization between the parallel tasks of the self-timed executing implementation is controlled by the blocking read and the blocking write behavior of the queues in the system and the depth of these queues. Making the RT of some tasks infinitesimally small checks the blocking behavior of the tasks (i.e. they should not start as long as the required inputs and output spaces are not available). Setting the RT of a single task to virtually zero, while keeping the WCRT for the other tasks, tests the blocking read of its inputs and/or the blocking writes of its outputs, depending on which queue first blocks. Similarly, keeping the RT of a single task equal to its WCRT, while reducing the RT of the other tasks to virtually zero, tests the blocking

write of its inputs and/or the blocking reads of its outputs. Repeating this for all tasks verifies the synchronization of the complete system. Incorrect synchronization will result in a different functional behavior. Further research should investigate the fault coverage of this approach and if other combinations in RTs are needed to increase this fault coverage.

Keeping the RT of all task equal to their WCRT, studies the obtained parallelism. Wrongly sized buffer lengths will result in unexpected stalls during steady state, while the balancing principle assumes full parallel operation (if the actors are not serialized due to ordering constraints) or in a different functional behavior. Setting the RT of the tasks to the most accurate worst case cycle assessment (WCET) allows an analysis of the concurrency and of the obtained throughput of the final system. Note that the design constraint when mapping the task (either on a processor or developing a HW accelerator) is at all moments an execution time smaller than or equal to WCRT, or WCET $\leq$ WCRT.

4.4 RTL Development and Verification Environment

As the number of stimuli required to completely test a functional module can be significant, the development environment supports simulation as well as testing on a fast prototyping platform (Figure 4.3).

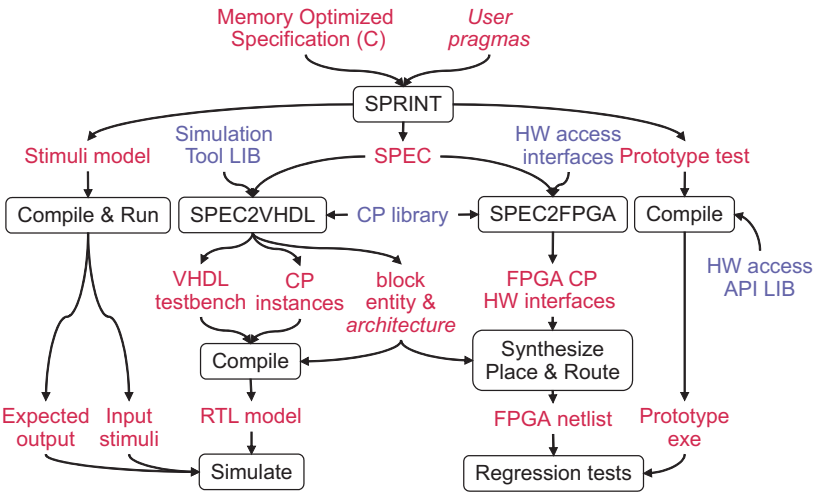


Figure 4.3: The RTL development and test environment supports simulation and fast prototyping so that the designer only needs to focus on the RTL architecture description.

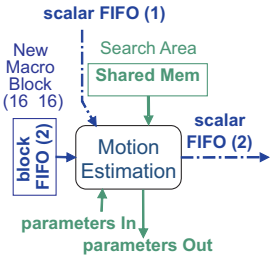
While the high signal visibility of simulation normally produces long simulation times, the fast prototyping platform supports much faster and more

extensive testing with the drawback of less signal observability. Supporting both test mediums, allows the designer to concentrate on the functional description of the component and its implementation.

Starting from the optimized C specification, SPRINT supports both the simulation and the fast prototyping environment (contribution 3). Through user pragmas, the functional block (or set of blocks) under development is selected. As the designer typically starts with simulations, SPRINT generates a specification file (SPEC) containing a list of ports and a stimuli model, that extends the optimized C specification with code for the generation of files containing input stimuli and expected output.

```
void MotionEstimation (uchar *searchArea,       // shared memory in
uchar *newMB,                                   // block fifo in
uchar hIn, uchar vIn, uchar hPosIn,             // scalar fifo in
uchar paramIn1, ushort paramInN                 // configuration regs in
uchar *hOut, uchar *vOut, short *scalarOutN     // scalar fifo out
uint *paramOut1, uchar *paramOutN               // configuration regs out)
```

(a) C Function



(b) Functional block

```
BLOCK MotionEstimation

PORT Primitive=DPRAM Name=SearchArea Type=INPUT Width=8 Size=4608
 File=Stimuli/ME_SearchArea_ref.in.txt

PORT Primitive=blockFIFO Name=NewMB Type=INPUT Width=8 Depth=2 Size=256
 File=Stimuli/ME_NewMB_ref.in.txt

PORT Primitive=scalarFIFO Name=CC2ME_FIFO Type=INPUT Width=15 Depth=1
 File=Stimuli/ME_CC2ME_FIFO_ref.in.txt
 # hPosIn = 3 bits vIn = 6 bits; hIn = 6 bits

PORT Primitive=updateREG Name=MotionEstimation_Update_Reg
 File=Stimuli/ME_Params_ref.in.txt
 # paramIn1; paramInN;

PORT Primitive=scalarFIFO Name=ME2MC_FIFO Type=OUTPUT Width=25 Depth=1
 File=Stimuli/ME2MC_FIFO_dat.out.txt
 # hOut = 6 bit; vOut = 6 bits; scalarOutN

PORT Primitive=validREG Name=MotionEstimation_Valid_Reg
 File=Stimuli/ME_Params_dat.out.txt
 # paramOut1; paramOutN
```

(c) SPEC file

Figure 4.4: Example SPEC file, listing all CPs of a motion estimation functional block.

The example SPEC file of Figure 4.4 shows the correspondence between the optimized specification used as golden model and the instantiated CPs for the motion estimation. The C function arguments (Figure 4.4(a)) are grouped according to their CP type. All inputs are first listed, then the outputs of the

function. Each function argument corresponds to a CP of the functional block of Figure 4.4(b). The SPEC file of the motion estimation functional block (Figure 4.4(c)) contains all CPs with a detailed characterization: CP type, name, direction, the bitwidth, the depth of the CP and a filename to read or write stimuli.

The SPEC2VHDL tool generates, based on the specification (SPEC) file, the VHDL testbenches, instantiates the communication primitives required by the block, and also generates the entity and an empty architecture of the designed block. The testbench includes a VHDL simulation library that links the stimuli/expected output files with the communication primitives. In the simulation library basic control is included to trigger full/empty behavior of the synchronizing queues. The communication primitives are instantiated from a design library, which will also be used for synthesis. At this point the designer can focus on manually completing the architecture of the block.

During the translation of an actor as HW block, the cyclo-static behavior needs to be preserved, i.e. the RTL has to obey and correctly implement the production and consumption sequences. This requires making the state machine only dependent on the availability of input containers and output container space (acquire) and on the position in the frame, and correctly releasing the completed containers.

As the user finishes the design of the block, the extensive testing makes the simulation time a bottleneck. In order to speed up the testing phase, a seamless switch to a FPGA based fast prototyping platform, using the same specification file, is supported. Running SPRINT for this environment will create a prototype program for the block under development. This software application extends the optimized C specification with calls to the hardware access APIs for all inputs and outputs of the block under development. The SPEC2FPGA tool reuses the same SPEC file to generate the hardware prototype consisting of: the platform/FPGA required interfaces, the previously generated entity and implemented architecture. After synthesis and place and route of the hardware and compilation of the prototype program, regression tests can be performed. The software can be used in two ways. First, it also executes the software version of the hardware block under test to compare the prototype outputs with the reference ones of the golden model. Or secondly, only the hardware prototype is called and the verification is done on the final system output.

4.5 RTL Verification

The combination of the two described test mediums, creates a powerful design and verification environment. This leads to an HDL design with a short debug cycle, where even subparts of a single functional block can be tested. As both simulation and hardware verification setups are functionally identical, the error can be identified on the fast prototyping platform, with a precision that will allow a reasonable simulation time in the simulation environment in view of the error correction. For instance, during regression testing, something breaks in sequence X, during the processing of frame Y. From the golden model, the test stimuli for only frame Y are dumped, so that the simulation is restricted to this specific frame (i.e. the simulation does not need to restart from frame 0 if an error occurs in the middle of the sequence). This easy switching from fast prototyping to simulation, without the need to restart from the beginning, enables rigorous verification. It is a consequence of the actor based design of a data-driven and self-timed executing system on condition that the internal state of a functional block is reset or read out at a certain granularity (e.g. a frame in video processing). As a result, all required state information (or history), in this case a frame, can be generated from the golden model and the simulation can (re)start at this granularity. This seamless switching between RTL verification environments in an automated development framework is the second part of the verification strategy of contribution 5.

To minimize the debug and composition effort of the different functional blocks, a verification process in two phases is proposed, both using the two environments of Figure 4.3. The approach is first applied to each functional block individually (Figure 4.5).

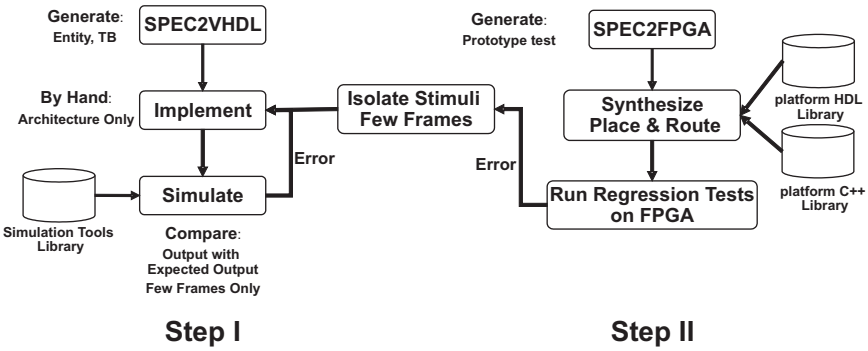


Figure 4.5: Design and verification of a single functional block, combining both simulation and fast prototyping.

Only when all separate blocks have proven to be functionally correct, by passing simulation (step I of Figure 4.5) and regression testing (step II of Figure

4.5), can system integration truly commence. The exhaustive testing of the individual blocks (including the interactions between the block and communication primitives) is the key to a smooth integration (see Figure 4.6). Again, both verification environments are combined during this system completion. First, the system designer generates the simulation environment using SPEC2VHDL and proceeds with the top cell description (step III of Figure 4.6). Then (step IV of Figure 4.6) the generation of the prototype hardware, compilation and running the regression tests complete the verification methodology.

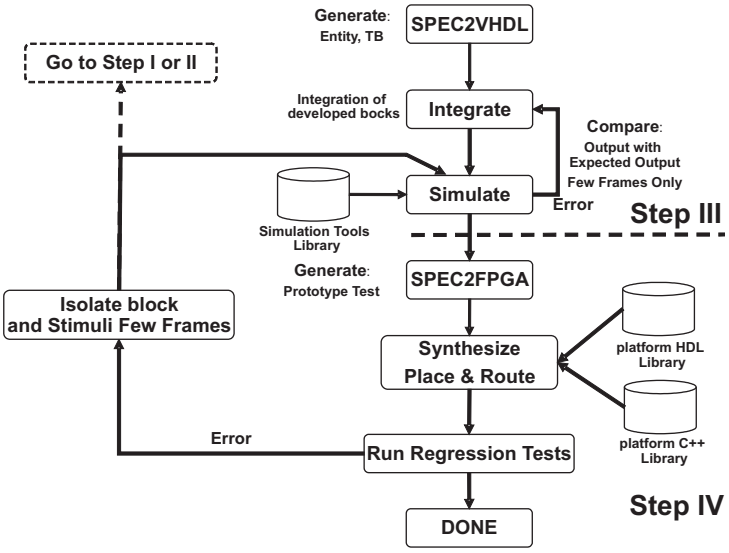


Figure 4.6: Gradual composition of the system. Its verification again combines simulation with fast prototyping.

In this way, the origin of errors in the complete system is restricted to a limited set of possible reasons: (i) inconsistent position of the data in the communication primitives or (ii) problems at the I/O borders of the complete system. An example of the first case is changing the order of scalars when they are concatenated in one word. Any other unexpected interference between blocks is very unlikely as the golden specification proved to be correct. This restricted possible source of errors explains the short integration debug cycle. When the system passes all tests the system is ready for use.

4.6 Related Work

Predictability is an important element in a design flow preserving the functional correctness and non-functional properties over different design levels. A verification strategy is required while translating a higher level specification

(C code) to RTL. Lee lists a set of actor-oriented design frameworks in [69], among which [101] is most close to our RTL development and verification environment. Starting from a MATLAB specification, a Kahn process network is used as model (as YAPI [33] code) in the partitioning phase. Stefanov's approach contains similar steps as the parallel phase of our design flow, but towards RTL level, only an automatic translation is supported. In contrast, we start from C code, build a CSDF model and include a RTL development and verification environment to allow the manual writing of VHDL.

The selected and clearly defined set of communication primitives is a key element of the proposed verification strategy. It allows exploiting the principle of separation of communication and computation [65] and enables an automated RTL development and verification strategy that combines simulation with fast prototyping, with a seamless transition between both.

Other design frameworks offer simulation and FPGA emulation [80], with improved signal visibility in [99], at different abstraction levels (e.g. transaction level, cycle true and RTL simulation), that trade accuracy for simulation time. Still, the RTL simulation speed is insufficient to support exhaustive testing and the detailed behavior of the final system is not repeated at higher abstraction levels. Moreover, there is no methodological approach for RTL development and debug. Amer [13] describes upfront verification using SystemC and fast prototyping [12] on an FPGA board, but the coupling between both environments is not formalized.

4.7 Conclusion

The development of a custom implementation of modern multimedia applications requires a systematic development and verification approach to reach a correct RTL-level description of the system. This chapter introduced a design and verification methodology, supporting the designer in this process (contribution 5). It relies on the fixed set of communication primitives to obtain clear separation of communication and computation. This allows the individual design of a function component from the system, while correctly modeling its communication. The functional verification is obtained by extracting input stimuli and expected output from the optimized specification as golden model.

The synchronization and correctness of the parallel system is first validated with a parallel SystemC model automatically generated by the SPRINT tool, before moving to the RTL-description (contribution 3). To allow extensive testing of the individual block, both testing by simulation of the RTL model and testing the RTL implementation on a fast prototyping platform are supported. The first environment offers a high signal visibility, but suffers from

long simulation times. The prototyping platform achieves up to real-time operation, but has lower signal observability. Based on a HW specification file describing the I/O of the block, testbench support and communication primitive instances are automatically generated by scripts, allowing the designer to concentrate on the architecture description of the hardware block. Easy switching from prototyping to simulation, without the need to restart the simulation from scratch, supports a closely coupled HDL design and debug with a short turn around time. Only after elaborate testing of each single component (passing successfully all cases of the test set), system integration starts. In this way, the debug cycle for integration is minimized. For the MPEG-4 design, a significant design time reduction (over a factor 2) is measured when comparing the encoder, to which the RTL development and verification approach (described above) was strictly applied, with the decoder for which this was not the case.

Chapter 5

Demonstrator: MPEG-4 SP Video Codec Design

If living is seeing
I'm holding my breath
I wonder, I wonder
What happen's next?
A new world
A new day to see

'New World' – Bjork

The (advanced) Simple Profile (SP) of the MPEG-4 part 2 video standard is used for internet streaming, wireless cameras and in mobile phones. Recently, new levels of this standard support up to SD PAL (720×576 at 25 frames per second) [9]. This chapter demonstrates the applicability of the design approach introduced in Chapter 2 to implement a high resolution low power video codec. The proposed flow is used for the development of a fully dedicated, scalable MPEG-4 part 2 Simple Profile video encoder and decoder, able to sustain respectively up to 4CIF (704×576) at 30 fps and XSGA (1280×1024) at 30 fps or any multi-stream combination that does not exceed these throughputs.

After a brief repetition of the MPEG-4 video encoder structure, only the implementation of the encoder is described. Figure 5.1 relates each design step of the flow to a section of the current chapter. Details on the decoder design are given in [97].

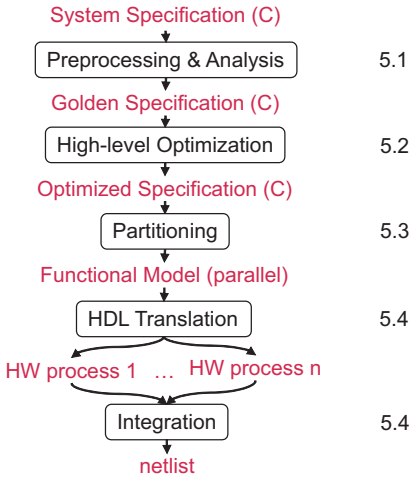


Figure 5.1: Different design steps and the sections where they are applied on the MPEG-4 video encoder.

MPEG-4 part 2 Video Coding Scheme The MPEG-4 part 2 video codec [8] belongs to the class of lossy hybrid video compression algorithms [17]. Figure 5.2 gives a high-level view of the encoder. A frame of width w and height h is divided in macroblocks, each containing 6 blocks of 8x8 pixels: 4 luminance and 2 chrominance blocks (see also Figure 1.2 of Chapter 1). The pair (h, v) defines the position of a pixel or macroblock in a frame, with h the horizontal coordinate and v the vertical one. A row of the frame spans $w_{MB} = \frac{w}{16}$ macroblocks, while a column spans $h_{MB} = \frac{h}{16}$ macroblocks.

The Motion Estimation (ME) exploits the temporal redundancy by searching for the best match for each new input block in the previously reconstructed frame. The motion vectors define this relative position. The remaining error information after Motion Compensation (MC) is decorrelated spatially, using a DCT transform and is then Quantized (Q). The inverse operations Q^{-1} and IDCT, and the motion compensation reconstruct the frame as generated at the decoder side. The chain of DCT, Q, Q^{-1} and IDCT is called the Texture Coding (TC). Finally, the motion vectors and quantized DCT coefficients are variable length encoded. Completed with video header information, they are structured in packets in the output buffer. A Rate Control (RC) algorithm sets the quantization degree, to achieve a specified average bitrate and to avoid overflow or underflow of this buffer.

The Simple Profile of the MPEG-4 part 2 standard is oriented to low-delay, low-complexity coding of rectangular video frames [94]. As a result, this profiles supports only a small set of MPEG-4 visual tools, listed in Table 5.1 [8]. Its main characteristic is the restriction to P frames as temporal prediction.

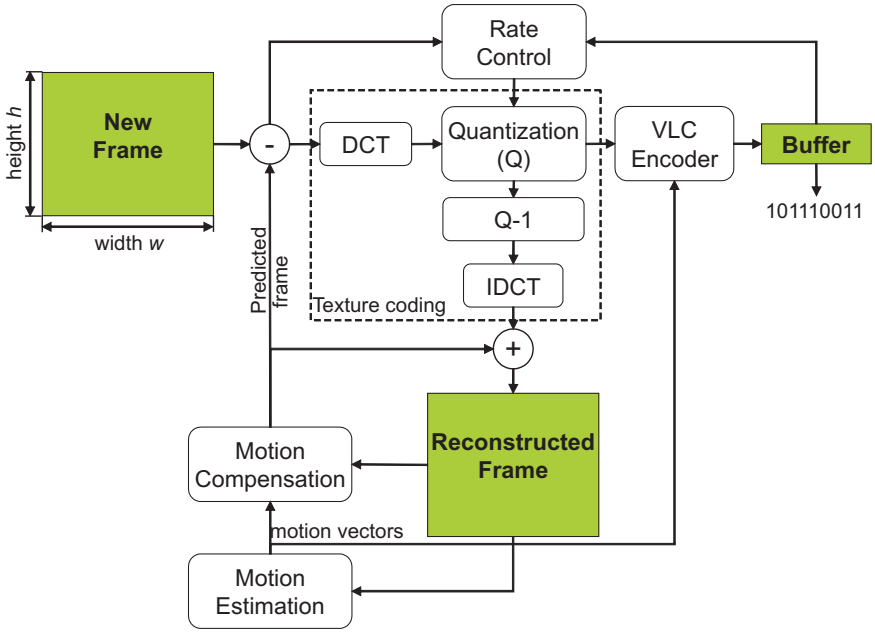


Figure 5.2: MPEG-4 part 2 SP encoder scheme.

Table 5.1: Visual tools of the Simple Profile.

Visual tools	Simple Profile
Basic	I frame (intra)
	P frame (inter)
	AC DC prediction
	4 motion vectors
	unrestricted motion vectors
Error resilience	Slice resynchronization
	Data partitioning
	Reversible variable length coding
Short video header	

5.1 Preprocessing and Analysis

The preprocessing and analysis step prepares the code for the rest of the design flow. It first prunes the executable specification to extract the parts of the code, relevant for the given class of applications. Then the testbench, used in the rest of the chapter to validate the performance of our design optimizations, is defined. An initial cycle and memory access distribution is presented for the typical test case of this set.

The software used as input for this design is the verification model accompanying the Final Draft International Standard natural visual part 2 [7]. This MPEG-4 software specification contains the video tools of all MPEG-4 part 2 profiles and levels, resulting in a large C code, distributed over many files. A necessary first step in the design is thus the extraction of the part of the reference code corresponding to the desired MPEG-4 part 2 Simple Profile. Applying ATOMIUM automatic pruning with a Simple Profile dedicated set of coding parameters (not detailed in this text) shrinks the code to 30 % of its original size (from 67k to 22k lines, see also Table 5.6 in Section 5.2.3). Thanks to this code reduction and simplification, manual and interactive reorganization/rewriting becomes feasible.

Table 5.2: Characteristics of the video sequences in the applied testbench.

Test sequence	Frame rates	Frame sizes	Bitrates (bps)
Mother & Daughter (M&D)	15, 30	QCIF	20k, 60k, 75k
	15, 30	CIF	100k, 120k, 200k
Foreman	12.5, 25, 30	QCIF	50k, 150k, 200k
	12.5, 15, 25, 30	CIF	150k, 400k, 450k, 800k
Calendar & Mobile (C&M)	10, 30	QCIF	300k, 500k
	10, 15, 30	CIF	1M, 1.5M, 2M, 3M
Harbour	15	QCIF	100k
	30	CIF	500k
	30	4CIF	2M
Crew	15	QCIF	200k
	30	CIF	1M
	30	4CIF	4M, 6M
City	15	QCIF	200k
	30	CIF	1M
	30	4CIF	4M, 6M

The testbench of Table 5.2 is used during the different design stages. The 5 selected coding samples have different resolutions, framerates and movement complexity. All sequences are encoded without data partitioning [94] and with rate control enabled (assuming an infinite buffer to avoid frame skipping). They are compressed at several bitrates. In total, 30 test sequences have been defined in this case. The Foreman CIF 450 kbps stream serves as typical representative, when more in depth analysis is made in the rest of se-

quential design phase. It is, with varying motion properties over time, a real life example.

Table 5.3: Memory accesses and time distribution over the main functional modules of the initial video encoder.

Function	Access frequency (Maccesses/s)	Relative # accesses (%)	Relative # cycles (%)
Motion estimation	4149.7	83.9	84.2
Texture coding	286.6	5.8	5.8
Motion compensation	99.6	2.0	1.8
Capture	19.0	0.4	0.7
Reconstruct	26.5	0.5	0.3
Calculate error	11.4	0.2	0.2
Manipulate frame	113.0	2.3	1.5
Padding	53.6	1.1	1.5
Others	186.3	3.8	4.1

Profiling information, gained from simulations, provides a sufficient basis for an initial estimation of the complexity [93]. Combining the memory requirements, measured with ATOMIUM/Analysis, with a cycle distribution obtained with Quantify on a HP9000/J7000, 440MHz RISC platform, allows the identification of the most demanding tasks of the coding algorithm and the verification of their data-dominance. Table 5.3 lists the memory access and time distribution over the main functional blocks of the initial video encoder specification. Next to the processing steps, detailed in Figure 5.2, frame padding and manipulation are added to the reference software. The first function extends a frame with a border to handle motion vectors pointing outside of the image, the second one makes internal copies of a frame.

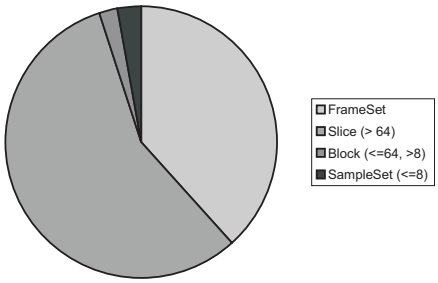


Figure 5.3: Initial distribution of the memory accesses over different memory sizes. Most accesses are made to frameSets and slices.

Figure 5.3 groups the accesses to the 4 memory size categories (of Subsection 2.1.1), related to division of a frame in macroblocks and blocks: frameSets with as minimal size the image, slices containing more than 64 elements, blocks with

9 to 64 elements and sampleSets with maximally 8 elements. In this analysis stage, the wordlength of the elements is not yet considered. Figure 5.3 represents the non-optimized status: 40 % of the total number of accesses is to frameSets, 55 % to slices, 2.3 % to blocks and 2.8 % to sampleSets. The dataflow of the reference encoder is frame-based (Figure 5.4: most of its accesses are to slices or frameSets. This is in contrast to the ultimate goal of the memory optimizations: a scheme with the absolute minimal number of (large and hence costly to access) frame memories and associated accesses to them. The typical subdivision of a frame in macroblocks in MPEG-4, makes a macroblock-based processing to increase data locality a logical solution. However, an important hurdle is the rate control, traditionally operating on statistics of a full frame (Figure 5.4). Subsection 5.2.1.1 explains the removal of this Rate-Distortion (R-D) modeling barrier.

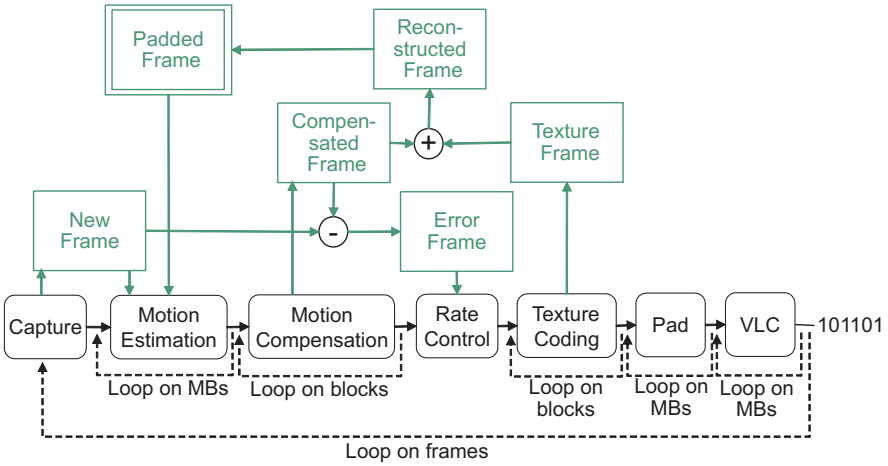


Figure 5.4: The initial video encoder has a frame-based processing. Each step of the algorithm processes the data locally, within a (macro)block loop. A complete MB-based flow is hindered by the rate control dependency.

No new major cycle consuming tasks are identified during the RISC platform analysis with Quantify (Table 5.3): the high correlation between the memory access ratios and the cycle distribution confirms the data-dominance of the functions in this table. The known main bottleneck of a video encoder is the motion estimation [66, 67], taking more than 80 % of the resources. This results from the full search algorithm, typically included in reference code as it guarantees finding the best match by evaluating every point in the search window. Practical implementations (in computation- or power-limited applications) use the freedom available at the encoder side (i.e. the ME algorithm is not specified by the standard) to replace the computationally too expensive full search ME by a fast version, that only searches a subset of locations within the search window. The selection of an appropriate fast ME algorithm

is essential in the optimization of a video encoder, as it has a high influence on the total data transfer and memory usage. Subsection 5.2.1.2 presents the low complexity search algorithm developed during the high-level optimizations.

5.2 High-level Optimizations

The high level optimizations combine algorithmic tuning with dataflow and loop transformations of the DTSE [22] methodology.

5.2.1 Algorithmic Tuning

Two kinds of algorithmic optimization are applied to the video encoder: modifications, to enable a one-pass sequential processing of the frame macroblocks (i.e. increasing memory access locality) (Subsection 5.2.1.1), and tuning, to reduce the required processing for each macroblock (Subsections 5.2.1.2 and 5.2.1.3). Both aim at trading a minimal compression performance loss for a substantial complexity reduction (Subsection 5.2.1.4).

