Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Research Project
at the Department of Information Technology
and Electrical Engineering

Towards Exploiting Intra-Application Dynamism using an H.264 Codec

Georgios Kathareios
MSc. Student, Delft University of Technology

Advisors:	Lars Schor
	Dr. Hoeseok Yang
	Dr. Iuliana Bacivarov
Professor:	Prof. Dr. Lothar Thiele
Supervisor at TUDelft:	Prof. Dr. Koen Bertels

Zurich
16th November 2013

Abstract

The computational requirements of applications targeting multi- and many-core systems increase rapidly. In order to achieve optimized execution, streaming applications must dynamically adapt to changes of their input. In this work, this intra-application dynamism is explored using an H.264 standard complying Codec pair.

Initially, an encoder/decoder pair of applications is implemented for the DAL model of computation, using code targeting the HOPES framework as baseline. Focus is given on the encoder application, where we apply the wavefront parallelism technique in order to increase the degree of parallelism. This degree is defined as a function of the input video's resolution, which introduces static dynamism to the application.

As a means of increasing the encoder's performance, a frame pipelining scheme is proposed. In addition, we argue that run-time dynamism can be achieved by balancing the tradeoff between execution time and compression rate. Three encoding parameters are proposed that can be adjusted at run-time to achieve run-time dynamism.

Finally, the implementation is evaluated, by calculating the maximum theoretical speedup that it can achieve and by exploring the effects of different mappings and different degrees of parallelism on the performance. The results show that we have achieved better maximum speedup than the HOPES implementation.

Acknowledgements

I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for giving me the opportunity to work on this project in his research group in ETHZ.

I would also like to deeply thank Prof. Dr. Koen Bertels for agreeing to be my supervisor in TUDelft.

Finally, this project would not have been completed without the help and support of my advisors Lars Schor, Dr. Hoeseok Yang and Dr. Iuliana Bacivarov. Their guidance, help and ideas during our discussions proved critical for solving the problems of this work.

Contents

List of Figures

1 Introduction

1.1 Motivation

In recent years, we have witnessed a paradigm shift from single-core to multi-core systems, a change that was inevitable since single-core systems hit the so-called "power wall". Almost free performance was achieved by increasing the clock frequency of such systems, at the cost of increasing power consumption to prohibitive levels. At the same time, advances in VLSI technology manage to constantly increase the number of transistors per chip, paving the way for multiple cores per chip. Thus, in the field of embedded systems, Multi-Processor Systems-On-Chip (MPSoC) became the norm, as they offer high computing capabilities and low power consumption. However, these gains come at the cost of more complex programming models, which raise a number of new challenges. One particular challenge is the optimized utilization of the cores for the execution of an application.

Furthermore, the behaviour and performance of modern applications highly depend on their input. Hence, in order to achieve the most efficient utilization of the hardware, applications must dynamically adapt and react to changes in their input. A prominent example of this kind of applications is a video encoder/decoder pair. Usually, the encoding/decoding process needs to meet throughput and latency constraints, which need to be ensured for every input video. As such, the system resources that the application utilizes must be dynamically adjusted in order to prevent over- or under-utilization of the underlying platform.

This research project explores this intra-application dynamism on an H.264 compliant encoder implemented using the DAL framework.

1.2 Contributions

- A H.264 complying encoder/decoder pair of applications is implemented using the DAL framework. These applications are used as the base for the rest of the project, but can also be used as benchmarks for the framework.
- We discuss the wavefront parallelism inherent to the Motion Estimation step of the encoder and present a way to calculate the maximum degree of parallelism that can be achieved as a function of the input video's resolution. The implemented encoder is modified to exploit this extra parallelism.
- Finally, we evaluate the resulting application and propose possible enhancements of the implementation, aiming in increasing its performance and achieving run-time dynamism.

1.3 DAL Framework

The *Distributed Application Layer* (DAL) [15] is a scenario-based design flow and software development framework to specify applications as platform-independent Kahn Process Networks (KPN) [3], map, and execute them on heterogeneous many-core platforms. It was developed by the Computer Engineering and Networking Laboratory (TIK) of the Swiss Federal Institute of Technology (ETHZ) as a contribution to the EU-FP7 project EURETILE [12].