5.2.1.1 Predictive Rate Control

The limited bandwidth resources of existing networks restrict the available output bitrate of a video encoder. Typically, standardized hybrid video coders (like MPEG-4) produce a highly variable output bitrate, depending on the characteristics of the video content. In practice, a FIFO buffer is placed between the coder and the network. As its size is limited (proportional to the transmission delay), the encoder has to carefully regulate its output rate to avoid buffer overflow or underflow, while producing an acceptable visual quality. The default RC adopted by MPEG-4 is a technique combining rate-distortion modeling with sequence pre-analysis, to achieve the best performance [25]. This approach makes a one-pass sequential encoder implementation impossible as a measure of the average compensation error energy, e.g. the Mean Absolute Difference (MAD) is needed to fix the quantization parameter (Figure 5.4). Because this pre-analysis step is performed on the whole compensation error frame, quantization and bitstream generation can not start before motion estimation has been performed on all the macroblocks of the frame. This forms a barrier for the memory optimizations (of Subsection 5.2.2). The development of a predictive rate control (a detailed description is given in [34]), calculating the MAD by only using past information, breaks this restriction and enables a true macro block-based processing. Additionally, the number of frame memories is reduced, as the storage of the compensation error frame becomes obsolete.

5.2.1.2 Directional Squared-Search Motion Estimation

The high computational complexity of the full search motion estimation requires algorithmic tuning, to decrease the number of SAD calculations during the search. Reference [35] describes a directional squared search motion estimation, based on a set of common design rules for efficient ME algorithms, derived from literature. This method, like other fast block matching algorithms, trades a minimal loss of coding efficiency to achieve a significant reduction of the number of searched positions and consequently has an improved cost efficiency. Moreover, the directional squared search ME is developed with hardware implementation constraints and data reuse possibilities in mind: the algorithm maintains a highly regular dataflow, allowing an efficient data reuse implementation and considers adjacent positions, enabling parallelism.

5.2.1.3 Intelligent Error Block Processing

The (I)DCT and quantization in the texture coding chain requires many multiplications and additions and hence requires a considerable amount of processing. When the error block holds a low amount of energy, it is likely that the quantization will reduce all these non-relevant coefficients to zero. Such all-zero blocks are called *skipped* as they do not need texture decoding. The coding status of an error block is only known after the DCT and quantization. For a skipped block the computations in these steps become overhead. Algorithmic tuning predicts the coding status, based on statistics of the block available from the ME. Next to skipped blocks, also blocks with only one row or column of non-zero coefficients are discriminated and predicted (see Chapter 6). The result of this intelligent block processing is a minimized effort in the texture coding, depending on the amount of energy in the error block.

5.2.1.4 Impact on the Compression Performance

Each of the three algorithmic optimization steps (predictive RC, directional squared-search ME, intelligent error block processing) sacrifices a minimal amount of compression performance for a substantial complexity reduction, as quantified in Subsection 5.2.3. For the typical test case (Foreman CIF 450 kbps) 0.5 dB quality loss is traded for a lower implementation cost: the ME requires 25 times less memory accesses, the texture coding makes 3 times less memory accesses and the bit rate controller is simplified. Figure 5.5 measures the impact on the quality, by means of Rate-Distortion (RD) graphs for the video sequences of the testbench. For each test sequence, an RD curve is drawn, representing the behavior of the initial (init) specification and one with the result of the optimizations in the sequential phase (optSeq). In many

cases, both curves are on top of each other, indicating no compression performance is lost. Maximally, 0.5 dB quality is traded for the significant complexity reduction. Table B.2 lists the obtained quality and bitrate (compression performance) with the difference with respect to the initial code (without algorithmic tuning) for all test cases in the testbench.

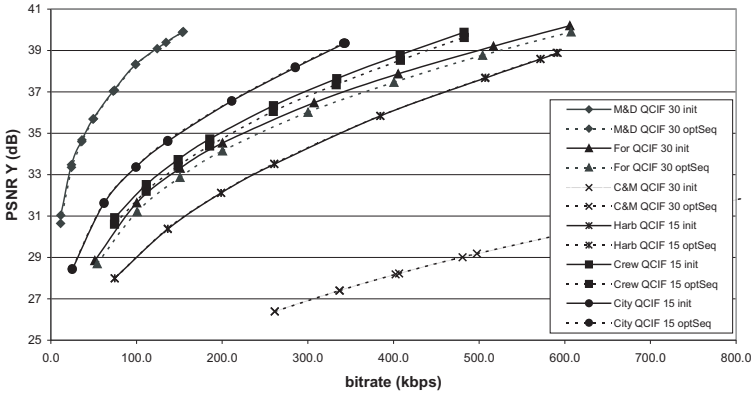
5.2.2 DTSE Optimizations

Global dataflow and loop transformations are a second type of optimizations in the high-level optimization design step. In contrast to the algorithmic tuning, they do not change the functional behavior. Their primary goal is to improve locality, by introducing a memory hierarchy and by changing the operation of the encoder into a macroblock-based processing.

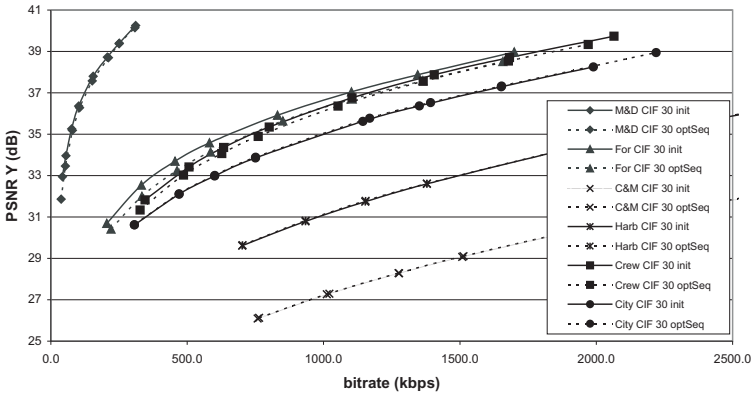
5.2.2.1 Macroblock-based Loop and Data Flow Transformations and Data Reuse

The initial analysis of Section 5.1 points to the motion estimation as the main memory bottleneck, and to the frame-based character of the reference software: a worst-case combination from a memory point of view. However, the motion estimation is intrinsically a localized process: the matching criterion computations repeatedly access the same set of neighboring pixels. To reduce these accesses to frame size memories, a memory hierarchy is introduced in the motion estimation, enabling data reuse. Heavily used data is copied from large (frame size) to minimal intermediate memories, used for the actual ME processing. The solution is more efficient as soon as the cost of extra memory transfers (due to copies) is balanced by the advantage of using smaller memories. The choice of an optimal hierarchical memory partition for motion estimation has been extensively studied in [21, 104], focusing on the full search ME algorithm.

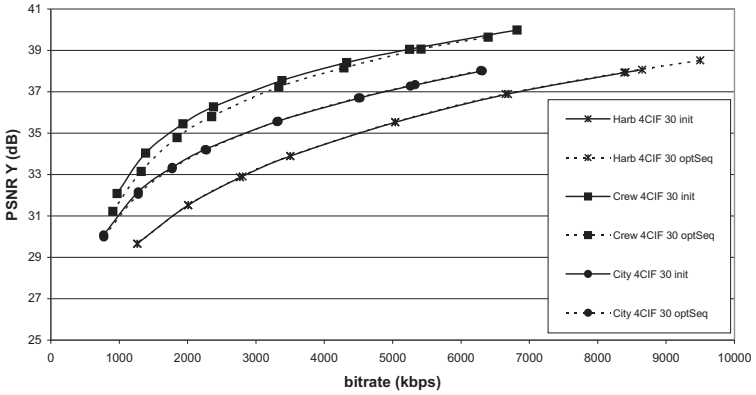
With regard to the adaptive dataflow of the algorithmically tuned ME (see Subsection 5.2.1.2 and [35]), only three levels of hierarchy are relevant (Figure 5.6). For the luminance information, a $(2 \times width + 3 \times 16) \times 16$ pixels buffer Y stores the data of the previous reconstructed frame required by the motion estimation/compensation. The search area buffer is a $(3 \times 16) \times (3 \times 16)$ local copy of the values, repetitively accessed during the SAD calculation. It is circular in the horizontal direction to reduce the amount of writes during the updating of this buffer. Both chrominance components have a similar pixels buffer U/V to copy the data of the previously reconstructed frame, needed by the motion compensation. In this way, the newly coded (macro)blocks can immediately be stored in the frame memory. The motion compensation is per-



(a) QCIF



(b) CIF



(c) 4CIF

Figure 5.5: Rate distortion graphs for the different sequence show a maximal loss of 0.5 dB in compression efficiency due to the algorithmic tuning.

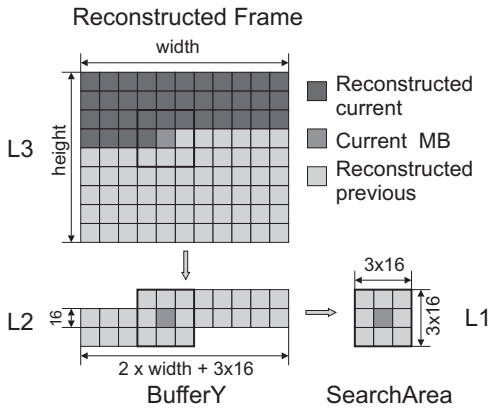


Figure 5.6: Three level memory hierarchy enabling data reuse on the motion estimation and compensation path.

formed on the Y/U/V buffer. Note that the two additional lowest data reuse levels, introduced in [21, 104], can inherently and adequately be implemented by the ME processing kernel (see Section 5.4).

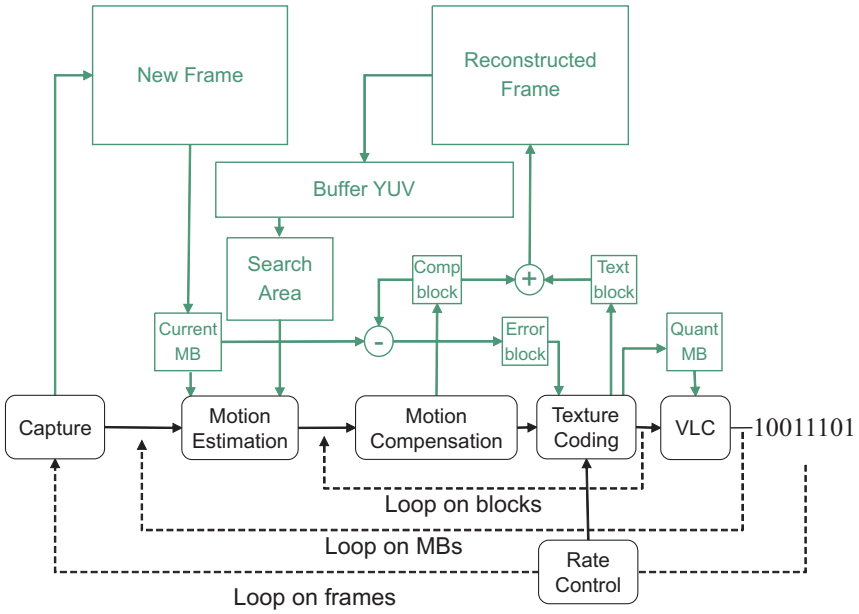


Figure 5.7: The optimized video encoder works in a macroblock-based way to localize the processing steps.

To increase locality, the encoding algorithm is reorganized according to Figure 5.7. Starting from Figure 5.4, the predictive RC allows moving this part to the end of the frame loop. The padding is only performed when needed in the ME and MC. The macroblocks loops over ME, MC, TC and VLC now span these

4 components, supporting macroblock-based processing. The frame memories between them are reduced to block sized (Texture Block, Comp. Block, Error Block inside the TC) and macroblock sized (Current MB, Quant. MB) buffers, matching the blockFIFO CP and layered extensions thereof (Search Area, Buffer YUV, see Figure 5.6), requiring the shared memory CP. The algorithmically tuned rate control (see Subsection 5.2.1.1), combined with the introduced redundancy and the macroblock-based dataflow of the coding process, makes two frame memories sufficient to complete the encoding (Figure 5.7): one for the input (new Frame) and one for the reconstructed data (reconstructed frame).

5.2.2.2 Sub-macroblock Localized Processing and Buffer Size Reduction

A block-based loop is implemented for the DCT/Q/Q^{-1}/IDCT chain of the texture coding. The typical row-column decomposition in a (I)DCT implementation allows a further refinement to column-based processing (Figure 5.8). After the row DCT processing of the 8×8 input block, the column processing sequentially applies the DCT, the quantization, inverse quantization and the IDCT on each column. The row IDCT completes the texture coding chain. This optimization thus interchanges the row and column processing of typical IDCT modules to a column-row decomposition version.

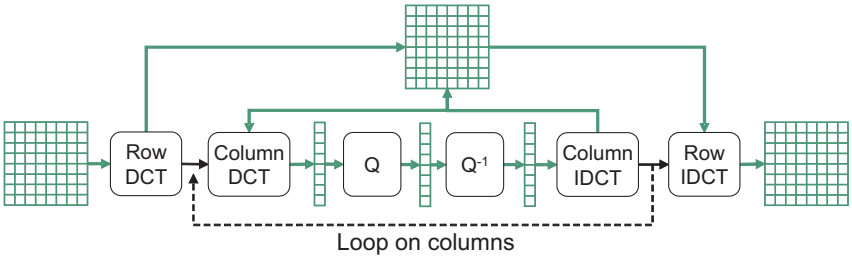
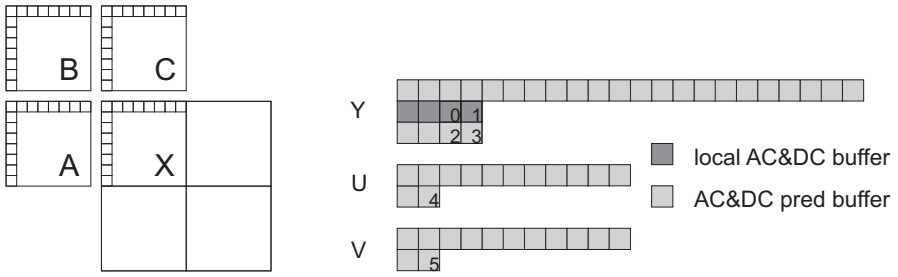


Figure 5.8: The texture coding process works column-based.

The motion compensation and texture update (+ operation in Figure 5.7) are merged with the block-based texture coding loop to improve locality. The presence of the buffer YUV (Figure 5.6) in the selected memory hierarchy allows the immediate storage of newly reconstructed blocks in the frame memory. This reconstructed frame memory also has a block-based data organization to enable burst oriented reads and writes. The updating is only performed if the content of the block has changed (i.e. blocks of the 'skipped' type with zero motion vectors are not copied in the frame memory).

AC&DC prediction tries to code the DC coefficient and some AC coefficients



(a) Neighboring blocks required for AC&DC prediction of block X.

(b) The combination of a local buffer with the reduced AC&DC prediction buffer provides the required neighboring samples

Figure 5.9: The use of a small local buffer reduces the AC&DC prediction buffer size.

in the frequency domain (in the first row or the first column) of an intra macroblock more efficiently, by exploiting the values of the neighboring blocks (Figure 5.9(a)). For every block, these fifteen coefficients need to be stored in memory to be able to complete this prediction process. When the neighboring blocks belong to an inter macroblock, default settings are used for the values of the DC and AC coefficients. By introducing a local buffer of 4×15 elements, the size of the AC&DC prediction buffer can be restricted to $(w_{MB} + 2) \times 4 \times 15$ (Figure 5.9(b)), with $w_{MB} = \frac{width}{16}$ (the width expressed as number of macroblocks). All data, required to predict the coefficients of the blocks (marked with 0 to 5) of the currently processed macroblock, is available in the local buffer or AC&DC prediction buffer. By using a third buffer containing an inter flag per macroblock in the AC&DC prediction buffer, the storing and reading back of default coefficients can be avoided. This drastically decreases the accesses from/to the AC&DC prediction buffer, as most MBs in a video sequence are inter coded (Figure 6.2).

5.2.3 High-Level Optimization Results

The main figures of merit, aimed for during the complete memory centric optimization process, are: (i) the total amount of memory accesses, (ii) the memory footprint (i.e. the accumulated memory size) and (iii) the locality of the accessed data. Additionally, the benefit of the memory optimizations measured on a generic computer platform is used as sanity check to validate the improved efficiency of the (optimized) functional model. This cross check allows using the profiling, completed with ATOMIUM/Analysis data, as first step in the system architecture development. Finally, an important side effect is the reorganization and clean-up of the C source code, resulting in a reduced

number of code lines and hence a better starting point for the functional model, required during the hardware development process.

5.2.3.1 Memory Access Frequency and Peak Memory Usage

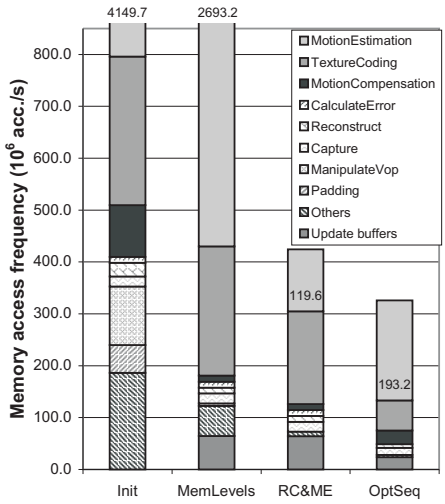
The practical implementation order of the optimizations is different from the one presented above. The algorithmic tuning and memory optimizations are jointly implemented on the video encoder. The evolution of the access distribution during the different optimization steps is shown over the main functional blocks in Figure 5.10(a) and over the different memory size categories in Figure 5.10(b). The labels used in these figures correspond to different software versions and are defined as follows:

1. Init: The initial encoder has a frame-based dataflow (Section 5.1).
2. MemLevels: The introduction of a memory hierarchy redirects the ME frame accesses to large buffers. This extra storage of the previously reconstructed data makes one frame memory sufficient to complete the encoding process and consequently the internal copying (Manipulate frame and Padding of Table 5.3) becomes superfluous (Subsection 5.2.2.1).
3. RC&ME: The algorithmic tuning of the ME minimizes the number of evaluation steps in the motion vector search, resulting in a compacted ME share in Figure 5.10(a) and less large buffer accesses in Figure 5.10(b). The predictive RC enables a true macroblock-based dataflow (less frame accesses in Figure 5.10(b)) (Subsection 5.2.1).
4. OptSeq[1]: The intelligent block processing, together with the rest of the optimizations, further reduce the texture coding contribution and continue favoring buffers and registers (Subsections 5.2.1.1 and 5.2.2.2).

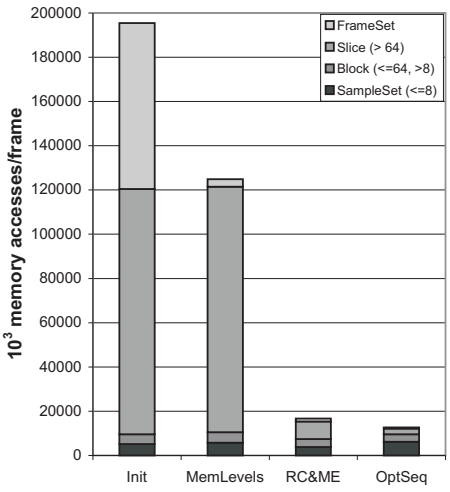
Most of the optimized encoder accesses are to sampleSets (48 %) and blocks (27 %). 20 % of the transfers are to slices, and the frameSets take 5 %. The motion estimation/compensation and texture coding consume almost all transfers.

Table 5.4 lists the total effect of the high-level optimizations on a selection of the testbench. Typically, the overall access frequency is reduced with almost a factor 15. The largest decrease is obtained for accesses to large memories. Almost 2 billion accesses per second are required to encode a 4CIF sequence

[1]The increase of the ME accesses in the OptSeq bar of Figure 5.10(b) is due to the implementation of data reuse at the buffer and register level. The error calculation is merged inside the MC in the OptSeq version.



(a) Evolution of the memory access distribution over the main functional blocks. The motion estimation part has been truncated for the first two steps. The number indicates the complete share.



(b) Evolution of the memory access distribution over different memory sizes. The total amount of accesses is reduced with more than a factor 15 and small memories are favored to restrict the associated cost.

Figure 5.10: Evolution of the memory accesses during the high-level optimization process using the for CIF 450 kbps test case.

Table 5.4: Effect of all high-level optimizations on the access frequency and the peak memory

Test case	Access Frequency (10^6 acc./s)			Peak memory usage (kB)		
	Initial	OptSeq	Reduction factor	Initial	OptSeq	Reduction factor
M&D QCIF 75 kbps 30 fps	1310.0	85.2	15.4	2816.7	131.9	21.4
M&D CIF 200 kbps 30 fps	5554.8	300.0	18.5	5635.7	354.6	15.9
Foreman QCIF 150 kbps 25 fps	1145.2	88.8	12.9	2969.6	131.0	22.7
Foreman CIF 450 kbps 25 fps	4945.8	326.1	15.2	5656.7	354.7	15.9
City CIF 1 Mbps 30 fps	5999.6	465.4	12.9	6850.6	354.7	19.3
City 4CIF 4 Mbps 30 fps	23444.4	1829.8	12.8	34800.7	1245.7	27.9
Crew CIF 1 Mbps 30 fps	6206.0	452.0	13.7	5645.2	354.7	15.9
Crew 4CIF 4 Mbps 30 fps	25654.0	1777.9	14.4	32727.1	1245.6	26.3
C&M QCIF 300 kbps 10 fps	520.3	46.6	11.2	2982.6	131.9	22.6
C&M CIF 1 Mbps 10 fps	2092.2	176.9	11.8	5666.8	354.7	16.0

at 30 fps. The peak memory usage denotes the maximum amount of memory allocated by the instrumented arrays during the execution of the program on a test case. It is an estimated lower bound for the actual memory usage (as scalars are not included). The macroblock-based encoder reduces the amount of memory with over a factor 15, to 132 kB for QCIF, 355 kB for CIF and 1.2 MB for 4CIF. Table B.3 contains the measurements for the complete testbench.

5.2.3.2 Performance

Testing the performance on an HP9000/J7000 RISC processor, running at 440 MHz with UNIX as operating system (Table 5.5), evaluates the effect of the memory optimizations and algorithmic tuning on the efficiency of the optimized specification. The measured speed up factor, ranging from 7 to almost 13, validates this improvement on a generic processor, used as test platform for this cross check. Note that real-time performance is not yet achieved (on this test platform). Results for the complete testbench are listed in Table B.4

5.2.3.3 Code Size Reduction

ATOMIUM pruning extracts the functionality corresponding to the selected profile (see Section IV). Further manual code reorganization and rewriting shrinks the number of lines with a factor of 7.6, compared to the original (Table 5.6). This last reduction is obtained by flattening the hierarchical function structure and by further simplification of the functionality, thanks to memory optimizations.

Table 5.5: Performance improvement of the high-level optimized encoder on a RISC processor at 180 MHz.

Test case	Initial (fps)	OptSeq (fps)	Speed up factor
M&D QCIF 75 kbps 30 fps	1.1	12	10.3
M&D CIF 200 kbps 30 fps	0.26	3.3	12.7
Foreman QCIF 150 kbps 25 fps	1.1	9.3	8.6
Foreman CIF 450 kbps 25 fps	0.25	2.6	10.4
City CIF 1 Mbps 30 fps	0.24	2.1	8.6
City 4CIF 4 Mbps 30 fps	0.060	0.53	8.8
Crew CIF 1 Mbps 30 fps	0.24	2.1	8.8
Crew 4CIF 4 Mbps 30 fps	0.058	0.54	9.4
C&M QCIF 300 kbps 10 fps	0.91	6.5	7.1
C&M CIF 1 Mbps 10 fps	0.23	1.7	7.7

Table 5.6: Encoder code size reduction after automatic pruning and manual reorganization

Code version	Number of lines	Reduction
Initial	67k	
Pruned	22k	3.1
Optimized	9k	7.6

5.3 Partitioning

The partitioning design step first construct a CSDF graph of the high-level optimized video encoder. This model is then used in a lifetime analysis to correctly size the buffers of the channels between the actors.

5.3.1 CSDF Graph

Starting from an analysis of the dataflow of the optimized encoder in Figure 5.7 and a complexity evaluation of the functional modules (Figure 5.10(a)), a partitioning enabling a functional pipeline is selected. The task borders are selected so that: (i) the control flow between them is minimized, (ii) the communicated data containers are small, (iii) the related functional behavior is grouped, and (iv) the computation complexity is distributed. As a result of this strategy and of the localization introduced by the memory optimizations, the tasks correspond to the main algorithmic building blocks of the video encoder. The construction of the video pipeline is made incrementally and is supported by SPRINT to gradually split off an increasing number of processes from the complete encoder. Once the amount of functionality contained in each process composing the system allows its custom HW implementation,

the obtained partitioning is represented as CSDF graph (Figure 5.11), using the special channels of Chapter 3.

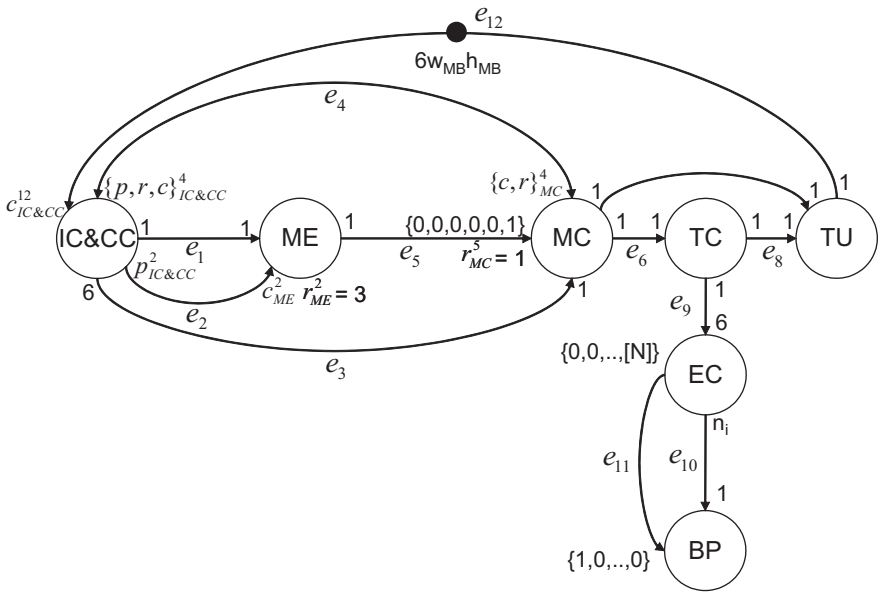


Figure 5.11: CSDF graph representing the partitioning of the MPEG-4 part 2 SP encoder scheme.

This dataflow graph is a combination of a CSDF graph with compile time parameters related to the maximum supported resolution ($256(w_{MB} \times h_{MB})$) and a DDF part after the Entropy Coding (EC). The meaning of the variables m and N in the graph relate to this dynamic behavior and will be explained later. The regular and special channels of Chapter 3 are used in the dataflow graph as short-hand notation: every drawn edge represents a forward and a backward CSDF edge (to model the bounded buffer sizes). Additionally, a self-cyle with one initial token is assumed on every actor.

The proposed encoder CSDF graph (Figure 5.11) consists of 7 actors connected by 12 edges numbered e_1 to e_{12}. Three edges, e_2, e_4 and e_5 are special channels. Edge e_2 is a non-destructive read special channel, modeling the sliding window with the search area data repetitively accessed by the motion estimation. Edge e_4 is drawn as a bidirectional edge, as it represents non-destructive read back behavior at the producer side, sharing data between the copy controller and the motion compensation. Edge e_5 is a non-destructive read special channel, passing the motion vectors to the motion compensation. These motion vectors are reused for the six blocks of the macroblock. Edge e_{12} is a regular channel with initial tokens (represented by the full dot and the number of initial tokens).

Table 5.7 details for every actor its full name, functionality and actor period.