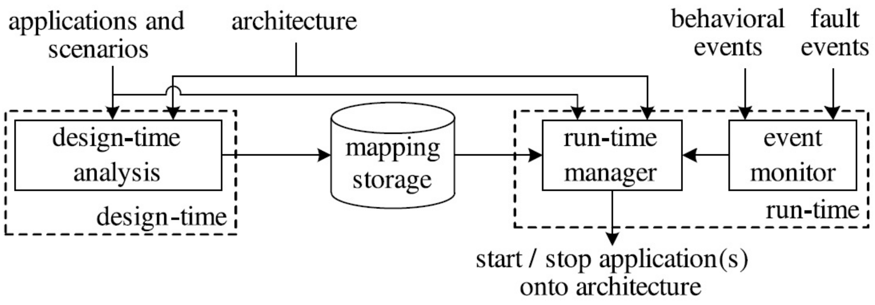


Figure 1.1: Stucture of the design flow of DAL [15].

DAL is designed for use in embedded systems, where typically a given set of applications shares the system, with a fraction of them running at a given time. The design flow it encompasses is shown in Fig.1.1. The behaviour of the system is described as a predefined set of *scenarios*, which is restricted in size and known at design-time. A scenario is defined as a system state at which a subset of the system's applications' set is running or paused. The transitions between scenarios effectively enable behavioural inter-application dynamism. They are described by a Finite State Machine (FSM) that represents the scenarios as states. These transitions are triggered by *events* generated by the applications or the run-time system.

The applications that run on the system are described by means of KPNs. Each individual application is composed of a set of autonomous processes, which interact by exchanging data through FIFO channels. The KPN processes are specified using a high-level API that includes the procedures INIT, FIRE, and FINISH for each one of them. The first procedure is called once for initialization whenever the application starts, and the last one is called to cleanup after the application is finished. The FIRE procedure is called repeatedly for the duration of the application by the system's scheduler, performing the main purpose of the process. Inter-process communication is achieved by the procedures *DAL_read* and *DAL_write* at any part of the process. In addition, events can be generated by any process with the *send_event* procedure.

Apart from the software, the design flow involves the description of the underlying system's architecture. The architecture model describes the platform in an abstract manner, using a hierarchical representation that separates intra-core, intra-tile and inter-tile communication. At the same time, spare cores are kept as backup at design-time to ensure transparent fault recovery at run-time, enabling architectural dynamism.

The final component of the design flow is the *mapping* of software elements (processes) to hardware elements (cores). The mapping process on DAL is performed on design-time. For each scenario an optimal mapping is generated, forming a set of mapping that the run-time manager can access and apply to start, stop, pause, and resume an application. In order to avoid reconfiguration overhead and process migration costs, the applications are assumed resident, i.e., they have the same mapping in two connected scenarios.

Extra features of the DAL framework include *Design-Time Analysis and Mapping Optimization* [4], and *Expendable Process Networks* (EPN) [16]. Design-Time Analysis and Mapping Optimization can be performed when the performance analyses of the system's applications are known. Also given the architectural model of the system as input, an evolutionary algorithm computes the set of optimal mappings that fulfil a set of predefined constraints. EPNs are a model of computation that can dynamically increase the degree of parallelism of KPNs or other streaming programming models, by abstracting several possible granularities in a single specification. Using methods such as process replication and unfolding, the process network of an application can be transformed to an equivalent network with different granularity.

The DAL framework will be used in this work for producing and running the H.264 encoder and decoder implementations.

1.4 Related Work

The H.264 standard is often ported and evaluated for a wide variety of platforms in related research work, due to its widespread use, high computational power demands and heavy data dependencies that make parallelization a challenge. Implementations of a standard-complying encoder on generic multi-processors, the Cell Processor and GPUs exist. In this section, we present some examples. For more details on technical terms, or information on wavefront parallelism, refer to Chapter 2 and Section 3.4, respectively.

Rodriguez et al. [14] consider parallelizing on the GOP[1] level and the slice level. However, multiple GOPs may not be available immediately, as in the case of streaming videos, or two-ways video communication. In addition, it is often the case that one slice per frame is used to achieve the best compression rate, and as such these techniques can not be employed in the general case.

On the other hand, Zrida et al. [18] use task parallelism and describe the encoder by means of a fine grained KPN. The granularity is then coarsened, according to their workload analysis, to a KPN resembling the one used in this work. Subsequently, they identify the macroblock dependencies and further parallelize the Motion Estimation task on the macroblock level, by dividing the frame in three columns. Although the number of columns could potentially change by spawning more identical tasks, this division does not give the maximum possible degree of parallelism as in other implementations.