Table 5.7: Detailed information of the actors in the encoder CSDF graph

Actor name	Acronym	Functionality	Cycle length L
Input Control and Copy Control	IC&CC	Fill the memory hierarchy and the new video inputs	$w_{MB}h_{MB}$
Motion Estimation	ME	Find the motion vectors	w_{MB}
Motion Compensation	MC	Get predicted block and calculate error	$6w_{MB}h_{MB}$
Texture Coding	TC	Transform, quantization and inverse	1
Texture Update	TU	Add and clip compensated and predicted blocks	1
Entropy Coding	EC	AC/DC, MV prediction and VLC coding	m
Bitstream Packetization	BP	Add headers and compose the bitstream	N

The production/consumption sequences reflect the behavior of the video encoder. They are represented as compactly as possible in Figure 5.11 due to the long actor periods: (i) if the sequence contains a repeated pattern, only this pattern is listed and (ii) a symbolic representation is used if the cycle of the sequence spans more than 6 phases. For instance e_2 has a cycle length equal to w_{MB}. Equations (5.1) and (5.2) define respectively its production and consumption sequence. Details on the sequences at producer and consumer side of edges e_4 and e_{12} are given in Equations (A.1) to (A.6).

$$p^2_{IC\&CC}(h) = \begin{cases} 3 & \text{if } h = 0 \\ 1 & \text{otherwise.} \end{cases} \tag{5.1}$$

$$c^2_{ME}(h) = \begin{cases} 3 & \text{if } h = w_{MB} - 1 \\ 1 & \text{otherwise.} \end{cases} \tag{5.2}$$

After the entropy coding, the number of generated bits varies depending on the type of sequence and the quantization degree. Edges e_{10} and e_{11} cooperate in a special way to deal with this. The compressed information is accumulated on edge e_{10} with the number of bits n_i varying per firing of the actor EC. When the size of a video packet is reached after m firings, the amount of bits $N = \sum_{i=0}^{m-1} n(i)$ accumulated on edge e_{10} is written on edge e_{11}. Once this is completed, actor BP can fire and consumes 1 token from edge e_{11}, containing a scalar with the total number of tokens to consume from edge e_{10}, resulting in N firings of BP, that consume 1 token from e_{10}. As the maximum number of bits allowed in a video packet is defined by the levels of the MPEG-4 part 2 standard, the buffer size of edge e_{10} is bounded.

Having all this information allows the construction of the topology matrix Γ, with the ordering of the actors (column number) like in Table 5.7. Row numbers correspond with the edge numbers of Figure 5.11. For instance, the

first row corresponds with edge e_1. Actor IC&CC produces $w_{MB}h_{MB}$ tokens during one IC&CC actor period (a complete frame), while actor ME consumes w_{MB} during one ME actor period (a row of a frame).

$$\Gamma = \begin{pmatrix} w_{MB}h_{MB} & -w_{MB} & 0 & 0 & 0 & 0 & 0 \\ (w_{MB}+2)h_{MB} & -(w_{MB}+2) & 0 & 0 & 0 & 0 & 0 \\ 6w_{MB}h_{MB} & 0 & -6w_{MB}h_{MB} & 0 & 0 & 0 & 0 \\ w_{MB}h_{MB} & 0 & -w_{MB}h_{MB} & 0 & 0 & 0 & 0 \\ 0 & w_{MB} & -w_{MB}h_{MB} & 0 & 0 & 0 & 0 \\ 0 & 0 & 6wMBh_{MB} & -1 & 0 & 0 & 0 \\ 0 & 0 & 6wMBh_{MB} & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -6m & 0 \\ 0 & 0 & 0 & 0 & 0 & N & -N \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ -6w_{MB}h_{MB} & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

The actor periods are represented in matrix L in the same ordering as Table 5.7

$$L = \begin{pmatrix} w_{MB}h_{MB} & w_{MB} & 6w_{MB}h_{MB} & 1 & 1 & m & N \end{pmatrix}$$

L_{diag} is the diagonal version of L, required to calculate the basic repetition vector q^b. The rank of $\Gamma = 6$, indicates that there is a solution for the balance equation. Solving this balance equation $\Gamma \cdot r^T = 0$ yields as period balance vector

$$r = \begin{pmatrix} m & mh_{MB} & m & 6mw_{MB}h_{MB} & 6mw_{MB}h_{MB} & w_{MB}h_{MB} & w_{MB}h_{MB} \end{pmatrix}$$

with m the number of firings of actor EC to reach a complete video packet. The repetition vector $q = r \cdot L_{diag}$ equals

$$q = mw_{MB}h_{MB} \begin{pmatrix} 1 & 1 & 6 & 6 & 6 & 1 & \frac{N}{m} \end{pmatrix}$$

The basic repetition vector q^b is identical to the repetition vector q.

In this basic repetition vector q^b, the actors MC, TC and TU, working on a block basis, execute 6 times, while the actors IC&CC, ME and EC processing macroblocks only fire once. The fractional repetition rate of the BP is due to the compression in the EC and shows that actor BP only fires when all data, required to build a video packet, is available.

5.3.2 Buffer Capacity Calculation

The calculation of the buffer sizes starts with identifying the chains and clusters (see Chapter 3) in the encoder CSDF graph of Figure 5.11. As the feedback loop through e_{12}, representing the reconstructed frame, contains the same amount of initial tokens as consumed during the period of actor IC&CC, this edge can first be safely removed to simplify the graph. This makes actor IC&CC the source of the simplified CSDF graph, and actors TU and BP the sinks. In this simplified graph, there is only one chain: (TC,EC), and there are 3 clusters: (IC&CC,ME,MC), (MC,TC,TU) and (EC,BP). The buffer sizes of their edges can be calculated independently (see Chapter 3 and [10]).

The next three subsections describe the buffer sizing of the (IC&CC,ME,MC) cluster and the (EC,BP) cluster and the addition of the feedback edge. The calculations for the (MC,TC,TU) cluster and the (TC,EC) chain is respectively made in Subsections A.2.2 and A.2.3. Subsection 5.3.2.4 summarizes the results.

5.3.2.1 IC&CC,ME and MC Cluster

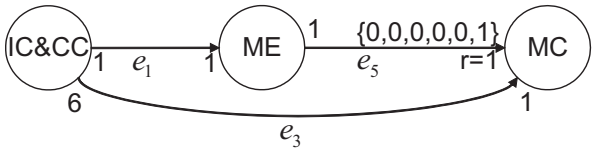


Figure 5.12: Simplified IC&CC, ME and MC cluster.

The (IC&CC,ME,MC) cluster is further simplified (Figure 5.12) by removing edge e_2 and e_4, as they connect the same actors in the same direction as respectively e_1 and e_3. After calculating the buffer sizes of the simplified graph, the capacity of the removed edges can be sized separately, by respecting the same firing order of their producer and consumer, like in the simplified cluster.

Table 5.8 lists the required buffer sizes to allow the firing of the producers and the buffer occupancy after the consumption at the end of the firing of the edges in the simplified (IC&CC,ME,MC) cluster, using the balancing principle of Chapter 3. It spans for every actor at least a number of firings equal to its cluster basic repetition rate. In the simplified cluster, the basic repetition rates of the IC&CC and ME equals 1, the repetition rate of the MC is 6. The response times are expressed relatively to the IC&CC source actor, using the basic repetition rates of q^b. Edges e_1 and e_5 need to be able to contain 2 tokens while edge e_3 can hold 18 tokens.

Adding edge e_2 to the simplified (IC&CC,ME,MC) cluster allows its dimen-

Table 5.8: Tracking the required buffer size of the edges in the simplified (IC&CC,ME,MC) cluster.

Actor firing	Time (RT)		# on e_1		# on e_3		# on e_5	
	t_{start}	t_{end}	req	end	req	end	req	end
IC&CC	0	1	1	1	6	6	0	0
IC&CC ‖ ME	1	2	2	1	12	12	1	1
IC&CC ‖ ME ‖ MC	2	$2+\frac{1}{6}$	2	-	18	11	2	-
... ‖ ... ‖ MC	$2+\frac{1}{6}$	$2+\frac{2}{6}$	2	-	18	10	2	-
... ‖ ... ‖ MC	$2+\frac{2}{6}$	$2+\frac{3}{6}$	2	-	18	9	2	-
... ‖ ... ‖ MC	$2+\frac{3}{6}$	$2+\frac{4}{6}$	2	-	18	8	2	-
... ‖ ... ‖ MC	$2+\frac{4}{6}$	$2+\frac{5}{6}$	2	-	18	7	2	-
... ‖ ... ‖ MC	$2+\frac{5}{6}$	3	2	1	18	12	2	1
maximum			2		18		2	

sioning. The extended cluster has increased cluster basic repetition rates, due to the w_{MB} period of the production and consumption sequences of e_2. Table 5.9 lists the usage of edge e_2, respecting the firing order of the simplified cluster, during w_{MB} firings of actors IC&CC and ME and $6w_{MB}$ firings of MC. Maximally, space for 6 tokens is needed.

Table 5.9: Tracking the required buffer size of edge e_2 (search area).

Actor firing	Time (RT)		# on e_2	
	t_{start}	t_{end}	req	end
IC&CC	0	1	3	3
IC&CC ‖ ME	1	2	4	3
...				
IC&CC ‖ ME	$w_{MB}-1$	w_{MB}	4	3
IC&CC ‖ ME	w_{MB}	$w_{MB}+1$	6	3
maximum			6	

Table 5.10 lists the usage of edge e_4, with $h'_{MB} = h_{MB} - 1$ as short notation, by adding it to the simplified (IC&CC,ME,MC) cluster and respecting its original firing order. In this extended cluster, the cluster basic repetition rates of the IC&CC, ME and MC respectively increase to $w_{MB}h_{MB}$,$w_{MB}h_{MB}$ and $6w_{MB}h_{MB}$. Maximally, space for $2w_{MB}+5$ tokens is needed.

As edge e_4 is a multiple consumer with non-destructive reads special channel, the two conditions of Equation (3.34), for safely reading back at the consumer side, need to be evaluated. Table A.1 confirms both conditions are met at every firing of a sequential schedule, that starts the consuming actor as soon its firing rule is true. It is assumed that actor IC&CC first writes on edge e_4 and can read this data back during the same firing.

Table 5.10: Tracking the required buffer size of edge e_4 (buffer YUV).

Actor firing	Time (RT)		# on e_4	
	t_{start}	t_{end}	req	end
IC&CC	0	1	$w_{MB}+2$	$w_{MB}+2$
IC&CC ‖ ME	1	2	$w_{MB}+3$	$w_{MB}+3$
IC&CC ‖ ME ‖ 6MC	2	3	$w_{MB}+4$	$w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	3	4	$w_{MB}+5$	$w_{MB}+5$
... continue until IC&CC reaches one but last element on the first row				
IC&CC ‖ ME ‖ 6MC	$w_{MB}-2$	$w_{MB}-1$	$2w_{MB}$	$2w_{MB}$
IC&CC ‖ ME ‖ 6MC	$w_{MB}-1$	w_{MB}	-	$2w_{MB}$
IC&CC ‖ ME ‖ 6MC	w_{MB}	$w_{MB}+1$	$2w_{MB}+2$	$2w_{MB}+2$
IC&CC ‖ ME ‖ 6MC	$w_{MB}+1$	$w_{MB}+2$	$2w_{MB}+3$	$2w_{MB}+3$
IC&CC ‖ ME ‖ 6MC	$w_{MB}+2$	$w_{MB}+3$	$2w_{MB}+4$	$2w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	$w_{MB}+3$	$w_{MB}+4$	$2w_{MB}+5$	$2w_{MB}+4$
... continue until IC&CC reaches one but last element on the current row				
IC&CC ‖ ME ‖ 6MC	$2w_{MB}-2$	$2w_{MB}-1$	$2w_{MB}+5$	$2w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	$2w_{MB}-1$	$2w_{MB}$	-	$2w_{MB}+3$
IC&CC ‖ ME ‖ 6MC	$2w_{MB}$	$2w_{MB}+1$	$2w_{MB}+5$	$2w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	$2w_{MB}+1$	$2w_{MB}+2$	$2w_{MB}+5$	$2w_{MB}+3$
IC&CC ‖ ME ‖ 6MC	$2w_{MB}+2$	$2w_{MB}+3$	$2w_{MB}+4$	$2w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	$2w_{MB}+3$	$2w_{MB}+4$	$2w_{MB}+5$	$2w_{MB}+4$
... continue until IC&CC reaches one but last element on the one but last row				
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}-2$	$h'_{MB}w_{MB}-1$	$2w_{MB}+5$	$2w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}-1$	$h'_{MB}w_{MB}$	-	$2w_{MB}+3$
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}$	$h'_{MB}w_{MB}+1$	-	$2w_{MB}+2$
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}+1$	$h'_{MB}w_{MB}+2$	-	$2w_{MB}$
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}+2$	$h'_{MB}w_{MB}+3$	-	$2w_{MB}$
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}+3$	$h'_{MB}w_{MB}+4$	-	$2w_{MB}-1$
... continue until IC&CC reaches one but last element on the last row				
IC&CC ‖ ME ‖ 6MC	$h_{MB}w_{MB}-2$	$h_{MB}w_{MB}-1$	-	$w_{MB}+4$
IC&CC ‖ ME ‖ 6MC	$h_{MB}w_{MB}-1$	$h_{MB}w_{MB}$	-	$w_{MB}+3$
ME ‖ 6MC	$h_{MB}w_{MB}$	$h_{MB}w_{MB}+1$	-	$w_{MB}+2$
6MC	$h_{MB}w_{MB}+1$	$h_{MB}w_{MB}+2$	-	0
maximum			$2w_{MB}+5$	

5.3.2.2 EC and BP Cluster

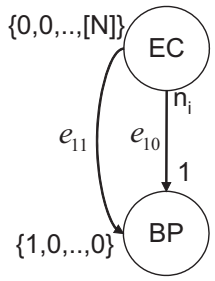


Figure 5.13: EC and BP cluster.

In the (EC,BP) cluster (Figure 5.13), the basic repetition rates of the EC and BP are respectively m and N. The relative response times (derived using the basis repetition rates of q^b) are respectively 1 and $\frac{m}{N}$. Assuming that after every m firings of EC, N tokens are produced (or $N = \sum_{i=0}^{m-1} n(i)$), the size of edge e_{10} can be determined under worst case conditions, given the maximum allowed video packet size (VP_{max}) in the MPEG-4 part 2 levels. When no compression is achieved, n_{max} bits are needed per macroblock and VP_{max} is then reached after a minimum m_{min} firings of EC. During a response time of 1, EC generates those n_{max} tokens, while the BP with a response time of $\frac{m_{min}}{VP_{max}}$ can fire $\frac{VP_{max}}{m_{min}} = n_{max}$ times, each time consuming 1 token. Hence in Table A.3, setting $\alpha = n(m)$ leads to a maximum buffer size of edge e_{10} equaling $N + n_{max}$.

Table 5.11: Tracking the required buffer size of the edges in the (EC,BP) cluster.

Actor firing	Time (RT)		# on e_{10}		# on e_{11}	
	t_{start}	t_{end}	req	end	req	end
EC	0	1	$n(0)$	$n(0)$	-	0
EC	1	2	$n(0) + n(1)$	1	-	0
continue until m-1						
EC	$m-1$	m	N	N	1	1
EC ‖ BP	m	$m + \frac{m}{N}$	$N + n(m)$	$N - 1$	-	0
EC ‖ BP	$m + \frac{m}{N}$	$m + \frac{2m}{N}$	$N + n(m)$	$N - 2$	-	0
...						
EC ‖ BP	$m + \frac{\alpha m}{N}$	$m + 1$	$N + n(m)$	$N + n(m) - \alpha$	-	0
EC ‖ BP	$m + 1$	$m + \frac{(\alpha+1)m}{N}$	$N + n(m) + n(m+1) - \alpha$	$N - n(m) - \alpha - 1$	-	0
...						
EC ‖ BP	$m + \frac{(N-1)m}{N}$	$2m$	$2N - (N-1)$	N	1	1
maximum			?		1	

5.3.2.3 Adding the Feedback Edge

To obtain the complete encoder CSDF graph of Figure 5.11, the usage of the feedback edge e_{12} is evaluated, respecting the already obtained firing order

during the previous buffer calculations. Table 5.12 lists the required buffer size and occupancy, with as simplified notation: $h_{MB}w_{MB} = R$ and $(h_{MB} - 1)w_{MB} = R'$. The amount of initial tokens on this edge is the maximum required.

Table 5.12: Tracking the required buffer size of edge e_{12} (reconstructed frame).

Actor firing										Time (RT)		# on e_{12}	
										t_{start}	t_{end}	req	end
IC&CC										0	1	-	$6(R'-2)$
IC&CC	‖	ME								1	2	-	$6(R'-3)$
IC&CC	‖	ME	‖	MC						2	$2+\frac{1}{6}$	-	$6(R'-3)$
...	‖	...	‖	MC	‖	TC				$2+\frac{1}{6}$	$2+\frac{2}{6}$	-	$6(R'-3)$
...	‖	...	‖	MC	‖	TC	‖	TU		$2+\frac{2}{6}$	$2+\frac{3}{6}$	$6(R'-3)+1$	$6(R'-3)+1$
...	‖	...	‖	MC	‖	TC	‖	TU		$2+\frac{3}{6}$	$2+\frac{4}{6}$	$6(R'-3)+2$	$6(R'-3)+2$
...	‖	...	‖	MC	‖	TC	‖	TU		$2+\frac{4}{6}$	$2+\frac{5}{6}$	$6(R'-3)+3$	$6(R'-3)+3$
...	‖	...	‖	MC	‖	TC	‖	TU		$2+\frac{5}{6}$	3	$6(R'-3)+4$	$6(R'-4)+4$
IC&CC	‖	ME	‖	MC	‖	TC	‖	TU		3	$3+\frac{1}{6}$	$6(R'-4)+5$	$6(R'-4)+5$
...	‖	...	‖	MC	‖	TC	‖	TU		$3+\frac{1}{6}$	$3+\frac{2}{6}$	$6(R'-3)$	$6(R'-3)$
... continue until IC&CC reaches one but last element on the one but last row													
IC&CC	‖	ME	‖	MC	‖	TC	‖	TU		$R'-2$	$R'-2+\frac{1}{6}$	$6(R'-4)+5$	$6(R'-4)+5$
...													
...	‖	...	‖	MC	‖	TC	‖	TU		$R'-2+\frac{5}{6}$	$R'-1$	$6(R'-3)+4$	$6(R'-4)+4$
IC&CC	‖	ME	‖	MC	‖	TC	‖	TU		$R'-1$	$R'-1+\frac{1}{6}$	$6(R'-4)+5$	$6(R'-3)+5$
...													
...	‖	...	‖	MC	‖	TC	‖	TU		$R'-1+\frac{5}{6}$	R'	$6(R'-3)+4$	$6(R'-3)+4$
... continue until IC&CC reaches the last element on the last row													
IC&CC	‖	ME	‖	MC	‖	TC	‖	TU		$R-1$	$R-1+\frac{1}{6}$	$6(R-3)+4$	$6(R'-3)+4$
...													
...	‖	...	‖	MC	‖	TC	‖	TU		$R-1+\frac{5}{6}$	R	$6(R-3)+4$	$6(R-3)+4$
		ME	‖	MC	‖	TC	‖	TU		R	$R+\frac{1}{6}$	$6(R-3)+5$	$6(R-3)+5$
...													
		...	‖	MC	‖	TC	‖	TU		$R+\frac{5}{6}$	$R+1$	$6(R-2)+4$	$6(R-2)+4$
				MC	‖	TC	‖	TU		$R+1$	$R+1+\frac{1}{6}$	$6(R-2)+5$	$6(R-2)+5$
...													
				MC	‖	TC	‖	TU		$R+1+\frac{5}{6}$	$R+2$	$6(R-1)+4$	$6(R-1)+4$
						TC	‖	TU		$R+2$	$R+2+\frac{1}{6}$	$6(R-1)+5$	$6(R-1)+5$
								TU		$R+2+\frac{2}{6}$	$R+2+\frac{2}{6}$	$6R$	$6R$
maximum												$6R$	

5.3.2.4 Resulting Buffer Capacities

The obtained buffer capacities of the edges, enabling full pipelined and parallel operation (balancing principle) of all actors, are summarized in Table 5.13. Also their name, their container size, the width of an element in a container and the communication primitive type, that is selected for the RTL implementation of the next section, is listed.

This edge size information is added to the SPRINT user's pragmas file, to generate a concurrent SystemC model from the optimized code, allowing the verification of the desired blocking write and blocking read behavior by changing the task cycle estimations (see Chapter 4). If the capacity of the buffers of one of the edges is calculated wrongly, this will result in: (i) a different output bitstream for the shared memories and (ii) a stall or a non-parallel behavior of the tasks for the FIFO queues. Correct capacities of all edge buffers result in

Table 5.13: Buffer capacities and container sizes with the corresponding CP type of all edges.

Edge	Name	Buffer capacity (# of containers)	Container size (words/ container)	Container width (bits/ word)	CP type
e_1	new macroblock	2	256	8	block FIFO
e_2	search area	6	768	8	shared memory
e_3	current macroblock	18	64	8	block FIFO
e_4	buffer YUV	$2w_{MB}+5$	384	8	shared memory
e_5	motion vectors	2	2	12	scalar FIFO
e_6	error block	2	64	9	block FIFO
e_7	compensated block	3	64	8	block FIFO
e_8	texture block	2	64	12	block FIFO
e_9	quantized macroblock	12	64	12	block FIFO
e_{10}	data buffer	$VP_{max}+n_{max}$	1	1	scalar FIFO
e_{11}	generate VP	1	1	1	scalar FIFO
e_{12}	reconstructed frame	$6w_{MB}h_{MB}$	64	8	shared memory

a concurrent, self-timed system, generating an exact output bitstream with all tasks running according to the desired schedule (i.e. fully pipelined and parallel). Figure 5.14 is an activity diagram of a simulation, using the SystemC model with cycle estimates of the HW that will be developed. The starting up of the pipeline and its parallel operation in regime mode is shown.

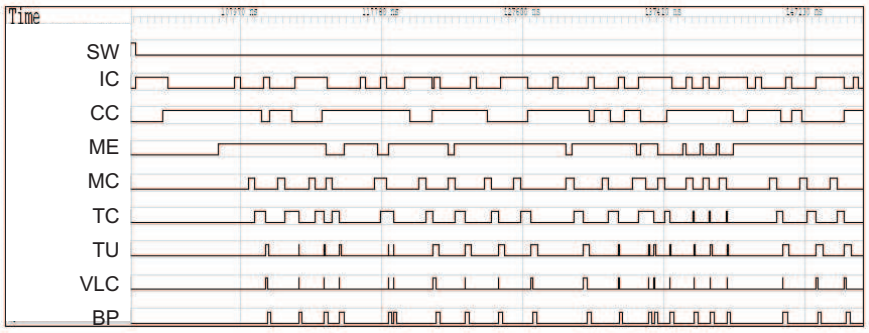


Figure 5.14: High-level simulation of the partitioned encoder, using the SystemC model with cycle estimations to verify correctness of the buffer capacities.

To compare the gain of the special channels, section C.3 dimensions edges e_2, e_4 and e_5 by replacing them by their strict CSDF equivalent. The required size of Table 5.14 includes the required space in the internal state and the relative increase is given for 4CIF resolution. Up to 50% extra buffer space is required. Additionally, the reused data needs to be copied to this internal

state, resulting in extra data transfers. The number of container copies per channel per frame is listed in Table 5.14.

Table 5.14: Buffer space increase of the encoder with CSDF equivalents.

Edge	Extension size	Equivalent size	Increase (%)	Copies
e_2	6	6	0	$3w_{MB}h_{MB}$
e_4	$2w_{MB}+5$	$3w_{MB}+6$	48.4	$2w_{MB}h_{MB}$
e_5	2	3	50.0	$w_{MB}h_{MB}$

5.4 RTL Development and Verification

Each actor of the encoder CSDF graph is now translated separately to an HDL description, using the RTL development and verification environment of Chapter 4. Most actors result in a single functional component, only the IC&CC one is split in two, to allow easy updating of the input control to other source formats, without affecting the copy control.

Based on previous experience, the target operation frequency of all components of this design is set to 100 MHz. The required throughput is 4CIF at 30 fps or 47520 macroblocks per second. The maximum width is set to 720 pixels to also support the processing of standard definition sequences. For this throughput and operation frequency, 2104 cycles are available for actors working on a macroblock basis. Using Equation (3.36) and the basic repetition rates of q^b, IC, CC, ME and EC have this cycle budget of 2104 cycles per firing. MC, TC and TU, working on a block basis, have one sixth of this budget, or 350 cyles per firing. At MPEG-4 part 2 simple profile video level 5, VP_{max} equals 16384 bytes [9]. In worst case conditions, BP has 0.7 cycles per firing.

Under these constraints, the designer has to come up with a suitable architecture and schedule for each functional component and describe it in HDL (in this case VHDL). For efficiency reasons, some of the decisions made, based on the CSDF model, are refined. For instance, the bitwise communication between the EC and BP is extended to a byte basis to achieve a higher throughput. Also the freedom offered by the shared memory is further exploited. To avoid a too long response time of CC when processing the first macroblock of a frame, only 4 of its $w_{MB}+2$ macroblocks are written in the bufferYUV, as the MC can start with only this data. The other macroblocks are written during the further processing of the first row.

The final video encoder architecture (Figure 5.15) contains, next to the communication channels of Table 5.13, some extra scalar FIFOs to transfer parameters on a (macro)block basis. Scalar FIFOs are indicated using dash-dotted

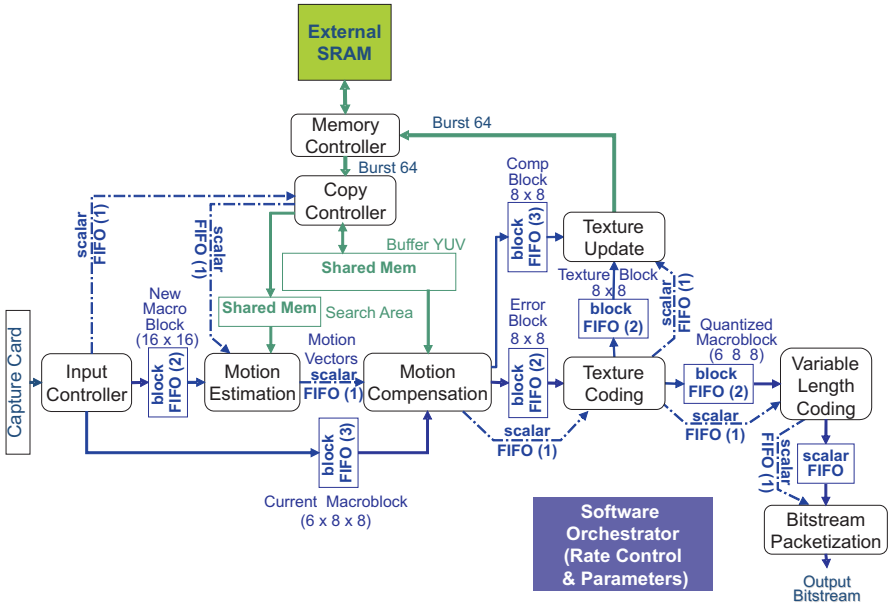


Figure 5.15: The MPEG-4 SP encoder architecture. The FIFO queues and the search area are L1 memory, bufferYUV is L2 and the external memory is L3.

lines, block FIFOs with dark boxes and full lines and shared memories with light boxes and lines. The number between brackets next to the FIFO queues gives their depth, while the number next to the name of the channel is the container size. Note that the depth of a scalar FIFO equals the calculated buffer capacity minus 1 (see Section 3.5).

The applied memory optimizations minimize the size of the containers in the queues and their data transfer rate. These FIFO queues together with the search area buffer are the L1 memory. The bufferYUV resides in L2. The introduced memory hierarchy enables exploiting high performance burst oriented external memory (L3), interfacing through a memory controller. The communication primitives, used in the selected video pipeline, avoid the need for a fast bus, that is typically present in hybrid implementations [26].

The resulting video pipeline has a self-timed behavior. The concurrency of its processes is assured by correctly sizing these communication primitives. In this way, the complete pipeline behaves like a monolithic hardware accelerator. To avoid interface overheads [95], the software orchestrator calculates the configuration settings (parameters, like the quantization parameter, frame width, frame height, etc.) for all functional modules on a frame basis.

After designing each functional component separately to meet the design constraints and verifying it carefully, using the combined simulation and fast

prototyping approach of Chapter 4, they are gradually composed to form the complete encoder.

5.5 Implementation Results

The resulting MPEG-4 part 2 SP encoder implementation is first mapped on the Xilinx Virtex-II 3000 (XC2V3000-4) FPGA, available on the Wildcard-II [3] used as prototyping/demonstration platform during verification. Then the Synopsys tool suite is combined with Modelsim to evaluate the power efficiency and size of an ASIC implementation in a 180 nm, 1.62V UMC technology.