Ko et al. [5] propose their own Motion Estimation algorithm which is suitable for GPU implementations, and combine it with frame-level parallelism. On the same context, Wu et al. [9] parallelize all steps of the encoding process for GPU, by taking advantage of the high degree of parallelism on such platforms, e.g., in inter-prediction, they omit motion vector prediction and therefore all dependencies from the current frame. In intra-prediction, a form of wavefront parallelism is used, as described later on in this work. In

[1]Group of Pictures: A group of consecutive frames starting with an I frame, that is encoded independently from any other frame.

this particular form, in order to further increase the degree of parallelism, two of the available prediction modes are omitted to reduce the number of dependencies for each macroblock, which can result in a lower compression rate. However innovative and successful, GPU implementations cannot be ported directly on MPSoCs, since these systems lack on raw computational power and fine granularity.

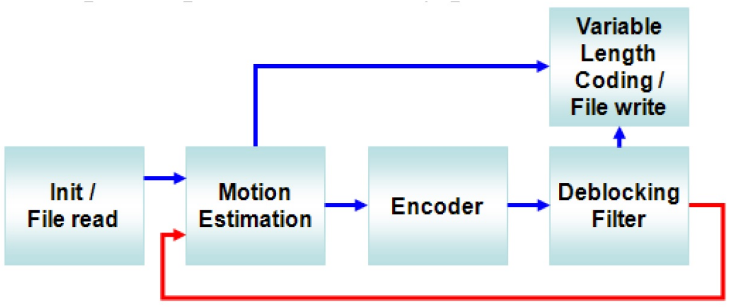


Figure 1.2: Flow chart of the H.264 encoder [7].

The work most closely related to this project is the research done in the Seoul National University [7, 11]. The HOPES framework [6] is employed to implement an H.264 encoder that takes advantage of task parallelism. This is similar to DAL, in the sense that the applications are described in a platform-independent format, called the Common Intermediate Code (CIC) format, resembling the KPN description of DAL. The architecture and the mapping of the target platform are also provided for the generation of the final executable program. A major difference between the two frameworks is that HOPES does not provide inter-application dynamism, as only a single mapping is provided. The initial implementation of the encoder in this work is based on the popular x264 open source encoder [1], which consists of the five tasks shown in Fig. 1.2. In order to improve this implementation, the Motion Estimation task is substituted by a wrapper task, including several instances of the Motion Estimation task. This scheme increases the degree of parallelism and takes advantage of the wavefront parallelism. The results show that 56% performance improvement is achieved by using 2 Motion Estimation tasks. The code of the initial implementation will be the basis for the DAL implementation in this work.

1.5 Outline

The rest of this project report is organized in three chapters.

Chapter 2 provides a brief introduction to the H.264 standard. We first provide an overview of the basic functionalities it includes, and describe some basic terminology that is used in the rest of the work. Furthermore, we show the structures of a typical standard-complying encoder and decoder, and describe details of the baseline profile that will be the target of this project.

In Chapter 3 the encoder/decoder pair's implementation is discussed. First, we show how each of the initial implementations is achieved and discuss modifications that had to be done in the provided HOPES implementations. Afterwards, we extend and improve the encoder, by further parallelizing the Motion Estimation process using the wavefront parallelism present in the application, thus enabling static dynamism. In the end of the chapter, we propose an enhancement for improving the performance of the final implementation.

Chapter 4 discusses methods that can utilized for achieving run-time intra-application dynamism on the application. In particular, three methods are discussed, which focus on balancing the tradeoff between the performance of the encoder and the compression rate of the output.

Chapter 5 shows our experimental results of the improved encoder and the effect that design parameters have on the performance.

2 The H.264 Standard

The H.264/MPEG-4 AVC (Advanced Video Codec) is one of the most widely used video coding standards in recent years, which is mainly attributed to its significantly increased achievable bitrate compression compared to its predecessors. However, the increased performance comes at the cost of increased complexity and a lengthy specification.

This chapter aims at describing the basic concepts of the standard and providing a very brief summary of the specification that are needed in order to present our work [1]. Section 2.1 provides an overview of the standard, while Section 2.2 covers the basic terminology. Furthermore, the structures of both the encoder and the decoder are presented in Section 2.3, and the baseline profile that was implemented is described in Section 2.4.

2.1 Overview

The H.264 standard specification [2] was developed by the ISO/IEC Moving Picture Experts Group (MPEG) and the ITU-T Video Coding Experts Group (VCEG), a partnership known as Joint Video Team (JVT). The first version of the standard was completed in 2003 under the formal name *ISO/IEC 14496–10 – MPEG 4 Part 10, Advanced Video Coding*. It was motivated by emerging networking technologies such as xDSL and UMTS,

[1]For a more detailed description, the reader is referred to [13], [17], and of course the standard's specification [2].

and was designed to outperform previous video coding standards, such as the basic MPEG-4 (part 2 of the MPEG4 standards) and H263, providing better compression rates and fault-tolerance. The standard reportedly achieves up to 50% better compression in various bitrates and resolutions, although its decoder is twice as complex as the basic MPEG-4 and its encoder can be as much as 10 times more complex [10].

The standard provides flexibility and is applicable in a variety of cases, ranging from two-way video communication to high quality video streaming over packet networks. This flexibility is achieved by the different profiles that are described in the specification. These include:

- The Baseline profile
- The Main profile
- The Extended profile

Each profile supports a particular set of coding functions and targets different applications, but is not limited by them. Their functionalities are summarized in Fig. 2.1. In this work, we focus on the baseline profile which has the lowest complexity and is more suited for embedded systems.

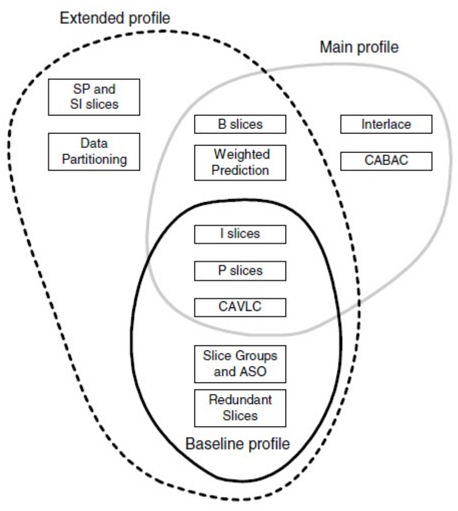


Figure 2.1: The functions of the H.264 profiles [13].

2.2 Terminology and Basic Concepts

The standard is implemented via a pair of algorithms called *Codec* (enCOder-/DECoder). The *encoder* takes a video sequence as input and processes it to produce the compressed *bitstream*, which can be transmitted via some communication medium or stored. The *decoder* reverses the compression and reconstructs the video sequence for playback.

A video consists of a sequence of *frames* that when alternated at the correct rate produce the moving picture. The frame consists of spatio-temporal samples called picture elements or *pixels.* The H.264 standard supports rectangular frames of the *YUV colour space* (a.k.a. YCrCb) sampled with the *4:2:0* format. As such, each picture is described by its *luminance* and *Cr and Cb chroma* coefficients *(Y, U, V, respectively)*, with the last two being sampled at half the rate of the first, both in the horizontal and vertical directions.

A frame, in order to be encoded and to produce a *coded picture*, is divided into *macroblocks*, each containing 16x16 luma samples, 8x8 Cr chroma samples and 8x8 Cb chroma samples. A macroblock can possibly be further divided into sizes as small as 4x4 Y, 2x2 U and 2x2 V samples, if needed. A *slice* is a set of macroblocks in raster scan order that belong in the same picture and can be encoded independently to other slices of the same frame. For simplicity, we will only use one slice per frame. A slice can be characterized as an *I-* or *P-slice* in the baseline profile[2]. An I-slice contains only *I macroblocks*, that is, macroblocks that are encoded (decoded) by utilizing data from previously encoded (decoded) macroblocks within the same slice and as such is independent from any other slice. A P-slice on the other hand contains both I and P macroblocks. *P macroblocks* are macroblocks that are encoded (decoded) by utilizing data from macroblocks that belong to a previously encoded (decoded) frame (called the *reference frame*). The process that enables the coding based on a previous frame is called *Motion Estimation* and *Motion Compensation.*

The idea behind the Motion Estimation and Compensation process is the fact that video sequences depict moving objects. Thus, it is not needed to re-transmit the data needed to describe an object, when the transmission of the object's movement suffices. The Motion Estimation algorithm calculates for each macroblock of the currently processed frame, the most similar macroblock of a certain search area of the reference frame, called the *predictor* macroblock. The difference between the current macroblock and the predictor is called the *residual.* A *motion vector* is used to describe the spatial displacement between the two. The residual most likely contains less energy than the initial macroblock, hence requires less space to be stored (or transmitted) along with the motion vector (a process called Motion Compensation). A P macroblock is said to be *inter-predicted*, as it depends on macroblocks of another slice, while an I macroblock is *intra-predicted*, as its prediction comes from previously processed macroblocks of the same slice.

[2]Also B-, SI- or SP-slice in other profiles.