5.5.1 Throughput and Size

Table 5.15 lists required operation frequency to sustain the throughput of different MPEG-4 SP levels. The current design can be clocked up to 100 MHz both on the FPGA [2] and on the ASIC, supporting 30 4CIF frames per second, exceeding the Level 5 requirements of the MPEG standard [9]. Additionally, the encoder core supports processing of multiple video sequences (for instance 4×30 CIF frames per second). The user can specify the required maximum frame size, through the use of HDL generics, to scale the design to his needs (Table 5.16).

Table 5.15: Required operation frequency and off-chip data rates, encoding the *City* reference video sequence

Throughput (fps) & Level	Operation Frequency (MHz)		external memory (kB)	32-bit external transfers ($10^6/s$)	
	measured	not optimized[3]		measured	not optimized[4]
15 QCIF (L1)	3.2	4.0	37	0.28	0.29
15 CIF (L2)	12.9	17.6	149	1.14	1.14
30 CIF (L3)	25.6	35.2	149	2.25	2.28
30 4CIF (>L5)	100.7	140.6	594	9.07	9.12

Figure 5.18 of Subsection 5.5.4 confirms the high degree of concurrency achieved in the video pipeline, i.e. no stalls occur in the pipeline. Compared to state of the art (Figure 5.16(a)), the proposed solution achieves a high throughput at a low clock frequency (i.e. a larger degree of parallelism), while having the

[2]Frequency Achieved for Virtex4 speed grade -10. Implementation on Virtex2 or Spartan3 may not reach this operating frequency.

[3]not optimized = proposed ME algorithm (directional squared search) without early stop criteria

[4]not optimized = reading and writing every sample once

Table 5.16: Scaling of the FPGA resources at different MPEG-4 part 2 levels

Throughput	Level	Slices	LUTs	FFs	BRAMs	Mults/DSP48
15 QCIF	L1	8303	12630	6093	16	17
15 CIF	L2	9019	12778	6335	23	17
30 CIF	L3	9019	12778	6335	23	17
30 4CIF	>L5	9353	13260	6575	36	17

lowest logic gate count (Table 5.18). This is a result of the problem simplification (i.e. the algorithmic tuning and memory optimizations) applied at the high-level and of the partitioning strategy. The recent implementations of Yamauchi [114], Watanabe [109] and Lin [73] provide an even better throughput versus operation frequency trade off at the cost of a higher area. To solve the ME critical path, [73, 114] use around 16 sum of absolute difference processing engines, while our directional squared search (Subsection 5.2.1.2) only needs 3.

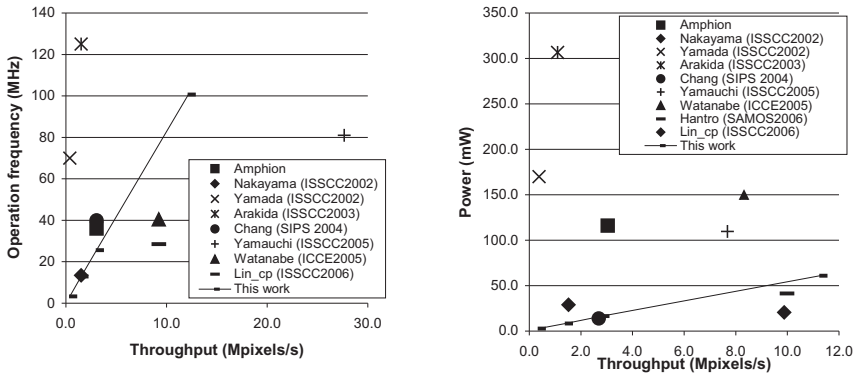
Note that next to the lowest logic gate count (Table 5.18), this work uses a significant amount of on-chip memory to avoid costly external memory accesses.

5.5.2 Memory requirements

On-chip BRAM (FPGA) or SRAM (ASIC) is used to implement the memory hierarchy and the required amount scales with the maximum frame size (Table 5.16). Both the copy controller (filling the bufferYUV and search area, see Figure 5.15) and the texture update make 32 bit burst accesses (of $16 \times 32\text{bit} = 64$ bytes) to the external memory, holding the reconstructed frame with a block-based data organization. At 30 4CIF frames per second, this corresponds in worst case to 9.2 Mtransfers per second (as skipped blocks are not written to the reconstructed frame, the values of the measured external transfers in Table 5.15 are lower). In this way, our implementation minimizes the off-chip bandwidth with at least a factor of 2.5, compared to [14, 24, 81, 114] without embedding a complete frame memory, as done in [15, 113] (see also Table 5.18). Additionally, our encoder only requires the storage of one frame in external memory.

5.5.3 Power Consumption

Running power simulations assesses the power efficiency of the proposed implementation. They consist of 3 steps: 1. Synopsys [4] DC Compiler generates a gate level netlist and the list of signals to be monitored for power (forward switching activity file). 2. ModelSim [5] RTL simulation tracks the actual



(a) Operation frequency

(b) Power consumption

Figure 5.16: Power and operation frequency versus throughput comparison of different MPEG-4 part 2 video encoders

toggles of the monitored signals and produces the back-annotated switching activity file. 3. Synopsys Power Compiler calculates power numbers based on the gate level netlist from step 1 and back-annotated switching activity file from step 2. Such pre-layout gate-level simulations do not include accurate wire-loads. Internal experiments indicate their impact is limited to a 20 % error margin. Additionally, I/O power is not included.

Table 5.17: Hardware characteristics of the encoder core

Process	UMC 180 nm Voltage 1.62 V
Area	3.1 x 3.1 mm2
Gate size	88 kGates + 400 kbit SRAM
off-chip memory	600 Kbyte
Power consumption	3.2 mW QCIF 15 fps (L1)
	9.7 mW CIF 15 fps (L2)
	19.2 mW CIF 30 fps (L3)
	71 mW 4CIF 30 fps (>L5)

Table 5.17 gives the characteristics of the ASIC encoder core when synthesized for 100 MHz, 4CIF resolution and lists the power consumptions at different levels, when clocked at the corresponding operation frequency of Table 5.15, while processing the City reference video sequence.

Carefully realizing the communication primitives on the ASIC allows balancing their power consumption compared to the logic (Figure 5.17): banking is applied to the large on-chip bufferYUV and the chip-enable of the communication primitives is precisely controlled to shut down the CP whenever it is

idle. Finally clock-gating is applied to the complete encoder to further reduce the power consumption[5].

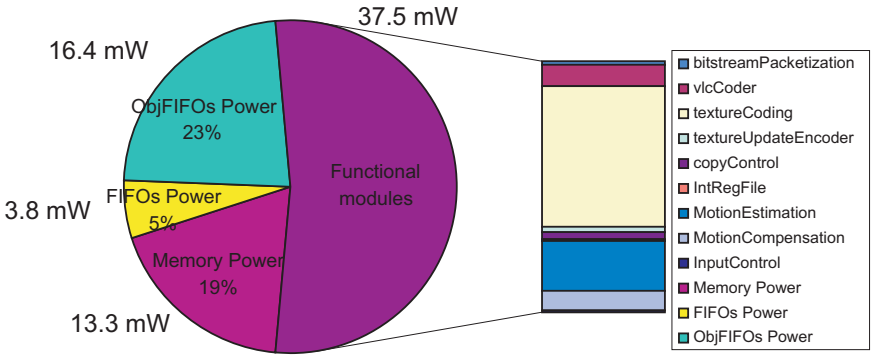


Figure 5.17: The power is equally distributed over logic and communication primitives.

To compare the energy efficiency of available state of the art solutions listed in Table 5.18, the power consumption is scaled to the 180 nm, 1.5 V technology node in Table 5.19, using Equation (5.3), where P is the power, V_{dd} is the supply voltage and λ is the feature size. The ITRS roadmap [6] and [58] indicate that beyond 130 nm, the quadratic power scaling ($\alpha = \beta = 2$ in Equation (5.3)) breaks down and follows a slower trend.

$$P_2 = P_1 \left(\frac{V_{dd,2}}{V_{dd,1}}\right)^{\alpha} \left(\frac{\lambda_2}{\lambda_1}\right)^{\beta} \tag{5.3}$$

Similarly, the throughput T needs to be scaled, as it directly relates to the achievable operation frequency, that depends on the technology node. A linear impact by both the feature size and the supply voltage is assumed in Equation (5.4).

$$T_2 = T_1 \left(\frac{V_{dd,2}}{V_{dd,1}}\right) \left(\frac{\lambda_1}{\lambda_2}\right) \tag{5.4}$$

The scaled throughput versus scaled power plot in Figure 5.16(b) and the scaled energy per pixel in the one but last column of Table 5.19 compare the available state of the art MPEG-4 video encoders in a 180 nm, 1.5 V

[5] An extensive prior art of 1D (I)DCT cores exists. However, due to the lack of available IP, a too straightforward and power hungry implementation (see Figure 5.17) is currently used. Hence the suboptimal realization of this block offers room for further improvement of the power efficiency.

[6] Moved on-chip using embedded DRAM

[7] Assuming 1 gate = 4 transistors

Table 5.18: Characteristics of state of the art MPEG-4 part 2 video implementations

Design	Throughput (Mpixels/s)	Process (nm, V)	Freq. (MHz)	Power (mW)	Area (kGates)	on-chip SRAM (kbit)	off-chip SDRAM (Byte)	external accesses per pixel
[14]	3.0 (L3)	180, -	36	116	170	43	161k	-
[81]	1.5 (L2)	180, 1.5	13.5	29	400	-	2M	-
[113]	0.4 (L1)	180, 1.5	70	170	-	128	128k[6]	0
[15]	1.5 (L2)	130, 1.5	125	160	3000	-	2M[6]	0
[24]	3.0 (L3)	350, 3.3	40	257	207[7]	39	2M	>5
[114]	27.6 (>L5)	130, 1.3	81	120	390	80	>2.6M	17
[109]	9.2 (L4)	130, 1.2	40.5	50	800	-	-	-
[95]	9.2 (L4)	130, 1	-	9.6	170	-	-	-
[73]	9.2 (L4)	180, 1.4	28.5	18	201	36	-	-
This work	12.2 (>L5)	180, 1.6	101	71	87	400	594k	2

technology node. The proposed core is clearly more low power than [114, 14, 81, 109]. The power consumption of [24], including a SW controller, is slightly better (note that this work does not include the SW orchestrator). However, taking the off-chip memory accesses into account to evaluate the total system power consumption, the proposed solution has a (at least) 2.5 times reduced off chip transfer rate, leading to the lowest total power consumption. Even when compared to the ASICs containing embedded DRAM [15, 113], our implementation is more power-efficient, as only 1.5 mW is required at L2 (15 CIF fps) to read and write every pixel from the external memory (assuming 1.3 nJ per 32 bit transfer). For L1 and L2 throughput respectively, [113] and [15] consume more than 150 mW (Table 5.18). Our implementation delivers 15 CIF fps (L2) consuming only $9.7 + 1.5 = 11.4$ mW. Using complementary techniques, like a low power CMOS technology, [95, 73] achieve an even better core energy efficiency. Insufficient details prevent a comparison including the external memory transfer cost.

The last column of Table 5.19 presents the scaled energy delay product. This measure includes the scaled throughput as delay per pixel. The previous observations also hold using this energy delay product, except for the 30 CIF fps of [24], having now a worse result than the proposed encoder. Note that a complete comparison should also include the coding efficiency of the different solutions, as algorithmic optimizations (like different ME algorithms) sacrifice compression performance to reduce complexity. Unfortunately, not all referred papers contain the required rate-distortion information to make such evaluation. The state-of-the art energy efficiency of the proposed encoder core, resulting from applying the design flow of Chapter 2, is the proof-of-concept of contribution 6.

Table 5.19: State of the art MPEG-4 part 2 video implementations scaled to a 180 nm, 1.5 V technology.

Design	Scaled Power (mW)	Scaled Throughput (Mpixels/s)	Scaled energy per pixel (nJ/pixel)	Scaled energy delay product ($\frac{\text{nJ}}{\text{pixel}} \cdot \frac{\mu\text{s}}{\text{pixel}}$)
[14]	116	3.0	38.1	12.5
[81]	29	1.5	19.1	12.5
[113]	170	0.4	447.2	1176.3
[15]	261	1.1	279.3	254.3
[24]	14	2.7	5.2	1.9
[114]	249	23.0	13.3	0.6
[109]	119	8.3	18.0	2.2
[95]	31	10.0	4.1	0.4
[73]	21	9.9	2.1	0.2
This work	61	11.3	5.4	0.5

5.5.4 Effect of the high-level optimizations

The algorithmic optimizations of Section III.A reduce the average number of cycles to process a macroblock compared to the worst case (typically around 25%, see Table 5.15) and hence the power consumption.

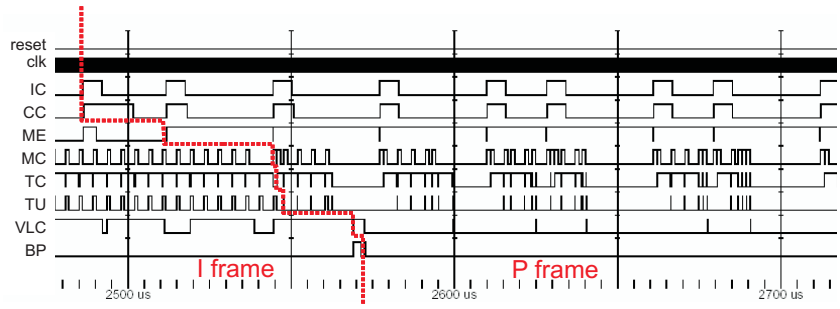


Figure 5.18: Activity diagram of the video pipeline in regime mode. The texture coding is the critical path in the I frame, motion estimation in the P frame.

Figure 5.18 shows the activity diagram of the pipelined encoder in regime mode. The signal names are the acronyms of the different functional modules in Figure 5.15. During the I frame (left side of the dotted line in Figure 5.18), the error block contains a lot of energy, forcing the texture coding to always fully process the block and making it the critical path (TC).

During a P frame (right side of the dotted line in Figure 5.18), the search for a good match in the previous frame is the critical path, making ME almost 100% active. The early stop criteria sometimes shorten the amount of cycles required during the motion estimation. When this occurs, often a good match is found, allowing the texture coding to apply its intelligent block pro-

cessing and also reducing the amount of cycles to process the blocks of the macroblock. In this way, both ME and TC critical paths are balanced (i.e. without algorithmic tuning of the texture coding, this process would become the new critical path in a P frame). Additionally, a good match leads to a higher amount of skipped blocks with possible zero motion vectors. Consequently, the algorithmic tuning reduces the amount of processing and data communication, leading to improved power efficiency.

5.5.5 Design Time

The required effort to transform the optimized specification into a custom encoder or decoder integrated on the Wildcard-II as demo platform is measured in Figure 5.19. At the decoder side a traditional approach was taken for the HDL design and debug step, while at the encoder side, the proposed methodology was strictly applied. The encoder consists of 8 functional components and the decoder only of 5, explaining the higher effort to create the HDL for them. Profound verification of each functional component separately, before moving to the integration phase, clearly results in shortened debug cycle when composing the system. Overall, the proposed methodology reduces the total design time, as the more complex encoder requires less man effort than the development of the decoder.

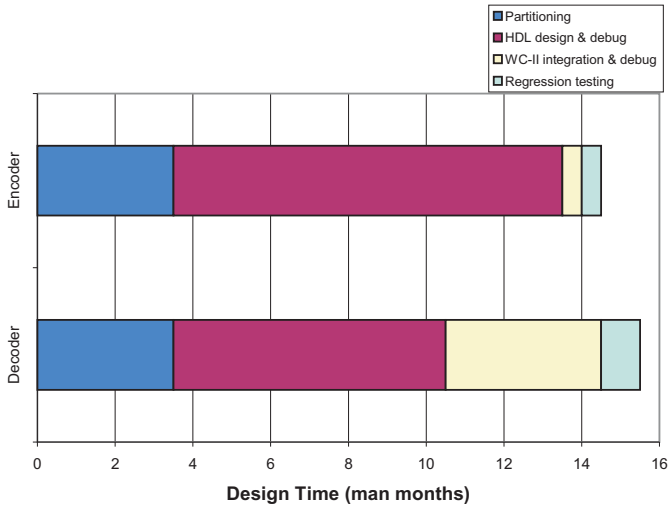


Figure 5.19: Strictly applying the RTL development and verification strategy reduces the effort during the integration phase.

5.5.6 Design Experiences

The presented design of a dedicated MPEG-4 part 2 Simple Profile encoder does not only confirm the applicability of the proposed design flow to block-based video processing, it also contributed to consolidate the design approach. The following list contains some design experiences gained from the encoder implementation.

- 'The flow needs to flow' (quote Bob Turney). All steps of the design flow should connect as seamlessly as possible to prevent, possibly error prone, code adaptation to enter the next step. This is also true for the tool flow supporting the design flow.
- The library of communication primitives should strictly defined and correctly implemented before the designer needs them during the development of his functional block.
- A board acceptance test is required to bring up the (prototyping) platform properly and to avoid board related issues during the mapping of the application. This also relates to the library of communication primitives that should be tested on the platform to assure their proper behavior.
- The RTL development and test environment is important to ease the hand-writing of VHDL and shorten its debug cycle.
- The current design demonstrator only uses a subset of the proposed special channels. Making another design experiment to implement a more recent video standard or another type of block-based video processing could also require the other ones and would further mature the design flow (see Section 7.2).
- 'Never think you have found the last bug' (quote Kees Vissers).

5.6 Related Work

Many implementations of MPEG-4 part 2 video encoders and decoders are proposed in literature, mainly focusing on the (Advanced) Simple Profile. Reference [91] gives a good overview, grouping the solutions in 3 categories: programmable approaches, hybrid architectures and dedicated architectures. Only the last category, to which this work belongs, offers the required throughput while meeting the low-power requirements and heat dissipation constraints.

Section 5.5 shows our design [40] is at that time more energy efficient than other MPEG-4 part 2 Simple Profile video encoders [14, 15, 24, 81, 109, 113, 114], when also taking the external memory access cost into account. Recently, Lin [73] presented at ISSCC2006 a VGA encoder, doubling the energy efficiency by focusing on the bandwidth of the ME and on the (I)DCT. Rintaluoma [95] includes power numbers of a Hantro encoder core with a 32% better energy efficiency. The latter one uses a low power CMOS library, a technique not yet applied in the proposed design. Further optimizations of this work can be achieved by also using low-power memory libraries, by exploiting voltage scaling techniques [106] and by improving the (I)DCT (as currently only a simple implementation is included due to the extensive prior art in this field but a lack of IP availability).

The resource scaling and multi-stream possibilities of the proposed video codec are features not clearly supported by the encoders and decoders [44, 55, 82] in literature.

5.7 Conclusion

The design methodology is demonstrated on the development of an MPEG-4 Simple Profile video encoder and decoder, capable of processing respectively 30 4CIF (704×576) and 30 XSGA (1280×1024) frames per second or multiple lower resolution sequences (contribution 6). Starting from the reference code, all design steps of Chapter 2 are applied. Memory optimizations, transforming the processing of the codec to a (macro)block-based operation, are combined with algorithmic tuning of the rate control, the motion estimation and the texture coding.

This high-level optimized codec with localized data processing is partitioned in a fully dedicated video pipeline. Using the special channels of Chapter 3, a CSDF model of the codec is constructed. To obtain the desired self-timed execution of the system, its synchronizing queues are dimensioned using the buffer capacity calculation on this CSDF model.

The separate HDL translation of each functional component of the video codec, using the RTL development and verification approach of Chapter 4, enables a short design time to integrate a functionally correct system on an FPGA and ASIC.

The resulting design has a high degree of concurrency, uses a dedicated memory hierarchy and exploits burst oriented external memory I/O leading to the theoretical minimum of off-chip accesses. The video codec implementation is scalable with a number of added compile-time parameters (e.g. maximum frame size and number of bitstreams), which can be set by the user to best suit

the application. The presented encoder core uses less than 71 mW (180 nm, 1.62 V UMC technology) when processing 4CIF at 30 fps. Further improvements of the energy efficiency are expected from using a low power CMOS technology (including low power memory libraries), from the introduction of voltage islands and from redesigning the currently straightforward implementation of the(I)DCT core.

Chapter 6

Intelligent Block Processing

We'll make mistakes and then
Life is the art of learning to live with it
Through time

'Through Time' – Roisin Murphy

The spatial model of a video codec uses transform and quantization operations to first decorrelate the spatial samples and then reduce the frequency domain result to the perceptually important values only. The inverse operation allows the reconstruction of the original spatial samples. Since all of these processing steps are compute intensive, this chapter details the development of a heuristic used in the algorithm optimization of MPEG-4 part 2 Simple Profile encoder (see Chapter 5). This algorithmic tuning case belongs to the high-level optimization step of the design flow. It trades a minimal amount of compression performance for a significant reduction of the computational complexity.

The heuristic predicts the coding status of a block after quantization: only zero values, only a first row or column of non-zero values or also other non-zero values. Combining this coding status, either from the bitstream at the decoder side or from prediction at the encoder side, with the value of its motion vectors, avoids motion compensation and storing of unaltered data in the reconstructed frame buffer.

This chapter first discusses the simplification of the texture coding at the encoder by predicting its coded block status. Section 6.2 makes the combination with the motion vector values.

6.1 Texture Coding Simplification

The texture coding task is the second largest access and cycle consumer of the MPEG-4 part 2 SP video encoder (5.8 % of the complexity of the initial video encoder, see Table 5.3). Its input is the prediction error $f(h, v)$, after motion compensating a new block $n(h, v)$ in a macroblock mb with the data in the reconstructed frame r, pointed to by the motion vectors mv_x and mv_y. The texture coding processes this residual signal on a block basis and contains DCT, quantization, inverse quantization and IDCT (Figure 6.1).

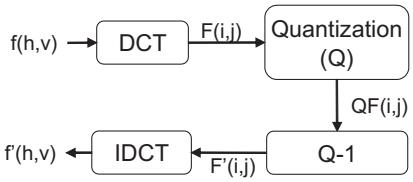


Figure 6.1: All stages in the coding chain are computation intensive.

Equations (6.1) and (6.2) respectively present the forward en inverse discrete cosine transform. Even with row-column decomposition and butterfly implementation, this transform requires many multiplications and additions.

$$F(i,j) = \frac{2}{N} C(i)C(j) \sum_{h=0}^{N-1} \sum_{v=0}^{N-1} f(h,v) \cos\left(\frac{\pi}{2N} i(2h+1)\right) \cos(\frac{\pi}{2N} j(2v+1)) \quad (6.1)$$

$$f(h,v) = \frac{2}{N} C(i)C(j) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} F(i,j) \cos\left(\frac{\pi}{2N} i(2h+1)\right) \cos(\frac{\pi}{2N} j(2v+1)) \quad (6.2)$$

$$C(i), C(j) = \begin{cases} \frac{1}{\sqrt{2}} & i,j = 0 \\ 1 & \text{otherwise} \end{cases} \quad (6.3)$$

The quantization is basically an integer division (Equation (6.4)), while the inverse quantization for an intra DC coefficient (Equation (6.5)) or other coefficients (Equation (6.6)) is mainly a complicated multiplication.

$$QF(i,j) = \begin{cases} \frac{F(0,0) + \frac{dc\_scaler}{2}}{dc\_scaler} & \text{intra DC} \\ \frac{F(i,j)}{2QP} & \text{otherwise} \end{cases} \quad (6.4)$$

$$F'(i,j) = dc\_scaler \cdot QF(0,0) \tag{6.5}$$

$$|F'(i,j)| = \begin{cases} (2|QF(i,j)|+1)\cdot QP & \text{odd QP and } QF(i,j) \neq 0 \\ (2|QF(i,j)|+1)\cdot QP - 1 & \text{even QP and } QF(i,j) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

Overall, texture coding is a computation intensive task. Consequently, this section omits all or part of the computations when the quantized coefficients can reasonably be estimated to be equal to zero.

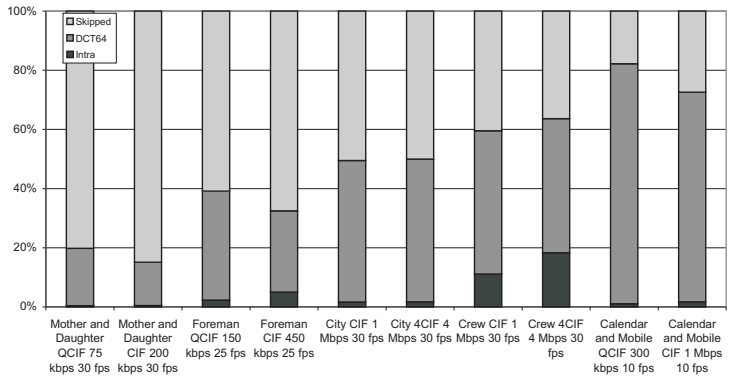


Figure 6.2: Relative block type proportions over different video sequences. The amount of skipped blocks is inversely proportional to available bitrate and the complexity of the sequence. The prediction of these blocks eliminates their texture coding.

In motion compensated blocks, the prediction error is likely to be reduced to zero due to the quantization of non-relevant coefficients. Such all-zeros blocks are referred to as *skipped* in the following, as their texture decoding stage can be omitted on the decoder side. Figure 6.2 shows the ratio between intra (no motion compensation), DCT64 (non-zero inter block after DCT and quantization) and skipped blocks for a selection of the test cases, of which the video complexity increases from left to right. This figure indicates that the amount of skipped blocks is inversely proportional to the available bitrate and to the complexity of the video sequence. The coding status of a prediction error block is only known after the DCT and quantization process. For a skipped block, the computations in these steps become overhead, as the resulting reconstructed texture block contains only zeros. Algorithmic tuning tries to predict this coding status avoiding useless processing. This coding status is determined based on Equation (6.9), by comparing the Sum of Absolute Differences of a block to a combination of the Quantization Parameter (QP) and

the Absolute Deviation ($AD_{macroblock}$) of the macroblock holding this block. The Sum of Absolute Differences of a block SAD_{block} is the prediction error between a new block n and the data in the reconstructed frame, pointed to by the motion vectors mv_x and mv_y of the block (Equation (6.7)). The Absolute Deviation $AD_{macroblock}$ of a macroblock mb quantifies its activity (Equation (6.8)). The threshold T is set according to the coding status under investigation.

$$SAD_{block} = \sum_{i=0}^{7}\sum_{j=0}^{7} |n(i,j) - r(i+mv_x, j+mv_y)| \tag{6.7}$$

$$AD_{macroblock} = \sum_{i=0}^{15}\sum_{j=0}^{15} (mb(i,j) - \overline{mb}) \tag{6.8}$$

$$codingstatus = SAD_{block} < T \cdot QP + \frac{AD_{macroblock}}{200} \tag{6.9}$$

The first term in Equation (6.9) is proportional to QP as a higher quantization level sets larger error coefficients to zero. References [86] and [71] show all quantized coefficients are zero when $SAD_{block} < 20 \cdot QP$. The second term in (6.9) is deducted empirically (more specifically, the factor 200 is derived using extensive testbenches) and reflects the dependency on the absolute deviation: a larger SAD_{block} is acceptable for macroblocks with high activity. The intuition behind it is that a large activity corresponds to a spreading of the energy among the DCT coefficients. If the energy has been spread among several coefficients, the magnitude of the largest ones is smaller than if the energy was concentrated on a few coefficients. Moreover, high activity also means high frequency coefficients, and these ones are sometimes (depending on the quantization matrix) quantized more coarsely. Hence, if the block prediction error is small enough (evaluation of (6.9) with $T = T_{skip} = 20$), the block is expected to result in all-zeros quantized coefficients and consequently, the algorithm decides to encode it as a skipped block.

Next to skipped blocks, observations show that most DCT blocks with small SAD values only contain non-zero quantized coefficients on either their first row or their first column (DCT8). Specifically, experiments show that using (6.9) with $T = T_{DCT8} = 30$ identifies the blocks only containing relevant coefficients in one direction. The prediction of these blocks allows for complexity reduction, because only one row or one column DCT needs to be computed. In practice, the 1-D DCT is calculated after initial summations along the rows or columns of the block.

Equation (6.9) and its values of T, i.e., $T_{skip} = 20$ and $T_{DCT8} = 30$ appear to

Table 6.1: Accuracy of the prediction formula for different threshold settings ($T_{skip} = \{15, 20, 25\}$ and $T_{DCT8} = \{20, 25, 30\}$) on a selection of the test cases in the testbench, compared to the result without using the intelligent block processing. Making the thresholds too flexible results in too many wrong predictions and hence in a loss of detail information.

Test case	T_{skip}	T_{DCT8}	skipped				DCT8		
			correct	asDCT8	asDCT64	wrong	correct	asDCT64	wrong
M&D QCIF 75 kbps 30 fps	15	25	46.1	29.1	24.8	0.0	10.1	89.9	1.4
	20	30	63.5	20.6	15.9	0.5	22.7	75.3	4.7
	25	35	76.0	15.3	8.7	3.5	30.9	57.5	11.3
M&D CIF 200 kbps 30 fps	15	25	54.3	29.6	16.1	0.0	18.7	81.3	2.9
	20	30	72.3	20.3	7.4	1.2	39.5	57.5	11.4
	25	35	84.1	13.5	2.4	9.8	47.2	32.4	26.7
Foreman QCIF 150 kbps 25 fps	15	25	36.4	38.0	25.6	0.1	18.3	81.3	1.5
	20	30	58.2	28.5	13.3	1.0	34.3	61.3	5.8
	25	35	75.4	18.4	6.2	5.5	38.6	41.2	13.2
Foreman CIF 450 kbps 25 fps	15	25	40.0	39.6	20.4	0.1	20.6	79.2	1.9
	20	30	63.4	26.9	9.8	1.3	38.6	57.6	7.2
	25	35	80.2	15.5	4.3	7.8	39.9	38.1	15.5
City CIF 1 Mbps 30 fps	15	25	35.0	36.9	28.1	0.0	16.8	83.2	0.8
	20	30	58.8	23.1	18.1	0.7	25.7	70.5	2.9
	25	35	72.5	17.8	9.7	3.6	23.4	57.7	7.2
City 4CIF 4 Mbps 30 fps	15	25	41.1	35.0	23.8	0.0	15.3	84.6	0.9
	20	30	62.8	24.6	12.6	0.8	25.9	70.3	3.8
	25	35	76.7	18.3	5.0	4.2	29.5	53.3	9.7
Crew CIF 1 Mbps 30 fps	15	25	33.6	42.8	23.6	0.0	12.5	87.5	1.7
	20	30	57.4	32.9	9.7	0.6	30.9	67.5	6.4
	25	35	77.9	19.1	3.0	5.6	40.8	45.7	13.6
Crew 4CIF 4 Mbps 30 fps	15	25	31.1	45.5	23.4	0.0	9.8	90.2	1.2
	20	30	56.0	35.3	8.7	0.4	28.0	71.0	5.4
	25	35	78.1	19.7	2.3	5.0	40.2	49.0	12.3
C&M QCIF 300 kbps 10 fps	15	25	16.4	28.2	55.4	0.0	6.0	93.7	0.2
	20	30	31.0	28.4	40.6	0.1	15.3	83.2	0.7
	25	35	46.9	28.5	24.7	0.6	26.3	65.6	2.3
C&M CIF 1 Mbps 10 fps	15	25	22.5	36.3	41.2	0.0	8.5	91.3	0.4
	20	30	41.3	34.9	23.8	0.2	22.1	75.7	1.7
	25	35	61.2	27.7	11.0	1.2	35.2	54.4	4.6
average	15	25	35.6	36.1	28.2	0.0	13.7	86.2	1.3
	20	30	56.6	27.6	16.0	0.7	28.3	69.0	5.0
	25	35	72.9	19.4	7.7	4.7	35.2	49.5	11.6

provide the best trade off between computational complexity reduction and preserving a homogeneous quality on the video frames. Table 6.1 measures the relative amount and the correctness of the predictions, compared to a version without block type prediction, when using 6.9 with different threshold settings. For skipped blocks, 4 categories are discriminated:

1. correct: accurate prediction. Expressed as percentage of the total number of skipped blocks without prediction.
2. asDCT8: incorrect prediction (predicted as DCT8, should have been predicted as skipped). This leads to too many computations, but does not affect the quality. Expressed as a percentage of the total number of skipped blocks without prediction.
3. asDCT64: incorrect prediction (predicted as DCT64, should have been predicted as skipped). This leads to even more computations, but does not affect the quality. Expressed as a percentage of the total number of skipped blocks without prediction.
4. wrong: incorrect prediction (predicted as skipped, should have been DCT8 or DCT64). This avoids computations, but deteriorates the quality. Expressed as a percentage of the total number of DCT8 and DCT64 blocks without prediction.

For DCT8 blocks, 3 categories are discriminated:

1. correct: accurate prediction. Expressed as percentage of the total number of DCT8 blocks without prediction.
2. asDCT64: incorrect prediction (predicted as DCT64, should have been DCT8). This leads to too many computations, but does not affect the quality. Expressed as a percentage of the total number of DCT8 blocks without prediction.
3. wrong: incorrect prediction (predicted as DCT8, should have been predicted as skipped). This avoids computations, but deteriorates the quality. Expressed as a percentage of the total number of DCT64 blocks without prediction.

Using a T_{skip} of 15 and a T_{DCT8} of 25 is a safe but compute intensive option: only a small amount of blocks ($< 3\%$) is wrongly predicted, while the number of correctly predicted skipped blocks remains below 50 %. The improved amount of correct predictions at T_{skip} of 25 and a T_{DCT8} of 35 comes at the cost of a large prediction error up to 25%. The proposed thresholds ($T_{skip} = 20$ and a $T_{DCT8} = 30$) offer the best compromise. Note that even if a skipped

block is inaccurately predicted as DCT8, still the amount of computations is significantly reduced, compared to full DCT64 processing when no intelligent block prediction is present.

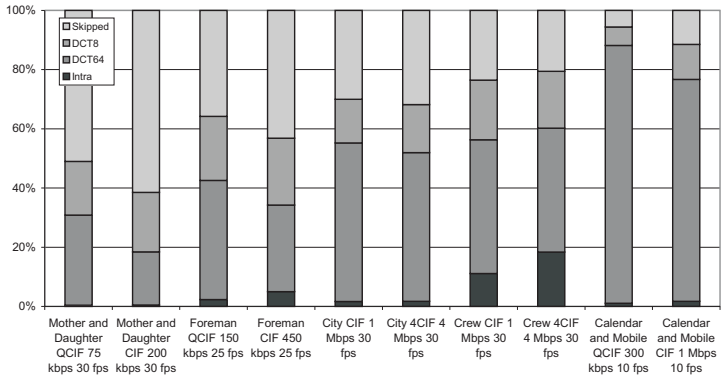


Figure 6.3: The block type after the texture coding is accurately predicted to reduce the texture coding complexity. This results in 40 % skipped blocks and 20 % with simplified processing (row or column only) for the typical Foreman CIF 450 kbps test case.

Figure 6.3 shows the proportion of each kind of texture blocks before processing (DCT/Q/Q^{-1}/IDCT). The amount of blocks, predicted for simplified processing, corresponds closely to the number of finally not coded blocks when no intelligent processing is used (Figure 6.2). The prediction avoids the computation overhead, by identifying in advance blocks for which the DCT is expected to result in zero quantized coefficients (contribution 7).

Table B.5 and Table B.6 respectively list the threshold setting measurements and the block type proportions for the complete testbench.

6.2 Conditional Motion Compensation

Additionally evaluating the motion vectors of a block with skipped coding status enables further simplification of the block processing steps. When the memory organization of the video codec is optimized to a hierarchy, containing only a single reconstructed frame memory (as described in Section 5.2.2), the motion compensation operations and the storing of the reconstructed result (texture update) can be avoided, if a skipped block also has zero motion vectors. Figure 6.4 extends the coding status with this new type (Skipped MV0). Depending on the complexity of the sequence, from a negligible amount up to 75 % of the blocks are skipped with zero motion vectors. Table B.7 lists the coding status proportions for the complete testbench.

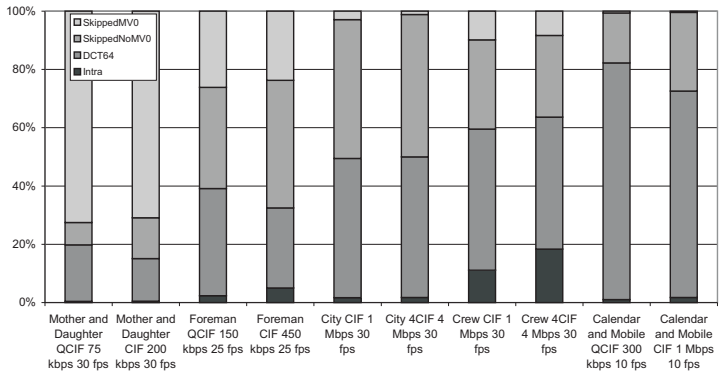


Figure 6.4: Relative block type proportions, including skipped blocks with zero motion vectors, over different video sequences.

Identifying skipped blocks with zero motion vectors leads to a conditional execution of the motion compensation in the encoder or decoder, as the motion compensation is skipped for these blocks. When the video codec has a single reconstructed frame, the data in this frame memory remains unchanged between two frames for a skipped MV0 block. Consequently, no storing is needed.

Combined with the optimizations of the previous section, a skipped block with zero motion vectors only requires the determination of its coding status: performing a motion estimation and using the heuristics for an encoder or extract it from the bitstream for a decoder. No further operations are required. This conditional motion compensation [36] is part of contribution 7.

6.3 Related Work

The computational load of texture coding operations, transform and quantization, is well known. One of the optimization techniques to reduce the DCT complexity [16] is exploiting the presence of many zero-valued coefficients and of complete zero-valued blocks (skipped).

Other works [56, 86, 115] mention the prediction at the encoder of these blocks full of zeros to control the complexity. Zhao [115] only predicts skipped blocks with zero motion vectors. In this way, also the motion estimation can be avoided. We use an algorithmically optimized ME with early stop criteria, representing a similar behavior. In contrast, we predict skipped blocks with non zero motion vectors and blocks with only one row or one column to reduce the complexity of the transform and quantization. Similarly, [86] predicts

skipped blocks, blocks with only a DC value and low frequency blocks. Their prediction is only based on the SAD, while we also include the mean absolute deviation to have a better heuristic (leading to further complexity reduction).

Additionally, we avoid the motion compensation and texture update for skipped blocks with zero motion vectors, as a result of the introduced memory hierarchy during the DTSE optimizations [36]. Recently, this technique has bee described (at the decoder side only) in [23].

6.4 Conclusion

To reduce the computational complexity of the transform and the quantization steps in the texture processing, this chapter proposes a heuristic to predict blocks with only zeros or blocks with only the first row or column with non-zero values (contribution 7). In this way, quantization and transform is avoided or simplified. This algorithmic tuning case belongs to the high-level optimization phase of the design flow and achieves an important reduction of the computational complexity.

By combining this coding status with the value of the motion vectors, also motion compensation and texture update in a video codec can be omitted. To allow this, a memory hierarchy with a single reconstructed frame is required. Skipped blocks with zero motion vectors then remain unaltered between two frames, hence no processing is required for them. Consequently, also the memory complexity is reduced.

Chapter 7

Conclusions

With one wish we wake the will
Within wisdom
With one will we wish the wisdom
Within waking
Woken, wishing, willing

'Song of Sophia' – Dead Can Dance

The low power requirements, resulting from the limited heat dissipation and battery lifetime of mobile devices, demand energy efficient implementations. This final chapter first summarizes the contributions of the design flow proposed to reach these goals for block-based video processing applications and of its use on the development of an MPEG-4 part 2 Simple Profile video encoder and decoder. Section 7.2 then proposes several topics for future research.

7.1 Summary

Modern multimedia devices support rich media exchange: voice, text, audio, video, graphics etc. These appliances seek to improve the user experience by invoking advanced audio visual algorithms and increasing resolutions. On the other hand, the power and heat dissipation limitations of portable devices require low-power implementations. These conflicting demands increase the challenge to reach a cost-efficient design.

This PhD thesis proposes a design flow oriented towards low power block-based video processing, bridging the gap between a high-level specification (typically

C) and the final realization. Its main cost factor, the energy efficiency, is expressed by the energy delay product that covers both the energy consumption and the sustained throughput. The applied design philosophy orders a set of various power optimization techniques over different design levels, according to the trade-off between their impact on the energy delay product and their development cost: (i) redefine the problem to reduce complexity, (ii) introduce parallelism to reduce delay per task (while keeping the energy per task constant) and (iii) add specialized hardware as it achieves the best energy efficiency. The typical data dominance of multimedia systems motivates the memory and communication focus of the design flow.

To cope with this memory and communication bottleneck, the design flow mainly extends the established Data Transfer and Storage Exploration (DTSE) methodology that focuses on memory optimizations. At the high-level, algorithmic tuning is added, while the lower levels are completed with a partitioning exploration, matched towards RTL development (contribution 1, Chapter 2). This results in a design flow with two phases.

The first phase of the design flow reformulates the problem to reduce the complexity. It combines memory optimizations with algorithmic tuning. As a result, the reference code is transformed into a block-based video application with localized processing and dataflow. Memory footprint and frame memory accesses are minimized. These optimizations prepare the system for introducing concurrency and provide a reference for the RTL development.

To provide a good abstraction means to reason about the parallelism during the second phase, a Cyclo-Static DataFlow (CSDF) Model of Computation (MoC) is used, as it matches well with the dataflow dominated behavior of multimedia processing. Among different dataflow models of computation, CSDF is one of the most expressive MoCs while keeping the full analysis potential (e.g. consistency checks, dead-lock analysis, etc). Also implementation specific aspects of communication channels, like data reuse, shared buffers or restricted buffer sizes, can be expressed in a CSDF graph. Such special behavior is often related to the use of a kind of shared circular buffers. This thesis proposes to represent a (special) communication channel by two dataflow edges to correctly model the synchronization and the free buffer space between the communicating tasks. Consequently, the graph remains completely analyzable and allows reasoning about the temporal behavior of the application, modeled by the CSDF graph (contribution 2, Section 3.2).

Based on such a CSDF model of computation, used in a specific way to also express implementation specific aspects, the required buffer sizes of the communication channels are calculated to obtain a fully pipelined and parallel self-timed execution (contribution 2, Section 3.4). With a high-level concurrent SystemC model, automatically generated by the SPRINT tool from the

(single threaded) optimized C specification, the correctness of the buffer capacities and degree of parallelism is cross-checked. After this upfront verification, the RTL link of SPRINT (contribution 3) is used to start the hardware development.

Towards dedicated hardware, the communication channels are implemented as a restricted but sufficient set of communication primitives (contribution 4, Section 3.5). By exploiting the principle of separation of communication and computation, each tasks is translated individually to HDL, while correctly modeling its communication. An automated environment supports the functional verification of such a component by combining simulation of the RTL model and testing the RTL implementation on a fast prototyping platform. This elaborate verification of each single component reduces the debug cycle for integration (contribution 5, Chapter 4). A significant design time reduction (over a factor 2) is measured on the MPEG-4 design, where the RTL development and verification approach (described above) was strictly applied at the encoder but not at the decoder side.

The design methodology is demonstrated on the development of an energy efficient MPEG-4 Simple Profile video encoder and decoder, capable of processing respectively 30 4CIF (704×576) and 30 XSGA (1280×1024) frames per second, or multiple lower resolution sequences (contribution 6, Chapter 5). Starting from the reference code, algorithmic simplifications (at the expense of a slight degradation in visual quality) are made to the texture coding, the rate control and the motion estimation. More specifically, this algorithmic tuning introduced an intelligent block processing in the texture coding, predicting the coding status of a block after quantization: full of zeros, one row or column of non-zero values only, or also other non-zero values (contribution 7, Chapter 6). Memory optimizations transform the video codec to make the processing (macro)block-based. Towards hardware, this processing chain is realized as a dedicated video pipeline mapped on an FPGA and ASIC, with a high degree of concurrency, using a dedicated memory hierarchy and exploiting burst oriented external memory I/O. This leads to the theoretical minimum of off-chip accesses. The video codec design is scalable with a number of added compile-time parameters (e.g. maximum frame size and number of bitstreams), which can be set by the user to best suit the application. The presented encoder core consumes 71 mW (180 nm, 1.62 V UMC technology) when processing 4CIF at 30 fps. This energy efficiency, achieved without using a low-power CMOS library, belongs to the state of art for high resolution video encoders (contribution 6, Section 5.5).

7.2 Future Research

Only the decoder and encoder of a single video standard (MPEG-4 part 2 Simple Profile) have currently been designed using the proposed design flow. To prove its stability, another, preferably more recent, video standard should be implemented as well. This would further mature the design flow. Additionally, the flow has the potential of becoming more general. The design approach with its sequential and parallel phase could be equally applied to the design of any other digital signal processing application core. Extra design experiments are required to further denote the application domain restrictions of the flow.

Rich media content, and especially visual information, is not only characterized by its high processing requirements, but also by its diversity of media formats and the continuing development of compression and rendering algorithms. The current hardware orientation of the last part of the design flow does not provide flexibility to deal with these changes. Consequently, future multimedia platforms will contain more and more programmable cores. The proposed design flow also has a SW tuning part in the mapping step. It should be investigated how well the design flow works towards a heterogenous architecture. Is the CSDF model with its special channels also able to represent the implementation aspects of such a platform? Is CSDF still an appropriate MoC when dynamism makes worst case design less relevant? Can CSDF be used for SW development? Is a software tuning step sufficient to map a task efficiently on a chosen processor, or are there extra steps required?

The high-level verification using a timed concurrent SystemC model currently only proposes the basic idea of playing with the response time of tasks, to test if the blocking behavior of the queues yields the proper synchronization and if the shared memories are sized correspondingly. Further research should identify the fault coverage of this approach and how it can be extended to improve this coverage.

Currently, the SPRINT tool requires user pragmas to define the partitioning of the system and the sizes of the queues. Using the proposed CSDF modeling of regular and special channels, an extension could be made to the SPRINT tool to automatically perform the buffer capacity calculation for a desired schedule, as provided by the user. Another important improvement to SPRINT is the automated generation of the CSDF model, including the production and consumption sequences as currently this manual step prevents its correct by construction property. Adding this feature provides a solid solution, making the high-level verification and its extension of the previous paragraph superfluous. This could finally lead to auto-parallelization support being added to this tool, supporting or even relieving the designer from the task to come up with a good partitioning. Having such automated approach would allow

an analysis of not the desired schedule, but also other schedules that better explore the trade-off between throughput, buffer capacity and response times.

To implement the special channels introduced in the CSDF modeling, a regular channel (or path of channels), having redundant synchronization, is assumed. This principle of redundant synchronization, explained in literature for HSDF, first needs to be further investigated in the context of CSDF. Secondly, adding auto-detection of redundant synchronization to the SPRINT tool would avoid unnecessary synchronization overhead.

An efficient use of the processing resources on a heterogeneous multi-processor requires a balanced split of the application over these resources. Currently only a subset of parallelization possibilities is studied, as the focus is on functional parallelism. Coarse data parallelism (at the task-level) should be included as well. This removes the assumption that a task only appears once in a graph. Consequently, it should be investigated how well data parallelism can be expressed in a dataflow graph. Additionally, the usefulness of the special channels (multiple consumer and multiple producer) needs to be evaluated.

Having an energy efficient design only is not sufficient, also an energy aware usage of the core is important. This relates to developing a quality manager, that does not only makes a rate distortion optimization, but also takes the energy into account. Recent standards like AVC have many encoding parameters that can be set to steer this rate-distortion-energy trade-off. Consequently, for each aspect of the trade-off a model is required. Good video codec energy models, working accurately for video sequences of different nature, are not yet available in literature.

To complete the MPEG-4 part 2 video codec design, other low-power techniques can still be applied. It was already noted that the current (I)DCT kernel is a straightforward implementation, while many more efficient versions are described in literature. Additionally, unexploited techniques, like voltage islands, low-power memory libraries, etc. could be added to further improve the energy delay product of the design. Finally, moving the design to a more recent technology node would provide a more relevant power figure and would also allow validating the used scaling approach.

Appendix A

CSDF Graph Details

All details of the MPEG-4 encoder CSDF graph (Figure 5.11) left out of Chapter 5 are included in this appendix: firing sequences (Section A.1) and buffer capacity calculations (Section A.2).

A.1 Encoder Firing Sequences

The firing sequences at producer and consumer side of edges e_4 and e_{12} are given in Equations (A.1) to (A.6).

$$p^4_{IC\&CC}(v \cdot w_{MB} + h) = \begin{cases} w_{MB} + 2 & \text{if } h = 0 \text{ and } v = 0 \\ 2 & \text{if } h = 0 \text{ and } 0 < v < h_{MB} - 1 \\ 1 & \text{if } h < w_{MB} - 1 \text{ and } v < h_{MB} - 1 \\ 0 & \text{otherwise.} \end{cases} \tag{A.1}$$

$$r^4_{IC\&CC}(v \cdot w_{MB} + h) = \begin{cases} w_{MB} + 1 & \text{if } h = 0 \text{ and } v = 0 \\ h + 2 & \text{if } 0 < h < w_{MB} - 1 \text{ and } v = 0 \\ 2w_{MB} & \text{if } h = w_{MB} - 1 \text{ and } v = 0 \\ 2w_{MB} + 1 & \text{if } h = (0, w_{MB} - 1) \text{ and } 0 < v < h_{MB} - 1 \\ w_{MB} + 1 & \text{if } 0 < h < w_{MB} - 1 \text{ and } 0 < v < h_{MB} - 1 \\ w_{MB} + 2 & \text{if } h = 0 \text{ and } v = h_{MB} - 1 \\ w_{MB} + 1 & \text{otherwise.} \end{cases} \tag{A.2}$$

$$a^4_{IC\&CC}(v \cdot w_{MB} + h) = \begin{cases} 0 & v = 0 \\ 2 & \text{if } h = 0 \text{ and } v > 0 \\ 1 & \text{if } h < w_{MB} - 2 \text{ and } v > 0 \\ 0 & \text{if } h = w_{MB} - 2 \text{ and } v > 0 \\ 1 & \text{if } h = w_{MB} - 1 \text{ and } 0 < v < h_{MB} - 1 \\ w_{MB} + 1 & \text{otherwise.} \end{cases} \tag{A.3}$$

$$c^4_{MC}(v \cdot w_{MB} + h) = \begin{cases} 0 & h = 0 \text{ or } v = 0 \\ 1 & \text{if } h < w_{MB} - 1 \text{ and } v > 0 \\ 2 & \text{if } h = w_{MB} - 1 \text{ and } 0 < v < h_{MB} - 1 \\ w_{MB} + 2 & \text{otherwise.} \end{cases} \tag{A.4}$$

$$r^4_{MC}(v \cdot w_{MB} + h) = \begin{cases} w_{MB} + h + 2 & h < w_{MB} - 1 \text{ and } v = 0 \\ 2w_{MB} & h = w_{MB} - 1 \text{ and } v = 0 \\ 2w_{MB} + 2 & h = (0, w_{MB} - 1) \text{ and } 0 < v < h_{MB} - 1 \\ 2w_{MB} + 3 & 0 < h < w_{MB} - 1 \text{ and } 0 < v < h_{MB} - 1 \\ w_{MB} + 2 & h = (0, w_{MB} - 1) \text{ and } v = h_{MB} - 1 \\ w_{MB} + 3 & \text{otherwise.} \end{cases} \tag{A.5}$$

$$c^{12}_{IC\&CC}(v \cdot w_{MB} + h) = 6p^4_{IC\&CC}(v \cdot w_{MB} + h) \tag{A.6}$$

A.2 Encoder Buffer Length Calculations

This section checks the conditions on the non-destructive read-back edge in the (IC&CC,ME,MC) cluster and calculates the buffer capacities of the (MC,TC,TU) cluster and the (TC,EC) chain.

A.2.1 IC&CC, ME and MC Cluster

Edges e_4 of the (IC&CC,ME,MC) cluster supports non-destructive read-back by the IC&CC. Table A.1 builds a sequential schedule to allow the checking of the conditions on such special channel (Equation (3.34)).

Table A.1: Tracking the conditions on reading back from edge e_4 (buffer YUV) with $h_{MB} - 1 = h'_{MB}$ and $h_{MB} - 2 = h''_{MB}$.

i	$A^4_{IC\&CC}(i+1)$	$C^4_{MC}(i+1)$	$P^4_{IC\&CC}(i+1) - A^4_{IC\&CC}(i)$	$r^4_{IC\&CC}(\mathrm{i})$
0	0	0	$w_{MB}+2$	$w_{MB}+1$
1	0	0	$w_{MB}+3$	3
2	0	0	$w_{MB}+4$	4
... continue until the one but last element on the first row				
$w_{MB}-2$	0	0	$2w_{MB}$	w_{MB}
$w_{MB}-1$	0	0	$2w_{MB}$	$2w_{MB}$
w_{MB}	2	0	$2w_{MB}+2$	$2w_{MB}+1$
$w_{MB}+1$	3	1	$2w_{MB}+1$	$w_{MB}+1$
$w_{MB}+2$	4	2	$2w_{MB}+1$	$w_{MB}+1$
... continue until the two but last element on the current row				
$2w_{MB}-3$	$w_{MB}-1$	$w_{MB}-3$	$2w_{MB}+1$	$w_{MB}+1$
$2w_{MB}-2$	$w_{MB}-1$	$w_{MB}-2$	$2w_{MB}+1$	$w_{MB}+1$
$2w_{MB}-1$	w_{MB}	w_{MB}	$2w_{MB}+1$	$2w_{MB}+1$
$2w_{MB}$	$w_{MB}+2$	w_{MB}	$2w_{MB}+2$	$2w_{MB}+1$
$2w_{MB}+1$	$w_{MB}+3$	$w_{MB}+1$	$2w_{MB}+1$	$w_{MB}+1$
$2w_{MB}+2$	$w_{MB}+4$	$w_{MB}+2$	$2w_{MB}+1$	$w_{MB}+1$
... continue until the two but last element on the one but last row				
$h'_{MB}w_{MB}-3$	$h''_{MB}w_{MB}-1$	$h''_{MB}w_{MB}-3$	$2w_{MB}+1$	$w_{MB}+1$
$h'_{MB}w_{MB}-2$	$h''_{MB}w_{MB}-1$	$h''_{MB}w_{MB}-2$	$2w_{MB}+1$	$w_{MB}+1$
$h'_{MB}w_{MB}-1$	$h''_{MB}w_{MB}$	$h''_{MB}w_{MB}$	$2w_{MB}+1$	$2w_{MB}+1$
$h'_{MB}w_{MB}$	$h''_{MB}w_{MB}+2$	$h''_{MB}w_{MB}$	$2w_{MB}$	$w_{MB}+2$
$h'_{MB}w_{MB}+1$	$h''_{MB}w_{MB}+3$	$h''_{MB}w_{MB}+1$	$2w_{MB}-2$	$w_{MB}+1$
$h'_{MB}w_{MB}+2$	$h''_{MB}w_{MB}+4$	$h''_{MB}w_{MB}+2$	$2w_{MB}-3$	$w_{MB}+1$
... continue until the two but last element on the last row				
$h_{MB}w_{MB}-3$	$h'_{MB}w_{MB}-1$	$h'_{MB}w_{MB}-3$	$w_{MB}+2$	$w_{MB}+1$
$h_{MB}w_{MB}-2$	$h'_{MB}w_{MB}-1$	$h'_{MB}w_{MB}-2$	$w_{MB}+1$	$w_{MB}+1$
$h_{MB}w_{MB}-1$	$h_{MB}w_{MB}$	$h_{MB}w_{MB}$	$w_{MB}+1$	$w_{MB}+1$

A.2.2 MC, TC and TU Cluster

In the (MC,TC,TU) cluster (Figure A.1), all basic repetition rates equal 1. The relative response times (derived using the basis repetition rates of q^b) are $\frac{1}{6}$. Table A.2 lists the buffer capacities calculation of edges e_6, e_7 and e_8 in this cluster.

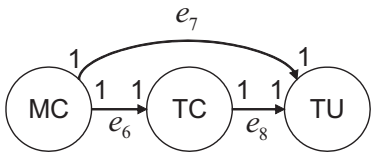


Figure A.1: MC, TC and MU cluster.

Table A.2: Tracking the required buffer size of the edges in the (MC,TC,TU) cluster.

Actor firing	Time (RT)		# on e_6		# on e_7		# on e_8	
	t_{start}	t_{end}	req	end	req	end	req	end
MC	0	$\frac{1}{6}$	1	1	1	1	0	0
MC ‖ TC	$\frac{1}{6}$	$\frac{2}{6}$	2	1	2	2	1	1
MC ‖ TC ‖ TU	$\frac{1}{6}$	$\frac{3}{6}$	2	1	3	2	2	1
maximum			2		3		2	

A.2.3 TC and EC Chain

In (TC,EC) chain (Figure A.2), the basic repetition rates of the TC and EC are respectively 6 and 1. The relative response times (derived using the basis repetition rates of q^b) are respectively $\frac{1}{6}$ and 1. Table A.3 lists the buffer capacity calculation of edge e_9 in this chain.

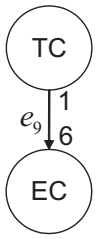


Figure A.2: TC and EC chain.

Table A.3: Tracking the required buffer size of the edge in the (TC,EC) chain.

Actor firing	Time (RT)		# on e_9	
	t_{start}	t_{end}	req	end
TC	0	$\frac{1}{6}$	1	1
...				
TC	$\frac{5}{6}$	1	6	6
TC ‖ EC	1	$1+\frac{1}{6}$	7	7
...				
TC ‖ EC	$1+\frac{5}{6}$	2	12	6
maximum		12		

Appendix B

Testbench Details

Information of all 30 test cases in the complete testbench is listed in this appendix, to complete the results of Chapter 5 and 6.

B.1 Testbench Properties

The testbench, listed in Table B.1, contains coding samples of different resolutions, framerates and movement complexities.

Table B.1: Detailed testbench properties of the 30 test cases.

Sequence name	Frame rate	Target bitrate (kbps)	# frames	# I frames	Width	Height	Bitrate (kbps)	Compression factor	PSNR Y
Mother and Daughter QCIF	15	20	150	1	176	144	20.08	227	33.89
Mother and Daughter QCIF	30	60	300	1	176	144	61.56	148	36.50
Mother and Daughter QCIF	30	75	300	1	176	144	75.56	121	37.16
Mother and Daughter CIF	15	100	150	1	352	288	99.54	183	37.61
Mother and Daughter CIF	30	120	300	1	352	288	120.92	302	36.92
Mother and Daughter CIF	30	200	300	1	352	288	207.25	176	38.72
Foreman QCIF	12.5	50	150	1	176	144	50.13	76	31.10
Foreman QCIF	25	150	300	1	176	144	150.29	51	34.08
Foreman QCIF	30	200	300	1	176	144	200.34	46	34.53
Foreman CIF	12.5	150	150	1	352	288	150.17	101	31.41
Foreman CIF	15	400	150	1	352	288	396.69	46	34.58
Foreman CIF	25	450	300	1	352	288	450.36	68	34.30
Foreman CIF	30	800	300	1	352	288	810.09	45	35.81
Calendar and Mobile	10	300	100	1	176	144	300.28	10	32.12
Calendar and Mobile	30	500	300	1	176	144	517.29	18	29.38
Calendar and Mobile	10	1000	100	1	352	288	998.88	12	32.43
Calendar and Mobile	15	1500	150	1	352	288	1489.35	12	32.52
Calendar and Mobile	15	2000	150	1	352	288	2000.11	9	34.45
Calendar and Mobile	30	3000	300	1	352	288	3017.09	12	32.94
Harbour	15	100	150	5	176	144	100.20	46	28.91
Harbour	30	500	300	5	352	288	519.01	70	28.36
Harbour	30	2000	300	5	704	576	2054.12	71	31.52
Crew QCIF	15	200	150	5	176	144	190.30	24	34.77
Crew CIF	30	1000	300	5	352	288	1044.62	35	36.50
Crew 4CIF	30	4000	300	5	704	576	4191.27	35	38.29
Crew 4CIF	30	6000	300	5	704	576	5669.58	26	39.41
City QCIF	15	200	150	5	176	144	192.84	24	35.91
City CIF	30	1000	300	5	352	288	999.08	37	34.85
City 4CIF	30	4000	300	5	704	576	4215.78	35	36.27
City 4CIF	30	6000	300	5	704	576	6311.93	23	38.01

B.2 High-Level Optimizations Results

To evaluate the effect of the high-level optimizations applied in section 5.2, Table B.2 lists the limited impact on the compression performance while Table B.3 and B.4 measure the complexity reduction, respectively in terms of a decrease of memory accesses and an improvement of performance.

Table B.2: Compression performance after the sequential phase.

Test case	Bitrate (kbps)	Delta bitrate (%)	PSNR Y	Delta PSNR
M&D QCIF 20 kbps 15 fps	19.97	-0.55	33.84	-0.05
M&D QCIF 60 kbps 30 fps	64.66	5.04	36.61	0.11
M&D QCIF 75 kbps 30 fps	77.44	2.49	37.31	0.15
M&D CIF 100 kbps 15 fps	99.29	-0.26	37.40	-0.21
M&D CIF 120 kbps 30 fps	123.10	1.80	36.82	-0.10
M&D CIF 200 kbps 30 fps	211.41	2.01	38.70	-0.02
Foreman QCIF 50 kbps 12.5 fps	50.01	-0.23	30.58	-0.52
Foreman QCIF 150 kbps 25 fps	150.40	0.07	33.65	-0.43
Foreman QCIF 200 kbps 30 fps	200.31	-0.02	34.15	-0.38
Foreman CIF 150 kbps 12.5 fps	150.04	-0.09	31.07	-0.34
Foreman CIF 400 kbps 15 fps	396.78	0.02	34.38	-0.20
Foreman CIF 450 kbps 25 fps	449.81	-0.12	33.79	-0.51
Foreman CIF 800 kbps 30 fps	824.47	1.77	35.50	-0.31
C&M QCIF 300 kbps 10 fps	299.31	-0.32	32.13	0.01
C&M QCIF 500 kbps 30 fps	524.67	1.43	29.45	0.07
C&M CIF 1 Mbps 10 fps	998.80	-0.01	32.34	-0.09
C&M CIF 1.5 Mbps 15 fps	1492.29	0.20	32.54	0.02
C&M CIF 2 Mbps 15 fps	1999.19	-0.05	34.49	0.04
C&M CIF 3 Mbps 30 fps	3050.10	1.09	33.02	0.08
Harbour QCIF 100 kbps 15 fps	100.30	0.10	28.96	0.05
Harbour CIF 500 kbps 30 fps	531.96	2.49	28.46	0.10
Harbour 4CIF 2 Mbps 30 fps	2053.92	-0.01	31.51	-0.01
Crew QCIF 200 kbps 15 fps	192.86	1.35	34.57	-0.20
Crew CIF 1 Mbps 30 fps	1000.37	-4.24	36.09	-0.41
Crew 4CIF 4 Mbps 30 fps	3999.33	-4.58	37.90	-0.39
Crew 4CIF 6 Mbps 30 fps	6019.04	6.16	39.59	0.18
City QCIF 200 kbps 15 fps	194.39	0.80	35.96	0.05
City CIF 1 Mbps 30 fps	1021.11	2.21	35.01	0.16
City 4CIF 4 Mbps 30 fps	4190.83	-0.59	36.31	0.04
City 4CIF 6 Mbps 30 fps	6338.13	0.42	38.06	0.05

Table B.3: Effect of the high-level optimizations on the access frequency and the peak memory usage.

Test case	Access Frequency (10^6 acc./s)		Reduction factor	Peak memory usage (kB)		Reduction factor
	Initial	OptSeq		Initial	OptSeq	
M&D QCIF 20 kbps 15 fps	692.0	39.4	17.6	2816.7	131.9	21.4
M&D QCIF 60 kbps 30 fps	1321.1	83.6	15.8	2816.7	131.9	21.4
M&D QCIF 75 kbps 30 fps	1310.0	85.2	15.4	2816.7	131.9	21.4
M&D CIF 100 kbps 15 fps	2829.0	149.0	19.0	5635.7	354.6	15.9
M&D CIF 120 kbps 30 fps	5710.1	281.8	20.3	5635.7	354.6	15.9
M&D CIF 200 kbps 30 fps	5554.8	300.0	18.5	5635.7	354.6	15.9
Foreman QCIF 50 kbps 12.5 fps	607.6	42.1	14.4	2969.6	131.0	22.7
Foreman QCIF 150 kbps 25 fps	1145.2	88.8	12.9	2969.6	131.0	22.7
Foreman QCIF 200 kbps 30 fps	1368.9	108.1	12.7	2969.6	131.0	22.7
Foreman CIF 150 kbps 12.5 fps	2606.6	153.5	17.0	5656.7	354.7	15.9
Foreman CIF 400 kbps 15 fps	3009.8	204.7	14.7	5656.7	354.7	15.9
Foreman CIF 450 kbps 25 fps	4945.8	326.1	15.2	5656.7	354.7	15.9
Foreman CIF 800 kbps 30 fps	5853.2	418.0	14.0	10485.1	354.7	29.6
C&M QCIF 300 kbps 10 fps	520.3	46.6	11.2	2982.6	131.9	22.6
C&M QCIF 500 kbps 30 fps	1590.6	134.5	11.8	2982.6	131.9	22.6
C&M CIF 1 Mbps 10 fps	2092.2	176.9	11.8	5666.8	354.7	16.0
C&M CIF 1.5 Mbps 15 fps	3118.6	265.5	11.7	5666.8	354.7	16.0
C&M CIF 2 Mbps 15 fps	3122.0	273.6	11.4	9913.3	354.6	28.0
C&M CIF 3 Mbps 30 fps	6217.7	536.9	11.6	10015.4	354.7	28.2
Harbour QCIF 100 kbps 15 fps	716.6	57.4	12.5	2977.1	131.9	22.6
Harbour CIF 500 kbps 30 fps	5700.6	445.8	12.8	10485.3	354.7	29.6
Harbour 4CIF 2 Mbps 30 fps	21448.4	1702.1	12.6	32750.1	1244.7	26.3
Crew QCIF 200 kbps 15 fps	782.5	61.5	12.7	2816.7	131.9	21.4
Crew CIF 1 Mbps 30 fps	6206.0	452.0	13.7	5645.2	354.7	15.9
Crew 4CIF 4 Mbps 30 fps	25654.0	1777.9	14.4	32727.1	1245.6	26.3
Crew 4CIF 6 Mbps 30 fps	25722.4	1945.7	13.2	34797.8	1244.1	28.0
City QCIF 200 kbps 15 fps	752.0	61.3	12.3	2970.1	131.9	22.5
City CIF 1 Mbps 30 fps	5999.6	465.4	12.9	6850.6	354.7	19.3
City 4CIF 4 Mbps 30 fps	23444.4	1829.8	12.8	34800.7	1245.7	27.9
City 4CIF 6 Mbps 30 fps	23324.5	1941.0	12.0	32848.7	1244.1	26.4

Table B.4: Performance improvement of the high-level optimized encoder on a RISC processor at 440 MHz.

Test case	Initial	OptSeq	Speed up factor
M&D QCIF 20 kbps 15 fps	1.1	13	12.2
M&D QCIF 60 kbps 30 fps	1.1	12	10.7
M&D QCIF 75 kbps 30 fps	1.1	12	10.3
M&D CIF 100 kbps 15 fps	0.25	3.4	13.4
M&D CIF 120 kbps 30 fps	0.26	3.6	14.2
M&D CIF 200 kbps 30 fps	0.26	3.3	12.7
Foreman QCIF 50 kbps 12.5 fps	1.0	10	9.7
Foreman QCIF 150 kbps 25 fps	1.1	9.3	8.6
Foreman QCIF 200 kbps 30 fps	1.1	9.2	8.5
Foreman CIF 150 kbps 12.5 fps	0.24	2.8	11.7
Foreman CIF 400 kbps 15 fps	0.25	2.4	9.8
Foreman CIF 450 kbps 25 fps	0.25	2.6	10.4
Foreman CIF 800 kbps 30 fps	0.25	2.4	9.4
C&M QCIF 300 kbps 10 fps	0.90	6.5	7.1
C&M QCIF 500 kbps 30 fps	0.91	6.8	7.5
C&M CIF 1 Mbps 10 fps	0.23	1.7	7.7
C&M CIF 1.5 Mbps 15 fps	0.23	1.7	7.6
C&M CIF 2 Mbps 15 fps	0.23	1.7	7.3
C&M CIF 3 Mbps 30 fps	0.23	1.7	7.5
Harbour QCIF 100 kbps 15 fps	1.0	8.5	8.2
Harbour CIF 500 kbps 30 fps	0.26	2.3	8.8
Harbour 4CIF 2 Mbps 30 fps	0.067	0.59	8.8
Crew QCIF 200 kbps 15 fps	0.98	7.7	7.8
Crew CIF 1 Mbps 30 fps	0.24	2.1	8.8
Crew 4CIF 4 Mbps 30 fps	0.058	0.54	9.4
Crew 4CIF 6 Mbps 30 fps	0.058	0.48	8.3
City QCIF 200 kbps 15 fps	0.99	7.8	7.9
City CIF 1 Mbps 30 fps	0.24	2.1	8.6
City 4CIF 4 Mbps 30 fps	0.060	0.53	8.8
City 4CIF 6 Mbps 30 fps	0.060	0.49	8.2

B.3 Intelligent Block Processing Results

The algorithmic optimization applied to the texture coding in Chapter 6 needs to correctly determine the thresholds T_{skip} and T_{DCT8} so that the coding status of a block is accurately predicted. Table B.5 lists the behavior for different threshold settings. The resulting prediction for $T_{skip} = 20$ and $T_{DCT8} = 30$ is given in Table B.6. Skipped blocks with zero MVs are added in Table B.7.

Table B.5: Accuracy of the prediction formula for different threshold settings ($T_{skip} = \{15, 20, 25\}$ and $T_{DCT8} = \{25, 30, 35\}$).

Test	T_{skip}	T_{DCT8}	skipped				DCT8		
			correct	asDCT8	asDCT64	wrong	correct	asDCT64	wrong
M&D QCIF 20 kbps 15 fps	15	25	55.1	26.9	17.9	0.0	13.6	86.4	2.9
	20	30	70.9	20.3	8.8	0.8	31.6	66.5	11.7
	25	35	82.3	14.6	3.1	6.5	43.5	42.0	27.7
M&D QCIF 60 kbps 30 fps	15	25	45.9	29.2	24.9	0.0	10.3	89.6	1.5
	20	30	64.1	20.4	15.5	0.6	23.1	75.1	5.5
	25	35	75.7	16.8	7.6	3.8	33.4	55.1	14.9
M&D QCIF 75 kbps 30 fps	15	25	46.1	29.1	24.8	0.0	10.1	89.9	1.4
	20	30	63.5	20.6	15.9	0.5	22.7	75.3	4.7
	25	35	76.0	15.3	8.7	3.5	30.9	57.5	11.3
M&D CIF 100 kbps 15 fps	15	25	60.9	27.0	12.1	0.0	21.3	78.6	3.7
	20	30	77.2	17.8	5.0	1.2	43.1	53.9	13.7
	25	35	88.0	10.6	1.4	9.8	47.9	29.2	28.6
M&D CIF 120 kbps 30 fps	15	25	59.9	27.3	12.7	0.0	21.5	78.5	4.4
	20	30	76.3	18.7	5.0	0.9	45.3	52.7	17.5
	25	35	87.3	11.4	1.3	10.6	50.4	27.1	36.6
M&D CIF 200 kbps 30 fps	15	25	54.3	29.6	16.1	0.0	18.7	81.3	2.9
	20	30	72.3	20.3	7.4	1.1	39.5	57.5	11.4
	25	35	84.1	13.5	2.4	7.9	47.2	32.4	26.7
Foreman QCIF 50 kbps 12.5 fps	15	25	47.7	33.1	19.2	0.0	16.2	83.8	2.1
	20	30	67.4	22.7	9.9	0.7	33.7	64.1	8.4
	25	35	81.2	14.4	4.4	6.5	38.3	44.5	17.8
Foreman QCIF 150 kbps 25 fps	15	25	36.4	38.0	25.6	0.1	18.3	81.3	1.5
	20	30	58.2	28.5	13.3	1.0	34.3	61.3	5.8
	25	35	75.4	18.4	6.2	5.5	38.6	41.2	13.2

Continued on next page

Table B.5 – continued from previous page

Test	T_{skip}	T_{DCT8}	skipped				DCT8		
			correct	asDCT8	asDCT64	wrong	correct	asDCT64	wrong
Foreman QCIF 200 kbps 30 fps	15	25	35.2	38.2	26.6	0.1	19.3	80.3	1.4
	20	30	57.0	29.5	13.5	1.1	34.5	60.5	5.4
	25	35	74.5	19.2	6.3	5.5	37.1	41.3	12.2
Foreman CIF 150 kbps 12.5 fps	15	25	55.1	31.3	13.6	0.0	20.5	79.4	3.0
	20	30	74.7	18.6	6.7	1.1	40.2	57.4	10.4
	25	35	86.6	10.3	3.1	10.1	39.1	39.3	20.4
Foreman CIF 400 kbps 15 fps	15	25	41.1	37.8	21.1	0.1	19.3	80.5	1.5
	20	30	62.7	27.8	9.5	1.1	37.0	59.3	5.7
	25	35	79.7	16.5	3.8	6.7	39.4	39.5	12.5
Foreman CIF 450 kbps 25 fps	15	25	40.0	39.6	20.4	0.1	20.6	79.2	1.9
	20	30	63.4	26.9	9.8	1.3	38.6	57.6	7.2
	25	35	80.2	15.5	4.3	7.8	39.9	38.1	15.5
Foreman CIF 800 kbps 30 fps	15	25	35.5	39.3	25.2	0.1	19.8	79.9	1.4
	20	30	57.4	30.8	11.8	1.2	37.1	58.4	5.6
	25	30	75.7	19.6	4.7	6.5	40.1	38.2	12.7
C&M QCIF 300 kbps 10 fps	15	25	16.4	28.2	55.4	0.0	6.0	93.7	0.2
	20	30	31.0	28.4	40.6	0.1	15.3	83.2	0.7
	25	35	46.9	28.5	24.7	0.6	26.3	65.6	2.3
C&M QCIF 500 kbps 30 fps	15	25	10.7	24.5	64.9	0.0	5.3	94.7	0.1
	20	30	24.0	24.4	51.6	0.1	12.0	86.8	0.6
	25	35	37.1	26.7	36.2	0.5	19.8	74.0	2.0
C&M CIF 1 Mbps 10 fps	15	25	22.5	36.3	41.2	0.0	8.5	91.3	0.4
	20	30	41.3	34.9	23.8	0.2	22.1	75.7	1.7
	25	35	61.2	27.7	11.0	1.2	35.2	54.4	4.6
C&M CIF 1.5 Mbps 15 fps	15	25	22.6	36.8	40.7	0.0	9.9	89.8	0.4
	20	30	41.8	35.5	22.6	0.2	23.7	73.9	1.7
	25	35	61.8	28.0	10.3	1.3	36.2	52.0	4.7
C&M CIF 2 Mbps 15 fps	15	25	20.1	37.6	42.4	0.0	10.4	89.3	0.3
	20	30	39.7	36.7	23.7	0.2	24.2	73.0	1.4
	25	35	60.2	29.5	10.3	1.1	36.8	50.9	3.8
C&M CIF 3 Mbps 30 fps	15	25	20.0	36.8	43.2	0.0	10.6	89.1	0.4
	20	30	39.7	34.8	25.5	0.2	24.0	73.3	1.5
	25	35	59.1	28.8	12.1	1.2	35.3	52.0	4.4

Continued on next page

Table B.5 – continued from previous page

Test	T_{skip}	T_{DCT8}	skipped				DCT8		
			correct	asDCT8	asDCT64	wrong	correct	asDCT64	wrong
Harbour QCIF 100 kbps 15 fps	15	25	50.8	10.6	38.6	0.0	1.9	98.1	0.2
	20	30	57.4	12.1	30.5	0.1	5.3	94.3	1.2
	25	35	61.6	19.9	18.5	0.4	16.4	81.7	6.6
Harbour CIF 500 kbps 30 fps	15	25	48.2	17.5	34.3	0.0	3.8	96.2	0.9
	20	30	56.5	21.6	21.9	0.1	11.5	87.9	5.6
	25	35	65.7	24.5	9.8	1.4	28.5	67.6	17.2
Harbour 4CIF 2 Mbps 30 fps	15	25	53.2	30.1	16.8	0.0	9.5	90.5	3.3
	20	30	69.3	24.5	6.1	0.5	26.3	72.2	13.6
	25	35	83.3	15.5	1.3	4.5	45.9	44.4	28.5
Crew QCIF 200 kbps 15 fps	15	25	31.9	36.8	31.2	0.1	9.8	89.8	1.0
	20	30	51.4	33.5	15.1	0.6	23.5	74.5	3.8
	25	30	70.6	24.0	5.4	3.3	34.0	55.3	8.7
Crew CIF 1 Mbps 30 fps	15	25	33.6	42.8	23.6	0.0	12.5	87.5	1.7
	20	30	57.4	32.9	9.7	0.6	30.9	67.5	6.4
	25	35	77.9	19.1	3.0	5.6	40.8	45.7	13.6
Crew 4CIF 4 Mbps 30 fps	15	25	31.1	45.5	23.4	0.0	9.8	90.2	1.2
	20	30	56.0	35.3	8.7	0.4	28.0	71.0	5.4
	25	35	78.1	19.7	2.3	5.0	40.2	49.0	12.3
Crew 4CIF 6 Mbps 30 fps	15	25	20.9	40.2	39.0	0.0	3.9	96.1	0.4
	20	30	40.8	40.5	18.7	0.1	15.7	84.0	2.1
	25	30	63.3	30.7	6.0	1.8	30.6	64.7	6.4
City QCIF 200 kbps 15 fps	15	25	21.6	48.3	30.1	0.0	15.8	84.2	0.7
	20	30	50.8	31.5	17.6	0.3	34.6	63.5	2.3
	25	35	72.2	17.8	10.0	3.0	31.7	48.6	4.6
City CIF 1 Mbps 30 fps	15	25	35.0	36.9	28.1	0.0	16.8	83.2	0.8
	20	30	58.8	23.1	18.1	0.7	25.7	70.5	2.9
	25	35	72.5	17.8	9.7	3.6	23.4	57.7	7.2
City 4CIF 4 Mbps 30 fps	15	25	41.1	35.0	23.8	0.0	15.3	84.6	0.9
	20	30	62.8	24.6	12.6	0.8	25.9	70.3	3.8
	25	35	76.7	18.3	5.0	4.2	29.5	53.3	9.7
City 4CIF 6 Mbps 30 fps	15	25	30.8	43.6	25.6	0.0	18.0	82.0	0.4
	20	30	58.4	28.0	13.6	0.6	34.0	63.0	1.8
	25	35	75.8	18.1	6.0	4.2	31.6	46.9	4.9

Table B.6: Block type proportions and predictions using $T_{skip} = 20$ and $T_{DCT8} = 30$.

Test case	Intra (%)	No prediction		Predicted		
		DCT64 (%)	Skipped (%)	DCT64 (%)	DCT8 (%)	Skipped (%)
M&D QCIF 20 kbps 15 fps	0.9	10.1	89.0	16.1	19.9	63.1
M&D QCIF 60 kbps 30 fps	0.4	15.5	84.1	27.0	18.6	54.0
M&D QCIF 75 kbps 30 fps	0.4	19.4	80.2	30.5	18.1	51.0
M&D CIF 100 kbps 15 fps	1.3	13.0	85.7	14.1	18.3	66.3
M&D CIF 120 kbps 30 fps	0.5	8.5	91.0	10.7	19.3	69.5
M&D CIF 200 kbps 30 fps	0.5	14.6	84.9	18.0	20.1	61.5
Foreman QCIF 50 kbps 12.5 fps	6.6	25.2	68.2	27.9	19.4	46.2
Foreman QCIF 150 kbps 25 fps	2.3	36.8	60.8	40.2	21.6	35.8
Foreman QCIF 200 kbps 30 fps	2.5	39.0	58.5	42.2	21.6	33.8
Foreman CIF 150 kbps 12.5 fps	9.1	17.1	73.8	18.1	17.5	55.3
Foreman CIF 400 kbps 15 fps	10.9	32.0	57.1	32.6	20.3	36.2
Foreman CIF 450 kbps 25 fps	5.0	27.4	67.5	29.2	22.6	43.1
Foreman CIF 800 kbps 30 fps	4.9	35.8	59.4	37.6	23.0	34.5
C&M QCIF 300 kbps 10 fps	1.1	81.2	17.8	87.1	6.2	5.6
C&M QCIF 500 kbps 30 fps	0.3	73.0	26.6	85.7	7.5	6.4
C&M CIF 1 Mbps 10 fps	1.7	70.9	27.4	75.0	11.8	11.5
C&M CIF 1.5 Mbps 15 fps	0.9	73.2	26.0	76.5	11.6	11.0
C&M CIF 2 Mbps 15 fps	0.8	78.4	20.8	81.1	9.7	8.4
C&M CIF 3 Mbps 30 fps	0.4	74.9	24.8	78.8	10.8	10.0
Harbour QCIF 100 kbps 15 fps	3.4	41.0	55.6	57.2	7.4	31.9
Harbour CIF 500 kbps 30 fps	1.7	34.4	63.9	46.1	16.0	36.2
Harbour 4CIF 2 Mbps 30 fps	1.9	36.5	61.6	34.3	20.9	42.9
Crew QCIF 200 kbps 15 fps	15.6	58.2	26.2	57.2	13.4	13.8
Crew CIF 1 Mbps 30 fps	11.1	48.4	40.5	45.2	20.2	23.5
Crew 4CIF 4 Mbps 30 fps	18.3	45.3	36.4	41.9	19.2	20.6
Crew 4CIF 6 Mbps 30 fps	18.1	57.1	24.8	58.0	13.7	10.2
City QCIF 200 kbps 15 fps	3.4	57.4	39.3	60.8	15.8	20.1
City CIF 1 Mbps 30 fps	1.7	47.8	50.5	53.6	14.7	30.0
City 4CIF 4 Mbps 30 fps	1.7	48.2	50.0	50.2	16.3	31.8
City 4CIF 6 Mbps 30 fps	1.1	59.8	39.2	60.3	15.4	23.2

Table B.7: Block type proportions including skipped blocks with zero motion vectors.

Test case	Intra (%)	DCT64 (%)	Skipped MVs not 0 (%)	Skipped MVs 0 (%)
M&D QCIF 20 kbps 15 fps	0.9	10.1	16.1	72.8
M&D QCIF 60 kbps 30 fps	0.4	15.5	8.7	75.4
M&D QCIF 75 kbps 30 fps	0.4	19.4	7.7	72.5
M&D CIF 100 kbps 15 fps	1.3	13.0	21.6	64.1
M&D CIF 120 kbps 30 fps	0.5	8.5	15.7	75.3
M&D CIF 200 kbps 30 fps	0.5	14.6	14.0	70.9
Foreman QCIF 50 kbps 12.5 fps	6.6	25.2	48.6	19.6
Foreman QCIF 150 kbps 25 fps	2.3	36.8	34.7	26.2
Foreman QCIF 200 kbps 30 fps	2.5	39.0	32.9	25.6
Foreman CIF 150 kbps 12.5 fps	9.1	17.1	56.8	17.1
Foreman CIF 400 kbps 15 fps	10.9	32.0	45.7	11.4
Foreman CIF 450 kbps 25 fps	5.0	27.4	43.8	23.7
Foreman CIF 800 kbps 30 fps	4.9	35.8	39.7	19.7
C&M QCIF 300 kbps 10 fps	1.1	81.2	17.2	0.6
C&M QCIF 500 kbps 30 fps	0.3	73.0	13.3	13.3
C&M CIF 1 Mbps 10 fps	1.7	70.9	27.0	0.4
C&M CIF 1.5 Mbps 15 fps	0.9	73.2	25.1	0.9
C&M CIF 2 Mbps 15 fps	0.8	78.4	20.0	0.7
C&M CIF 3 Mbps 30 fps	0.4	74.9	21.2	3.6
Harbour QCIF 100 kbps 15 fps	3.4	41.0	16.6	39.0
Harbour CIF 500 kbps 30 fps	1.7	34.4	24.5	39.4
Harbour 4CIF 2 Mbps 30 fps	1.9	36.5	36.4	25.2
Crew QCIF 200 kbps 15 fps	15.6	58.2	20.4	5.7
Crew CIF 1 Mbps 30 fps	11.1	48.4	30.6	9.9
Crew 4CIF 4 Mbps 30 fps	18.3	45.3	28.0	8.4
Crew 4CIF 6 Mbps 30 fps	18.1	57.1	18.7	6.2
City QCIF 200 kbps 15 fps	3.4	57.4	37.4	1.9
City CIF 1 Mbps 30 fps	1.7	47.8	47.6	2.9
City 4CIF 4 Mbps 30 fps	1.7	48.2	48.9	1.1
City 4CIF 6 Mbps 30 fps	1.1	59.8	38.3	0.8

Appendix C

Special Edges in Strict CSDF

Without the somewhat creative interpretation of CSDF, by decoupling the tokens from containers in Chapter 3, more expensive equivalents are needed to express the special channels in strict CSDF. The extra cost comes from a higher token exchange rate or the need for extra buffer capacity to temporally store data in the internal state.

This appendix first proposes an equivalent in strict CSDF for all special edges of Chapter 3. Then buffer sizes for the strict equivalents are recalculated, first for the example in Chapter 3, then for the MPEG-4 video encoder of Chapter 5.

C.1 Special Edges

By using a higher token exchange rate of more buffer space, all special channels of Chapter 3 can be represented as a strict CSDF equivalent.

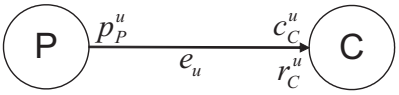
C.1.1 Non-Destructive Read

Figure C.1(b) proposes strict CSDF equivalents of the non-redundant read special edge. The top equivalent (1) is the most general case, using a duplicator actor D making copies of tokens to provide the consuming actor with the requested number of tokens. Equivalent (2) and (3) are instantiations of (1)

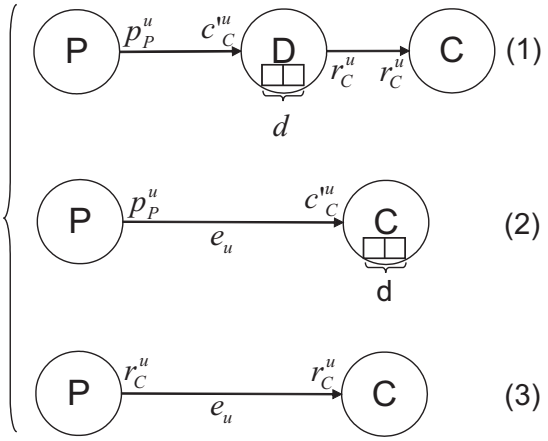
with the duplicator respectively moved inside the producing actor P or the consuming actor C.

The duplicator uses internal state to store the tokens $r(j)$ that are reused during the next firing. Their number is obtained with Equation (3.14). The required buffer size d equals the maximum number of reused tokens (Equation (C.1)). The consume sequence c'^u_D of the duplicator only consumes a number of tokens to make the sum with the reused tokens of the previous firing equal to the requested tokens to fire (Equation (3.18)).

$$d = \max_{0 \leq j \leq L_C - 1} (r(j)) \tag{C.1}$$



(a) Non-destructive read extension



(b) Non-destructive read equivalents

Figure C.1: Non-destructive reads between a producer P with period L_P and sequence $p_P^u = \{p_P^u(0), \ldots, p_P^u(L_P - 1)\}$ and a consumer C with period L_C and sequences $r_C^u = \{r_C^u(0), \ldots, r_C^u(L_C - 1)\}$ and $c_C^u = \{c_C^u(0), \ldots, c_C^u(L_C - 1)\}$ for which $c_C^u(j) \leq r_C^u(j)$.

Equivalent (3) causes a higher amount of transferred tokens than the non-destructive read special edge, since the requested number of tokens is always greater than or equal to the consumed one:

$$c_C^u(j) \leq r_C^u(j)$$

If the producer can not recalculate the reused tokens, equivalent (3) will also need buffer space in his internal state to store them.

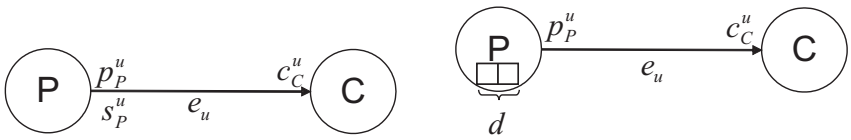
Equivalent (2) transfers the same number of tokens as the edge extension, but requires extra buffer space in the internal state of the consumer to store the reused tokens. It can be proved that, at any moment, the special edge buffer size is smaller than or equal to the edge buffer size of the equivalent incremented with the size for the internal state.

$$P_P^u(k) - C_C'^u(l) + \max_{0 \le j < l}(r(j)) \ge P_P^u(k) - C_C^u(l)$$

Note that all 3 equivalents cause more accesses either to store data in the internal state or during the recalculation.

C.1.2 Partial Update

The only CSDF equivalent (Figure C.2(b)) of the partial update edge extension stores the unfinished tokens $s(i)$ (Equation (3.25)) in its local state and only writes the tokens on the edge that will be committed at the end of the firing. The required buffer size d equals the maximum number of unfinished tokens.



(a) Partial update extension (b) Partial update equivalent

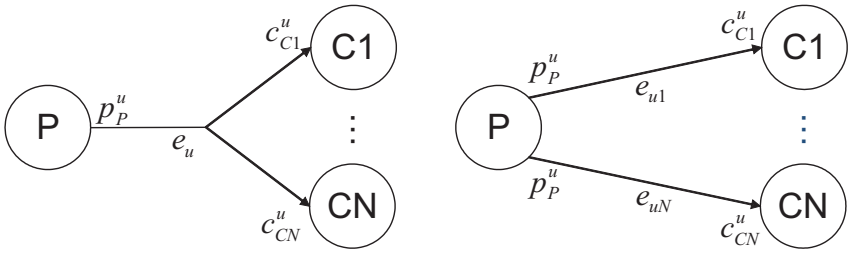
Figure C.2: Partial updates between a producer P with period L_P and sequences $p = \{p_P^u(0), \ldots, p_P^u(L_P - 1)\}$ and $s = \{s_P^u(0), \ldots, s_P^u(L_P - 1)\}$ for which $p_P^u(i) \le s_P^u(i)$ and a consumer C with period L_C and sequence $c = \{c_C^u(0), \ldots, c_C^u(L_C - 1)\}$.

The proposed CSDF equivalent exchanges the same number of tokens as the partial update extension, but requires extra buffer space for the internal state of the producer to store the unfinished tokens. It can be proved that, at any moment, the required special channel buffer size to allow firing of the producer is smaller than or equal to the required buffer size of the equivalent incremented with the size for the internal state.

$$s_P^u(k-1) + P_P^u(k-1) - C_C^u(l) \le P_P^u(k) - C_C^u(l) + \max_{0 \le i < k}(s(i))$$

C.1.2.1 Multiple Consumers

The CSDF equivalent (Figure C.3(b)) of the multiple consumer special channel (Figure C.3(a)) uses a separate edge e_{ub} to each consumer Cb.



(a) Multiple consumers extension (b) Multiple consumers equivalent

Figure C.3: Multiple consumers on an edge between a producer P with period L_P and sequence $p = \{p_P^u(0), \ldots, p_P^u(L_P - 1)\}$ and N consumers $C1, \ldots, CN$ with periods $L_{C1}, \ldots, L_{CN}$ and sequences $c1 = \{c_{C1}^u(0), \ldots, c_{C1}^u(L_{C1} - 1)\}, \ldots, cN = \{c_{CN}^u(0), \ldots, c_{CN}^u(L_{CN} - 1)\}$.

This leads to a N times higher number of tokens exchanged. Additionally, it can be proved that the equivalent requires as much or more buffer space than the special channel.

$$P_p^u(k) - C_{cc}^u(l_c) \leq \sum_{b=1}^{N} \left(P_p^{ub}(k) - C_{Cb}^{ub}(l_b) \right), \text{ with } l_c = \sum_{b=1}^{N} l_b$$

C.2 Example Buffer Length Recalculations

The example of Figure 3.26 used to illustrate the buffer length calculations of Chapter 3 is expressed with strict CSDF equivalents in Figure C.4. The d on edges e_1, e_3 and e_5 gives the required buffer for the local state, needed by the strict CSDF equivalent of the special channels.

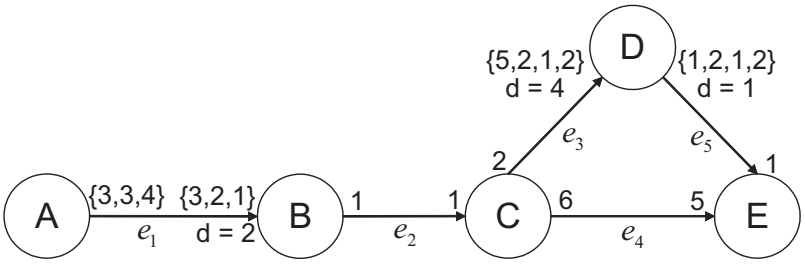


Figure C.4: Example CSDF graph of Figure 3.26 expressed with strict CSDF equivalents. The extra buffer space needed for the internal state of a special channel is indicated by d.

Tables C.3 and C.2 respectively dimension the edges in the chain and the

cluster. Between 10% and 30% extra buffer space is required compared to the special edges (Table C.1).

Table C.1: Buffer space increase of the CSDF equivalent.

Edge	Extension size	Equivalent size	Increase (%)
e_1	11	12	9.1
e_3	11	14	27.3
e_5	4	5	25.0

Table C.2: Tracking the required buffer size of the equivalent edges in the cluster (C,D,E) cluster.

Actor firing	Time (RT)		# on e_3		# on e_4		# on e_5	
	t_{start}	t_{end}	req	end	req	end	req	end
C	0	12	2	2	6	6	-	-
C	12	24	4	4	12	12	-	-
C	24	36	6	6	18	18	-	-
C ‖ D	36	48	8	8	24	24	1	-
C ‖ ...	48	51	10	3	30	-	1	1
C ‖ D ‖ E	51	60	10	5	30	30	3	-
C ‖ ... ‖ ...	60	61	7	-	36	25	3	0
... ‖ ... ‖ -	61	66	7	3	36	-	3	2
... ‖ D ‖ E	66	72	7	5	36	31	3	-
C ‖ ... ‖ ...	72	76	7	-	37	26	3	1
... ‖ ... ‖ E	76	81	7	4	37	-	3	2
... ‖ D ‖ ...	81	84	7	6	37	32	4	-
C ‖ ... ‖ ...	84	86	8	-	38	27	4	1
... ‖ ... ‖ E	86	96	8	6	38	28	4	2
C ‖ D ‖ E	96	106	8	-	34	23	3	1
... ‖ ... ‖ E	106	108	8	8	34	29	3	-
C ‖ ... ‖ ...	108	111	10	3	35	-	3	2
... ‖ D ‖ ...	111	116	10	-	35	24	4	1
... ‖ ... ‖ E	116	120	10	5	35	30	4	-
C ‖ ... ‖ ...	120	126	7	3	36	25	4	2
maximum			10		38		4	

Table C.3: Tracking the required buffer size of the equivalent edges in the chain (A,B,C).

Actor firing						Time (RT)		# on e_1		# on e_2	
						t_{start}	t_{end}	req	end	req	end
A						0	20	3	3	-	-
A	‖	B				20	32	6	0	1	1
...	‖	-	‖	C		32	40	6	3	-	-
A	‖	B	‖	...		40	44	7	-	2	0
...	‖	...	‖	-		44	52	7	1	2	1
...	‖	B	‖	C		52	60	7	5	2	-
A	‖	...	‖	...		60	64	8	4	2	1
...	‖	B	‖	C		64	76	8	1	2	1
...	‖	-	‖	C		76	80	8	4	-	-
A	‖	B	‖	...		80	88	7	-	2	0
...	‖	...	‖	-		88	92	7	2	2	1
...	‖	B	‖	C		92	100	7	5	2	-
A	‖	...	‖	...		100	104	9	4	2	1
...	‖	B	‖	C		104	116	9	1	2	1
...	‖	-	‖	C		116	120	9	5	-	-
A	‖	B	‖	...		120	128	8	-	2	0
...	‖	...	‖	-		128	132	8	3	2	1
...	‖	B	‖	C		132	140	8	6	2	1
A	‖	...	‖	...		140	144	9	5	2	1
...	‖	B	‖	C		144	156	9	2	2	1
...	‖	B	‖	C		156	160	9	5	2	-
A	‖	...	‖	...		160	168	9	3	2	1
...	‖	B	‖	C		168	180	9	6	2	1
A	‖	B	‖	C		180	192	9	3	2	1
...	‖	B	‖	C		192	200	9	6	2	-
A	‖	...	‖	...		200	204	9	4	2	1
...	‖	B	‖	C		204	216	9	3	2	1
...	‖	B	‖	C		216	220	9	6	2	1
A	‖	...	‖	...		220	228	10	3	2	1
...	‖	B	‖	C		228	240	10	5	2	1
A	‖	B	‖	C		240	252	8	4	2	1
...	‖	B	‖	C		252	260	8	7	2	-
A	‖	...	‖	...		260	264	10	4	2	1
A	‖	B	‖	C		264	276	10	2	2	1
...	‖	B	‖	C		276	280	10	5	2	-
A	‖	...	‖	...		280	288	9	4	2	1
...	‖	B	‖	C		288	300	9	5	2	1
A	‖	B	‖	C		300	312	8	3	2	1
maximum								10		2	

C.3 Encoder Buffer Length Recalculations

All special edges present in the CSDF graph of the MPEG-4 encoder (Figure 5.11) are recalculated in the next subsections using the equivalents in the strict interpretation.

C.3.1 Edge e_5 (motion vectors)

$d'^5_{MC} = 1$ and $c'^5_{MC} = \{1, 0, 0, 0, 0, 0\}$

Table C.4: Tracking the required buffer size of the equivalent edges in the simplified (IC&CC,ME,MC) cluster.

Actor firing							Time (RT)		# on e_1		# on e_3		# on e_5	
	t_{start}		t_{end}	req	end		req	end	req	end				
IC&CC							0	1	1	1	6	6	0	0
IC&CC	‖	ME					1	2	2	1	12	12	1	1
IC&CC	‖	ME	‖	MC			2	$2 + \frac{1}{6}$	2	-	18	11	2	0
...	‖	...	‖	MC			$2 + \frac{1}{6}$	$2 + \frac{2}{6}$	2	-	18	10	2	-
...	‖	...	‖	MC			$2 + \frac{2}{6}$	$2 + \frac{3}{6}$	2	-	18	9	2	-
...	‖	...	‖	MC			$2 + \frac{3}{6}$	$2 + \frac{4}{6}$	2	-	18	8	2	-
...	‖	...	‖	MC			$2 + \frac{4}{6}$	$2 + \frac{5}{6}$	2	-	18	7	2	-
...	‖	...	‖	MC			$2 + \frac{5}{6}$	3	2	1	18	12	2	1
maximum									2		18		2	

C.3.2 Edge e_2 (search area)

$$d'^2_{ME}(h) = \begin{cases} 0 & \text{if } h = w_{MB} - 1 \\ 2 & \text{otherwise.} \end{cases} \tag{C.2}$$

$$c'^2_{ME}(h) = \begin{cases} 3 & \text{if } h = 0 \\ 1 & \text{otherwise.} \end{cases} \tag{C.3}$$

Table C.5: Tracking the required buffer size of edge e_2 (search area) equivalent.

Actor firing	Time (RT)		# on e_2	
	t_{start}	t_{end}	req	end
IC&CC	0	1	3	3
IC&CC ‖ ME	1	2	4	1
IC&CC ‖ ME	1	2	2	1
...				
IC&CC ‖ ME	$w_{MB}-1$	w_{MB}	2	1
IC&CC ‖ ME	w_{MB}	$w_{MB}+1$	4	3
maximum			4	

C.3.3 Edge e_4 (buffer YUV)

$$d^4_{MC}(v{\cdot}w_{MB}{+}h) = \begin{cases} w_{MB}+h+2 & h < w_{MB}-1 \text{ and } v = 0 \\ 2w_{MB} & h = w_{MB}-1 \text{ and } 0 \le v < h_{MB}-1 \\ 2w_{MB}+2 & 0 \le h < w_{MB}-1 \text{ and } 0 < v < h_{MB}-1 \\ 2w_{MB} & h = 0 \text{ and } v = h_{MB}-1 \\ 2w_{MB}-h & 0 \le h < w_{MB}-1 \text{ and } v = h_{MB}-1 \\ 0 & \text{otherwise.} \end{cases} \tag{C.4}$$

$$c'^4_{MC}(v{\cdot}w_{MB}{+}h) = \begin{cases} w_{MB}+2 & h = 0 \text{ and } v = 0 \\ 1 & \text{if } 0 < h < w_{MB}-1 \text{ and } 0 \le v < h_{MB}-1 \\ 0 & \text{if } h = w_{MB}-1 \text{ or } v = h_{MB}-1 \\ 2 & \text{otherwise.} \end{cases} \tag{C.5}$$

Table C.6: Tracking the required buffer size of edge e_4 (buffer YUV) with $h_{MB}-1 = h'_{MB}$.

Actor firing	Time (RT)		# on e_4	
	t_{start}	t_{end}	req	end
IC&CC	0	1	$w_{MB}+2$	$w_{MB}+2$
IC&CC ‖ ME	1	2	$w_{MB}+3$	$w_{MB}+3$
IC&CC ‖ ME ‖ 6MC	2	3	$w_{MB}+4$	2
IC&CC ‖ ME ‖ 6MC	3	4	3	2
... continue until IC&CC reaches one but last element on the first row				
IC&CC ‖ ME ‖ 6MC	$w_{MB}-2$	$w_{MB}-1$	3	2
IC&CC ‖ ME ‖ 6MC	$w_{MB}-1$	w_{MB}	-	1
IC&CC ‖ ME ‖ 6MC	w_{MB}	$w_{MB}+1$	3	2
IC&CC ‖ ME ‖ 6MC	$w_{MB}+1$	$w_{MB}+2$	3	3
IC&CC ‖ ME ‖ 6MC	$w_{MB}+2$	$w_{MB}+3$	4	2
IC&CC ‖ ME ‖ 6MC	$w_{MB}+3$	$w_{MB}+4$	3	2
... continue until IC&CC reaches one but last element on the current row				
IC&CC ‖ ME ‖ 6MC	$2w_{MB}-2$	$2w_{MB}-1$	3	2
IC&CC ‖ ME ‖ 6MC	$2w_{MB}-1$	$2w_{MB}$	-	1
IC&CC ‖ ME ‖ 6MC	$2w_{MB}$	$2w_{MB}+1$	3	2
IC&CC ‖ ME ‖ 6MC	$2w_{MB}+1$	$2w_{MB}+2$	3	3
IC&CC ‖ ME ‖ 6MC	$2w_{MB}+2$	$2w_{MB}+3$	4	2
IC&CC ‖ ME ‖ 6MC	$2w_{MB}+3$	$2w_{MB}+4$	3	2
... continue until IC&CC reaches one but last element on the one but last row				
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}-2$	$h'_{MB}w_{MB}-1$	3	2
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}-1$	$h'_{MB}w_{MB}$	-	1
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}$	$h'_{MB}w_{MB}+1$	-	0
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}+1$	$h'_{MB}w_{MB}+2$	-	-
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}+2$	$h'_{MB}w_{MB}+3$	-	-
IC&CC ‖ ME ‖ 6MC	$h'_{MB}w_{MB}+3$	$h'_{MB}w_{MB}+4$	-	-
... continue until IC&CC reaches one but last element on the last row				
IC&CC ‖ ME ‖ 6MC	$h_{MB}w_{MB}-2$	$h_{MB}w_{MB}-1$	-	-
IC&CC ‖ ME ‖ 6MC	$h_{MB}w_{MB}-1$	$h_{MB}w_{MB}$	-	-
ME ‖ 6MC	$h_{MB}w_{MB}$	$h_{MB}w_{MB}+1$	-	-
6MC	$h_{MB}w_{MB}+1$	$h_{MB}w_{MB}+2$	-	-
maximum			$w_{MB}+4$	

Samenvatting

De huidige generatie multimedia-apparaten ondersteunt de uitwisseling van diverse media: spraak, tekst, audio, video, grafische elementen, enz. Deze moderne toestellen streven ernaar om de ervaring van de gebruiker te verbeteren door gebruik te maken van geavanceerde audio-visuele algoritmes en hogere beeldresoluties. De beperkte warmteafvoer en energievoorraad van draagbare toestellen vereisen echter implementaties met een laag energie-verbruik. Deze tegenstrijdige behoeften maken de uitdaging van een kost-efficiënt ontwerp moeilijker.

Deze doctoraatstudie stelt een ontwerpmethode voor die georiënteerd is naar laag verbruik, blok-gebaseerde videosignaalverwerking als brug tussen een hoog niveau beschrijving (typisch C code) en de uiteindelijke realisatie. De belangrijkste kost factor, de energie-efficiëntie, is uitgedrukt als een energie-vertragingsproduct, dat zowel het energieverbruik als de ondersteunde doorvoersnelheid bevat. De toegepaste ontwerpfilosofie rangschikt een verzameling van verscheidene technieken voor vermogenoptimalisatie over verschillende ontwerpniveaus, volgens de afweging tussen hun impact op het energie-vertragingsproduct en de geassocieerde toepassingskost: (i) een herdefinitie van het probleem om de complexiteit te reduceren, (ii) parallellisatie om de vertraging per taak te verlagen (terwijl de energie per taak constant blijft) en (iii) gespecialiseerde hardware toevoegen omdat deze de hoogste energie-efficientie heeft. De typische datadominantie van moderne multimediasystemen motiveert de geheugen- en communicatiefocus in de ontwerpstappen.

Om dit geheugen- en communicatieknelpunt aan te pakken, breidt de ontwerpmethode vooral de bestaande Data Transfer and Storage Exploration (DTSE) methodologie uit, omdat deze zich al concentreert op geheugenoptimalisaties. Op hoog niveau wordt verfijning van het algoritme toegevoegd, terwijl de lagere ontwerpniveaus uitgebreid zijn met een partitieexploratie gericht op

RTL-ontwikkeling (bijdrage 1, Hoofdstuk 2). Dit leidt tot een ontwerpmethode met twee fases.

De eerste fase van de ontwerpmethode herformuleert het probleem om zo de complexiteit te reduceren. Ze combineert geheugenoptimalisaties met algoritmische verfijning. Dit resulteert in een referentiecode die getransformeerd is naar een blok-gebaseerde videotoepassing met een gelokaliseerde signaalverwerking en datastroom. De vereiste geheugengrootte en geheugentoegang zijn geminimaliseerd. Deze optimalisaties bereiden het systeem voor op het introduceren van parallellisme en maken een referentie voor de verdere RTL-ontwikkeling klaar.

Om een goed abstractiemiddel te voorzien, om te redeneren over het parallellisme in de tweede fase, wordt een Cyclo-Static DataFlow (CSDF) model gebruikt. Zo een model komt goed overeen met het datastroom-gedomineerde gedrag van multimedia signaalverwerking. Tussen verschillende datastroom berekeningsmodellen is CSDF één van de meest expressieve, terwijl het volledige analyse potentieel behouden blijft (bv. consistentie controle, dead-lock analyse, enz.). Ook implementatiespecifieke aspecten van communicatiekanalen, zoals datahergebruik, gedeelde buffers of beperkte buffer groottes, kunnen in een CSDF-diagram uitgedrukt worden. Dit speciaal gedrag is vaak gerelateerd aan het gebruik van een vorm van gedeelde circulaire buffers. Deze thesis stelt voor om (speciale) communicatiekanalen uit te drukken in het datastroom-diagram als twee pijlen, om zo de synchronisatie tussen twee taken en de vrije bufferruimte correct te modelleren. Bijgevolg blijft het diagram volledig analyseerbaar en laat het toe te redeneren over het temporeel gedrag van de applicatie, die gemodelleerd wordt door het CSDF-diagram (bijdrage 2, Sectie 3.2).

Op basis van een CSDF-berekeningsmodel, op een specifieke wijze gebruikt om ook implementatiespecifieke aspecten uit te drukken, worden de nodige buffer capaciteiten van de communicatie kanalen uitgerekend om een volledig parallel en pipelined self-timed uitvoering van het systeem te bekomen (bijdrage 2, Sectie 3.4). Aan de hand van een hoog-niveau parallel SystemC-model, automatisch gegenereerd door de SPRINT-tool van de (sequentiële) geheugen geoptimaliseerde C-specificatie, wordt de correctheid van de buffergroottes en de graad van parallellisme gecontroleerd. Na deze voorafgaande verificatie wordt de RTL-link van SPRINT (bijdrage 3) gebruikt om het pad naar de hardware ontwikkeling op te starten.

De communicatiekanalen worden in gespecialiseerde hardware geïmplementeerd als een beperkte, maar voldoende set van communicatie primitieven (bijdrage 4, Sectie 3.5). Door het principe van scheiding van communicatie en berekening te gebruiken, kan elke taak afzonderlijk naar HDL vertaald worden terwijl zijn communicatie correct gemodelleerd wordt. Een geautomatiseerde

omgeving ondersteunt de functionele verificatie van elke aparte component door een combinatie van simulatie en fast prototyping. Deze uitgebreide verificatie van elke component verkort de debug-cyclus tijdens integratie (bijdrage 5, Chapter 4). Een significante verkorting van de ontwikkeltijd (meer dan een factor 2) is gemeten op het MPEG-4 ontwerp waar de RTL ontwikkel- en verificatieaanpak (zoals hierboven beschreven) strikt toegepast werd aan de encoder zijde, maar niet aan de decoder zijde.

De ontwerpmethode is gedemonstreerd met de ontwikkeling van een energie-efficiënte MPEG-4 Simple Profile video encoder en decoder, die respectievelijk 30 4CIF (704×576) en 30 XSGA (1280×1024) beelden per seconde kunnen verwerken, of (simultaan) meerdere sequenties van lagere resolutie (bijdrage 6, Hoofdstuk 5). Vertrekkend van de MPEG-referentiecode, worden eerst vereenvoudigingen aangebracht in het algoritme van de texture coding, de rate control en de motion estimation ten koste van een lichte degradatie van de visuele kwaliteit. Meer specifiek, in de texture coding is een intelligente blok verwerking geïntroduceerd, die de coding status van een blok na quantisatie voorspelt: alleen nullen, enkel een eerste rij of een eerste kolom met niet-nul waarden, of nog andere niet-nul waarden (bijdrage 7, Hoofdstuk 6). Geheugen optimalisaties transformeren de video codec in een (macro)blok-gebaseerd systeem. Naar hardware toe, wordt deze signaalverwerkingsketen gerealiseerd als een gespecialiseerde video-pipeline en is ze afgebeeld op een FPGA en op een ASIC. De implementatie bereikt een hoge graad van parallellisme, bevat een toegespitste geheugenhiërarchie en gebruikt het theoretische minimum aan burst-georiënteerde transfers naar het externe geheugen. De video codec core schaalt met een aantal parameters (zoals maximale beeldgrootte, het maximum aantal bitstreams, enz.), die door de gebruiker bij compilatie instelbaar zijn, om zo optimaal aan te sluiten bij zijn applicatie. De voorgestelde encoder HW implementatie verbruikt 71 mW (180 nm, 1.62 V UMC technologie), wanneer een 4CIF video sequentie aan 30 fps verwerkt wordt. Deze energie-efficiëntie, verkregen zonder gebruik te maken van een laag-vermogen CMOS bibliotheek, behoort tot de state of the art voor hoge resolutie video encoders (bijdrage 6, Sectie 5.5).

Bibliography

[1] http://www.imec.be/atomium.

[2] http://www.imec.be/design/sprint.

[3] http://www.annapmicro.com/wildcard2.html.

[4] http://www.synopsys.com.

[5] http://www.model.com.

[6] http://www.itrs.net/Links/2006Update/FinalToPost/02_Design_2006Update.pdf .

[7] *Information technology - Generic coding of audio-visual objects - Part 5: Reference software.* ISO/IEC 14496-5:2001, December 2001.

[8] *Information technology - Generic coding of audio-visual objects - Part 2: Visual.* ISO/IEC 14496-2:2004, June 2004.

[9] *Information Technology - Generic Coding of Audio-Visual Objects - Part 2: Visual, Amendment 2: New levels for Simple Profile.* Technical Report N6496, ISO/IEC JTC1/SC29WG11, 2004.

[10] Marleen Ade. *Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets.* PhD thesis, Katholieke Universiteit Leuven, 1996.

[11] Marleen Ade, Rudy Lauwereins, and Jean Peperstraete. Data memory minimization for synchronous data flow graphs emulated on DSP-FPGA targets. In: *Conference on Design Automation, DAC*, pp. 64–69, June 1997.

[12] Ihab Amer, Choudhury A. Rahman, Tamer Mohamed, Mohammed Sayed, and Wael Badawy. A hardware-accelerated framework with IP-blocks for application in MPEG-4. In: *International Workshop on System-on-Chip for Real-Time Applications, IWSOC*, pp. 211–214, July 2005.

[13] Ihab Amer, Mohammed Sayed, Wael Badawy, and Graham Jullien. On the way to an H.264 HW/SW reference model: a SystemC modeling strategy to integrate selected IP-blocks with the H.264 software reference model. In: *IEEE Workshop on Signal Processing Systems Design and Implementation, SIPS*, pp. 178–181, November 2005.

[14] Amphion. Standalone MPEG-4 video encoders, 2003. CS6701 product specification.

[15] Hideho Arakida, Masafumi Takahashi, Yoshiro Tsuboi, Tsuyoshi Nishikawa, Hideaki Yamamoto, Toshihide Fujiyoshi, Yoshiyuki Kitasho, Yasuyuki Ueda, Manabu Watanabe, Tetsuya Fujita, Toshihiro Terazawa, Kenji Ohmori, Masahiro Koana, Hiroki Nakamura, Eiichi Watanabe, Hirofumi Ando, Takeshi Aikawa, and Tohru Furuyama. A 160 mW, 80 nA standby, MPEG-4 audiovisual LSI with 16 Mb embedded DRAM and a 5 GOPS adaptive post filter. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC*, volume 1, pp. 42,476, February 2003.

[16] Nataniel J. August and Dong Sam Ha. Low power design of DCT and IDCT for low bit rate video codecs. *IEEE Transactions on Multimedia*, 6 (3): pp. 414–422, June 2004.

[17] V. Bhaskaran and K. Konstantinides. *Image and Video Compression Standards, Algorithms and Architectures*. Kluwer Academic Publishers, 1997.

[18] Shuvra S. Bhattacharyya, Sundararajan Sriram, and Edward A. Lee. Resynchronization for multiprocessor dsp systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47 (11): pp. 1597–1609, November 2000.

[19] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44 (2): pp. 397–408, February 1996.

[20] Jan Bormans, Kristof Denolf, Sven Wuytack, Lode Nachtergaele, and Ivo Bolsens. Integrating system-level low power methodologies into a real-life design flow. In: *International Workshop Power and Timing*

Modeling, Optimization and Simulation, PATMOS, pp. 19–28, October 1999.

[21] Erik Brockmeyer, Lode Nachtergaele, Francky Catthoor, Jan Bormans, and Hugo De Man. Low power memory storage and transfer organization for the MPEG-4 full pel motion estimation on a multimedia processor. *IEEE Transactions on Multimedia*, 1 (2): pp. 202–216, June 1999.

[22] Francky Catthoor, Sven Wuytack, Lode Nachtergaele, Eddy De Greef, F. Balasa, and Arnout Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[23] Nelson Yen-Chung Chang and Tian-Sheuan Chang. Combined frame memory architecture for motion compensation in video decoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 16 (10): pp. 1806–1809, October 2006.

[24] Yung-Chi Chang, Wei-Min Chao, and Liang-Gee Chen. Platform-based MPEG-4 video encoder soc design. In: *IEEE Workshop on Signal Processing Systems, SIPS*, pp. 251–256, 2004.

[25] T. Chiang and Y.-Q. Zhang. A new rate control schme using quadratic rate distortion model. *IEEE Transactions on Circuits and Systems for Video Technology*, 7 (1): pp. 246–250, February 1997.

[26] Shao-Yi Chien, Yu-Wen Huang, Ching-Yeh Chen, H.H. Chen, and Liang-Gee Chen. Hardware architecture design of video compression for multimedia communication systems. *IEEE Communications Magazine*, 43 (8): pp. 122–131, August 2005.

[27] Adrian Chirila-Rus, Kristof Denolf, Bart Vanhoof, Paul Schumacher, and Kees Vissers. Communication primitives driven hardware design and test methodology applied on complex video applications. In: *Rapid System Prototyping workshop, RSP*, pp. 246–248, June 2005.

[28] Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl. SPRINT: a tool to generate concurrent transaction level models from sequential code. *EURASIP Journal on Advances in Signal Processing, Special Issue on Transforming Signal Processing Applications into Parallel Implementations*, 2007: pp. Article ID 75373, 15 pages, 2007.

[29] Henk Corporaal. *PROGRESS White papers 2006*, Chapter Embedded Systems Design, pp. 7–27. Technologiestichting STW, Utrecht, 2006.

[30] David E. Culler. Convergence of parallel architectures. Course Notes, CS 258, 1999.

[31] Josef Dalcolmo, Rudy Lauwereins, and Marleen Ad. Code generation of data dominated DSP applications for FPGA targets. In: *IEEE International Workshop on Rapid System Prototyping*, pp. 162–167, June 1998.

[32] Abhijit Davare, Qi Zhu, John Moondanos, and Alberto Sangiovanni-Vincentelli. JPEG encoding on the Intel MXP5800: a platform-based design case study. In: *IEEE Workshop on Embedded Systems for Real-time Multimedia, ESTIMedia*, pp. 89–94, September 2005.

[33] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and Kees A. Vissers. Yapi: application modeling for signal processing systems. In: *Conference on Design Automation, DAC*, pp. 402–405, June 2000.

[34] Christophe De Vleeschouwer. Model-based rate control implementation for low-power video communication systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 13 (12): pp. 1187–1194, December 2003.

[35] Christophe De Vleeschouwer, Tord Nilsson, Kristof Denolf, and Jan Bormans. Algorithmic and architectural co-design of a motion-estimation engine for low-power video devices. *IEEE Transactions on Circuits and Systems for Video Technology*, 12 (12): pp. 1093–1105, December 2002.

[36] Kristof Denolf. Method and device for block-based conditional motion compensation. US Patent number 0152149 A1, August 2003.

[37] Kristof Denolf, Marco Bekooij, Johan Cockx, Diederik Verkest, and Henk Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *accepted for EURASIP Journal on Applied Signal Processing, Special Issue on Transforming Signal Processing Applications into Parallel Implementations*, 2007.

[38] Kristof Denolf, Adrian Chirila-Rus, Paul Schumacher, Robert Turney, Kees Vissers, Diederik Verkest, and Henk Corporaal. A systematic approach to design low power video codec cores. *accepted for EURASIP Journal on Embedded Systems, Special Issue on Embedded Systems for Portable and Mobile Video Platforms*, 2007.

[39] Kristof Denolf, Adrian Chirila-Rus, Robert Turney, Paul Schumacher, and Kees Vissers. Memory efficient design of an MPEG-4 video encoder for FPGAs. In: *International Conference on Filed Programmable Logic and Applications, FPL*, pp. 391–396, August 2005.

[40] Kristof Denolf, Adrian Chirila-Rus, and Diederik Verkest. Low-power MPEG-4 video encoder design. In: *IEEE Workshop on Signal Processing Systems, SIPS*, pp. 284–289, November 2005.

[41] Kristof Denolf, Christophe De Vleeschouwer, Robert Turney, Gauthier Lafruit, and Jan Bormans. Memory centric design of an MPEG-4 video encoder. *IEEE Transactions on circuits and systems for video technology*, 15 (5): pp. 609–619, May 2005.

[42] Kristof Denolf, Kees Vissers, Paul Schumacher, Robert Turney, Nathan Jachimiec, Adrian Chirila-Rus, Bart Vanhoof, and Jan Bormans. A systematic design of an MPEG-4 video encoder and decoder for FPGAs. In: *Global Signal Processing Conferences and Expos for Industry, GSPx*, September 2004.

[43] Adam Donlin. Transaction level modeling: flows and use models. In: *International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS*, pp. 75–80, September 2004.

[44] J. Dunlop, A. Simpson, S. Masud, M. Wylie, J. Cochrane, and R. Kinkead. Semiconductor IP core for ultra low power MPEG-4 video decode in system-on-silicon. In: *International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, pp. 681–684, April 2003.

[45] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85 (3): pp. 366–390, March 1997.

[46] Toshihide Fujiyoshi, Shinichiro Shiratake, Shuou Nomura, Tsuyoshi Nishikawa, Yoshiyuki Kitasho, Hideho Arakida, Yuji Okuda, Yoshiro Tsuboi, Mototsugu Hamada, Hiroyuki Hara, Tetsuya Fujita, Fumitoshi Hatori, Takayoshi Shimazawa, Kunihiko Yahagi, Hideki Takeda, Masami Murakata, Fumihiro Minami, Naoyuki Kawabe, Takeshi Kitahara, Katsuhiro Seta, Masafumi Takahashi, Yukihito Oowaki, and Tohru Furuyama. A 63-mW H.264/MPEG-4 audio/visual codec LSI with module-wise dynamic voltage/frequency scaling. *IEEE Journal of Solid-State Circuits*, 41 (1): pp. 54–62, January 2006.

[47] D. Gajski and R. Kuhn. New VLSI tools. *IEEE Computer*, 16 (12): pp. 11–14, December 1983.

[48] Andreas Gerstlauer and Daniel D. Gajski. System-level abstraction semantics. In: *International Symposium on System Synthesis, ISSS*, pp. 231–236, October 2002.

[49] A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. In: *Formal Methods in Computer Aided Design*, pp. 68–75, November 2006.

[50] Steve Goddard and Kevin Jeffay. Managing latency and buffer requirements in processing graph chains. *The Computer Journal*, 44 (6): pp. 486–503, 2001.

[51] Jeremiah Golston and Ajit Rao. Video codecs tutorial: trade-offs with H.264, VC-1 and other advanced codecs. EEtimes Online, http://www.eetimes.com/showArticle.jhtml?articleID=184417335, March 2006.

[52] R. Govindarajan, Guang R. Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31 (3), August 2002.

[53] Robert Grou-Szabo, Hany Ghattas, Yvon Savaria, and Gabriela Nicolescu. Component-based methodology for hardware design of a dataflow processing network. In: *International Workshop on System-on-Chip for Real-Time Applications, IWSOC*, pp. 289–294, July 2005.

[54] Fiorella Haim, Mainak Sen, Dong-Ik Ko, Shuvra S. Bhattacharyya, and Wayne Wolf. Mapping multimedia applications onto configurable hardware with parameterized cyclo-static dataflow graphs. In: *International Conference on Acoustics, Speech, and Signal Processing, ICASSP*, volume 3, pp. 1052–1055, May 2006.

[55] T. Hashimoto, M. Ohashi, M. Matsuo, S. Kuromaru, T. Mori-iwa, M. Hamada, Y. Sugisawa, H. Tomita, M. Hoshino, T. Nakamura, K. Ishida, K. Watada, T. Fukunaga, and J. Michiyama. A 27-MHz/54-MHz 11-mW MPEG-4 video decoder LSI for mobile applications. *IEEE Journal of Solid-State Circuits*, 37 (11): pp. 1574–1581, November 2002.

[56] Atsushi Hatabu, Takashi Miyazaki, and Ichiro Kuroda. QVGA/CIF resoluation MPEG-4 video codec based on a low-power and general-purpose DSP. In: *IEEE Workshop on Signal Processing Systems, SIPS*, pp. 15–20, October 2002.

[57] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantative Approach*. Morgan Kauffmann, 2003.

[58] Mark Horowitz, Elad Alon, Dinesh Patil, Samuel Naffziger, Rajesh Kumar, and Kerry Bernstein. Scaling, power and the future of CMOS. In: *IEEE International Electron Devices Meeting Technical Digest, IEDM*, December 2005.

[59] 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.*

[60] Jens Horstmannshoff and Heinrich Meyr. Efficient building block based RTL code generation from synchronous data flow graphs. In: *Conference on Design Automation, DAC*, pp. 552–555, 2000.

[61] Hyeyoung Hwang, Taewook Oh, Hyunuk Jung, and Soonhoi Ha. Conversion of reference C code to dataflow model: H.264 encoder case study. In: *Conference on Asia South Pacific Design Automation, ASP-DAC*, pp. 152–157, January 2006.

[62] Mary Jane Irwin, Mahmut T. Kandemir, Narayanan Vijaykrishnan, and Anand Sivasubramaniam. A holistic approach to system level energy optimization. In: *International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation, PATMOS*, pp. 88–107. Springer-Verlag, September 2000.

[63] Hyunuk Jung, Kangnyoung Lee, and Soonhoi Ha. Efficient hardware controller synthesis for synchronous dataflow graph in system level design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10 (4): pp. 423–428, August 2002.

[64] Jeffrey Kang, Albert van der Werf, and Paul Lippens. Mapping array communication onto fifo communication - towards an implementation. In: *International Symposium on System Synthesis*, pp. 207–213, September 2000.

[65] Kurt Keutzer, Sharad Malik, A. Richard Newton, Jan M. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19 (12): pp. 1523–1543, December 2000.

[66] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann. The MPEG-4 video coding standard: a VLSI point of view. In: *IEEE Workshop on Signal Processing Systems, SIPS*, pp. 43–52, 1998.

[67] Peter M. Kuhn and Walter Stechele. Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation. In: *International Society Optical Engineering, SPIE*, volume 3309, pp. 498–509, January 1998.

[68] Andy Lambrechts, Praveen Raghavan, Anthony Leroy, Guillermo Talavera, Tom Vander Aa, Murali Jayapala, Francky Catthoor, Diederik Verkest, Geert Deconinck2, Henk Corporaal, Frederic Robert, and Jordi

Carrabina. Power breakdown analysis for a heterogeneous NoC platform running a video application. In: *International Conference on Application-specific Systems, Architectures and Processors, ASAP*, pp. 179–184, July 2005.

[69] Edward A. Lee. Embedded software. *Advances in Computers, Elsevier*, 56: pp. 56–97, 2002.

[70] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36 (1): pp. 24–35, January 1987.

[71] Weiping Li, Jens-Rainer Ohm, Mihaela van der Schaar, Hong Jiang, and Shipeng Li. *MPEG-4 Video Verification Model version 17.0*. ISO/IEC JTC1/SC29/WG11 N3515, July 2000.

[72] Chung-Jr Lian, Yu-Wen Huang, Hung-Chi Fang, Yung-Chi Chang, and Liang-Gee Chen. JPEG, MPEG-4, and H.264 codec IP development. In: *Design, Automation and Test in Europe, DATE*, volume 2, pp. 1118–1119, 2005.

[73] Chia-Ping Lin, Po-Chih Tseng, Yao-Ting Chiu, Siou-Shen Lin, Chih-Chi Cheng, Hung-Chi Fang, Wei-Min Chao, and Liang-Gee Chen. 5mW MPEG-4 SP encoder with 2D bandwidth sharing motion estimation for mobile applications. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC*, volume 1, pp. 412, 662, February 2006.

[74] Radu Marculescu, Umit Y. Ogras, and Nicholas H. Zamora. Computation and communication refinement for multiprocessor SoC design: a system-level perspective. *ACM Transactions on Design Automation of Electronic Systems*, 11 (3): pp. 564–592, July 2006.

[75] Luca Mazzoni. Power-aware design for embedded systems. *Electronics Systems and Software*, 1 (5): pp. 12–17, October-November 2003.

[76] Praveen K. Murthy and Shuvra S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20 (2): pp. 177–198, February 2001.

[77] Lode Nachtergaele, Francky Catthoor, Bhanu Kapoor, S. Janssens, and Dennis Moolenaar Dennis. Low power data transfer and storage exploration for H.263 video decoder system. *IEEE Journal on Selected areas in Communications, Special issue on very low bit-rate video coding*, 16 (1): pp. 120–129, January 1998.

[78] Lode Nachtergaele, Toon Gijbels, Jan Bormans, Francky Catthoor, and Ivo Bolsens. Power and speed-efficient code transformation of video compression algorithms for RISC processors. *Journal of VLSI Signal Processing, Special issue on multimedia signal processing*, 27 (1-2): pp. 161–169, February 2001.

[79] Lode Nachtergaele, Dennis Moolenaar, Bart Vanhoof, Francky Catthoor, and Hugo De Man. System-level power optimization of video codecs on embedded cores: a systematic approach. *Journal of VLSI Signal Processing, Special issue on future directions in the design and implementations of DSP systems*, 18 (2): pp. 89–109, February 1998.

[80] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, and Ko Yoshikawa. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In: *Conference on Design Automation, DAC*, pp. 299–304, June 2004.

[81] H. Nakayama, T. Yoshitake, H. Komazaki, Y. Watanabe, H. Araki, K. Morioka, J. Li, L. Peilin, S. Lee, H. Kubosawa, and Y Otobe. An MPEG-4 video LSI with an error-resilient codec core based on fast motion estimation algorithm. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC*, volume 1, pp. 368,474, February 2002.

[82] M. Ohashi, T. Hashimoto, S.I. Kuromaru, M. Matsuo, T. Mori-iwa, M. Hamada, Y. Sugisawa, M. Arita, H. Tomita, M. Hoshino, H. Miyajima, T. Nakamura, K.I. Ishida, T. Kimura, Y. Kohashi, T. Kondo, A. Inoue, H. Fujimoto, K. Watada, T. Fukunaga, T. Nishi, H. Ito, and J. Michiyama. A 27 MHz 11.1 mW MPEG-4 video decoder LSI for mobile application. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC*, volume 1, pp. 366, 474, February 2002.

[83] Hyunok Ok and Soonhoi Ha. Memory-optimized software synthesis from dataflow program graphs with large data samples. *EURASIP Journal on Applied Signal Processing*, 2003 (6): pp. 514–529, June 2003.

[84] Jorn Ostermann, Jan Bormans, Peter List, Detlev Marpe, Matthias Narroschke, Fernando Pereira, Thomas Stockhammer, and Thomas Wedi. Video coding with H.264/AVC: tools, performance and complexity. *IEEE Circuits and Systems Magazine*, 4 (1): pp. 7–28, 2004.

[85] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory

optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6 (2): pp. 149–206, April 2001.

[86] Ming Pao and Ming-Ting Sun. Modelling dct coefficients for fast video encoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 9 (4): pp. 608–616, June 1999.

[87] Chanik Park and Soonhoi Ha. Hardware synthesis from SPDF representation for multimedia applications. In: *International Symposium on System Synthesis, ISSS*, pp. 215–220, September 2000.

[88] Chanik Park, Jaewoong Jung, and Soonhoi Ha. Extended synchronous dataflow for efficient dsp system prototyping. *Design Automation for Embedded Systems, Kluwer Academic Publishers*, 6 (3): pp. 295–322, March 2002.

[89] Thomas M. Parks, Jose Luis Pino, and Edward A. Lee. A comparison of synchronous and cycle-static dataflow. In: *Asilomar Conference on Signals, Systems and Computers*, p. 204, October 1995.

[90] Alessandro Pinto, Alvise Bonivento, and Alberto L. Sangiovanii-Vincentelli. System level design paradigms: platform-based design and communication synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 11 (3): pp. 537–563, July 2006.

[91] Peter Pirsch, Mladen Berekovic, Hans-Joachim Stolberg, and Jrn Jachalsky. VLSI architectures for MPEG-4. In: *International Symposium on VLSI Technology, Systems, and Applications*, pp. 208A–208E, October 2003.

[92] Peter Poplavko, Twan Basten, Marco Bekooij, Jef van Meerbergen, and Bart Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES*, pp. 63–72, October 2003.

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

[94] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia.* Wiley, 2003.

[95] Tero Rintaluoma, Olli Silven, and Juuso Raekallio. Interface overheads in embedded multimedia software. In: *International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS*, pp. 5–14, July 2006.

[96] Sergio Saponara, Kristof Denolf, Gauthier Lafruit, Carolina Blanch, and Jand Bormans. Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications. *EURASIP Journal on Applied Signal Processing*, 2004 (2): pp. 220–235, February 2004.

[97] Paul Schumacher, Kristof Denolf, Adrian Chirila-Rus, Robert Turney, Nick Fedele, Kees Vissers, and Jan Bormans. A scalable, multi-stream MPEG-4 video decoder for conferencing and surveillance applications. In: *International Conference on Image Processing, ICIP*, volume 2, pp. 886–889, September 2005.

[98] T. Sikora. Trends and perspectives in image and video coding. *Proceedings of the IEEE*, 93 (1): pp. 6–17, January 2005.

[99] Rawat Siripokarpirom and Friedrich Mayer-Lindenberg. Hardware-assisted simulation and evaluation of IP cores using FPGA-based rapid prototyping boards. In: *IEEE International Workshop on Rapid System Prototyping*, pp. 96–102, June 2004.

[100] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marchel Dekker, 2000.

[101] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using Kahn process networks: the compaan/laura approach. In: *Design Automation and Test in Europe Conference and Exhibition, DATE*, pp. 340–345, 2004.

[102] Sander Stuijk, Marc Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In: *Conference on Design Automation, DAC*, pp. 899–904, July 2006.

[103] Jrgen Teich and Shuvra S. Bhattacharyya. Analysis of dataflow programs with interval-limited data-rates. *Journal of VLSI Signal Processing*, 43 (2-3): pp. 247–258, June 2006.

[104] Jen-Chieh Tuan, Tian-Sheuan Chang, and Chein-Wei Jen. On the data reuse and memory bandwidth analysis for full-search block-matching VLSI architecture. *IEEE Transactions on Circuits and Systems for Video Technology*, 12 (1): pp. 61–72, January 2002.

[105] Hugo De Man. *Ontwerp van Micro-Elektronische Systemen*. Katholieke Universiteit Leuven, 2002.

[106] Kimiyoshi Usami, Mutsunori Igarashi, Takashi Ishikawa, Masahiro Kanazawa, Masafumi Takahashi, Mototsugu Hamada, Hideho Arakida, Toshihiro Terazawa, and Tadahiro Kuroda. Design methodology of ultra low-power MPEG-4 codec core exploiting voltage scaling techniques. In: *Conference on Design Automation, DAC*, pp. 483–488, June 1998.

[107] Bart Vanhoof, Mercedes Peon, Gauthier Lafruit, Jan Bormans, Marc Engels, and Ivo Bolsens. A scalable architecture for MPEG-4 zero tree coding. In: *Custom Integrated Circuits Conference*, pp. 65–68, May 1999.

[108] Marc A. Viredaz and Deborah A. Wallach. Power evaluation of a handheld computer. *IEEE Micro*, 23 (1): pp. 66–74, January/February 2003.

[109] Yasuhiro Watanabe, Toshiyuki Yoshitake, Kiyonori Morioka, Taro Hagiya, Hiroki Kobayashi, Hyuk-Jae Jang, Hiroshi Nakayama, Yukio Otobe, and Akihiro Higashi. Low power MPEG-4 ASP codec IP macro. In: *International Conference on Consumer Electronics Digest of Technical Papers, ICCE*, pp. 337–338, January 2005.

[110] Maarten Wiggers, Marco Bekooij, Pierre Jansen, and Gerard Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In: *International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS*, pp. 10–15, October 2006.

[111] Maarten Wiggers, Marco Bekooij, Pierre Jansen, and Gerard Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In: *To appear in IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, April 2007. Accepted for publication.

[112] Michael C. Williamson and Edward A. Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In: *Asilomar Conference on Signals, Systems and Computers*, volume 2, pp. 1340–1343, November 1996.

[113] T. Yamada, N. Irie, J. Nishimoto, Y. Kondoh, T. Nakazawa, K. Yamada, K. Tatezawa, T. Irita, S. Tamaki, H. Yagi, M. Furuyama, K. Ogura, H. Watanabe, R. Satomura, K. Hirose, F. Arakawa, T. Hattori, I. Kudo, I. Kawasaki, and K Uchiyama. A 133 MHz 170 mW 10 mA standby application processor for 3G cellular phones. In: *IEEE International*

Solid-State Circuits Conference Digest of Technical Papers, ISSCC, volume 1, pp. 370,474, 2002.

[114] Hideki Yamauchi, Shigeyuki Okada, Tsuyoshi Watanabe, Yoshihiro Matsuo, Mitsuru Suzuki, Yasuo Ishii, Tsugio Mori, and Yoshifumi Matsushita. An 81MHz, 1280 x 720pixels x 30frames/s MPEG-4 video/audio codec processor. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers, ISSCC*, volume 1, pp. 130,589, February 2005.

[115] Yafan Zhao and Iain E.G. Richardson. Macroblock skip-mode prediction for complexity control of video encoders. In: *International Conference on Visual Information Engineering, VIE*, pp. 5–8, July 2003.

List of Publications

Journal Papers

1. Kristof Denolf, Marco Bekooij, Johan Cockx, Diederik Verkest, and Henk Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *accepted for EURASIP Journal on Applied Signal Processing, Special Issue on Transforming Signal Processing Applications into Parallel Implementations*, 2007.

2. Johan Cockx, Kristof Denolf, Bart Vanhoof, and Richard Stahl. SPRINT: a tool to generate concurrent transaction level models from sequential code. *EURASIP Journal on Advances in Signal Processing, Special Issue on Transforming Signal Processing Applications into Parallel Implementations*, 2007: pp. Article ID 75373, 15 pages, 2007.

3. Kristof Denolf, Adrian Chirila-Rus, Paul Schumacher, Robert Turney, Kees Vissers, Diederik Verkest, and Henk Corporaal. A systematic approach to design low power video codec cores. *accepted for EURASIP Journal on Embedded Systems, Special Issue on Embedded Systems for Portable and Mobile Video Platforms*, 2007.

4. Kristof Denolf, Christophe De Vleeschouwer, Robert Turney, Gauthier Lafruit, and Jan Bormans. Memory centric design of an MPEG-4 video encoder. *IEEE Transactions on circuits and systems for video technology*, 15 (5): pp. 609–619, May 2005.

5. Sergio Saponara, Kristof Denolf, Gauthier Lafruit, Carolina Blanch, and Jand Bormans. Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications. *EURASIP Journal on Applied Signal Processing*, 2004 (2): pp. 220–235, February 2004.

6. Christophe De Vleeschouwer, Tord Nilsson, Kristof Denolf, and Jan Bormans. Algorithmic and architectural co-design of a motion-estimation engine for low-power video devices. *IEEE Transactions on Circuits and Systems for Video Technology*, 12 (12): pp. 1093–1105, December 2002.

Conference Papers

1. Kristof Denolf, Adrian Chirila-Rus, and Diederik Verkest. Low-power MPEG-4 video encoder design. In: *IEEE Workshop on Signal Processing Systems, SIPS*, pp. 284–289, November 2005.

2. Paul Schumacher, Kristof Denolf, Adrian Chirila-Rus, Robert Turney, Nick Fedele, Kees Vissers, and Jan Bormans. A scalable, multi-stream MPEG-4 video decoder for conferencing and surveillance applications. In: *International Conference on Image Processing, ICIP*, volume 2, pp. 886–889, September 2005.

3. Kristof Denolf, Adrian Chirila-Rus, Robert Turney, Paul Schumacher, and Kees Vissers. Memory efficient design of an MPEG-4 video encoder for FPGAs. In: *International Conference on Filed Programmable Logic and Applications, FPL*, pp. 391–396, August 2005.

4. Adrian Chirila-Rus, Kristof Denolf, Bart Vanhoof, Paul Schumacher, and Kees Vissers. Communication primitives driven hardware design and test methodology applied on complex video applications. In: *Rapid System Prototyping workshop, RSP*, pp. 246–248, June 2005.

5. Kristof Denolf, Kees Vissers, Paul Schumacher, Robert Turney, Nathan Jachimiec, Adrian Chirila-Rus, Bart Vanhoof, and Jan Bormans. A systematic design of an MPEG-4 video encoder and decoder for FPGAs. In: *Global Signal Processing Conferences and Expos for Industry, GSPx*, September 2004.

Patent Application

Kristof Denolf. Method and device for block-based conditional motion compensation. US Patent number 0152149 A1, August 2003.

Curriculum Vitae

Kristof Denolf was born in Kortrijk, Belgium, on December 30, 1975. He received the M.Eng. degree in electronics from the Katholieke Hogeschool, Brugge-Oostende, Belgium, in 1998, and the M.Sc. degree in electronic system design from Leeds Metropolitan University, Leeds, U.K., in 2000. He joined the Nomadic Embedded Systems division, at the Interuniversity Micro Electronics Centre (IMEC), Leuven, Belgium, in August 1998, where he is currently a senior research engineer in the MultiMedia group. From 2002 to 2007 he was also working towards the PhD degree with the department of Electrical Engineering at the Technische Universiteit Eindhoven, The Netherlands. His main research interests are the cost efficient design of advanced video processing systems and the end-to-end quality of experience.

The work described in this thesis
was carried out at IMEC