The Group of Pictures (GOP) is a set of successive frames, with every one of them independent of frames outside the group. Thus, GOPs are independent from one another, a desired attribute because faults inside one set are not propagated outside of it. The first frame of the group is an I-frame which is succeeded by P-frames that use the first one as a reference, thus ensuring the independence of the GOP.

Apart from the prediction, in order to utilize as less data as possible for each macroblock, the residuals are transformed by a *Hadamard* or *DCT*-like transformation and *quantized* (each value is divided by a constant *quantization parameter*). Additionally, the data are coded with some form of *Entropy encoding* such as a *Context-Adaptive Variable Length Coding (CAVLC)* scheme. The quantization processes is a *lossy* compression method, which means that data are lost during the process and cannot be reconstructed. It is more likely that the edges of the macroblocks suffer the biggest loss, resulting in visible blocks on the reconstructed video. In order to avoid this phenomenon, a *deblocking filter* is applied to every decoded macroblock. The final compressed bitstream is passed to the *Network Abstraction Layer* (NAL), which facilitates the use of the standard on a broad variety of systems.

2.3 H.264 Structure

In common with earlier video coding standards, the H.264 specification does not directly specify a Codec pair. On the contrary, it specifies in detail the syntax of a coded bitstream, the semantics of the syntax elements and their decoding process, by which the initial video sequence can be retrieved [17]. In practice however, the functional representations of an H.264 encoder and decoder pair most likely resemble the ones shown in Figures 2.2 and 2.3, respectively.

2.3.1 Encoder

The Encoder consists of two paths, a “forward” path that encodes the current macroblock, and a "backwards" path that reconstructs it for use as a future reference (Fig. 2.2). The reconstruction of the reference frames on the encoder is needed because, as stated before, the encoding process is a lossy form of compression. That being said, some data of the initial video signal will be lost during the encoding process, and as such a decoded macroblock will slightly differ from its initial counterpart. Thus, when the encoder bases its description of a macroblock on previous ones, the base must match exactly the one that the decoder will later utilize.

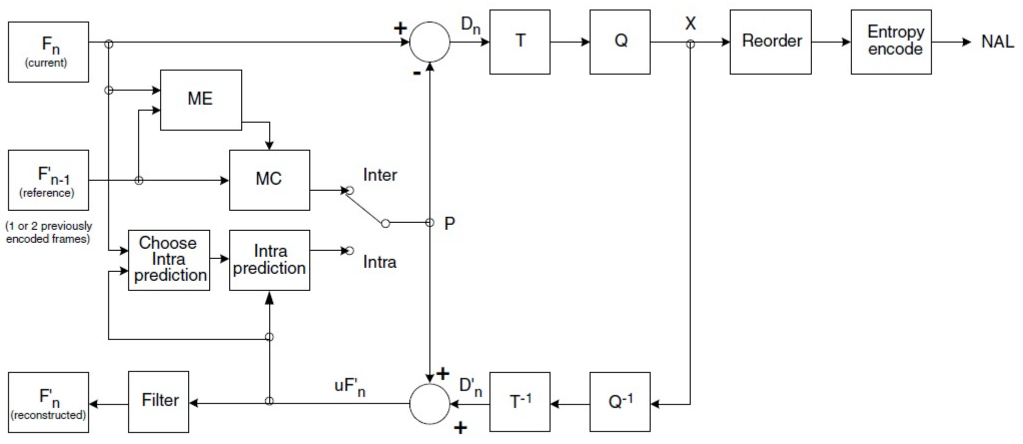


Figure 2.2: Functional specification of a H.264 encoder [13].

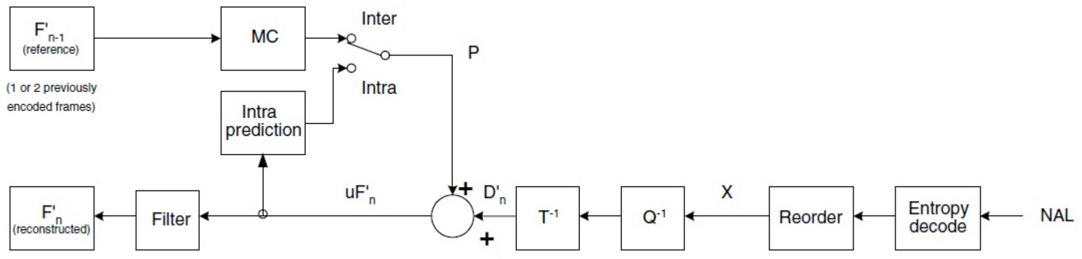


Figure 2.3: Functional specification of a H.264 decoder [13].

The forward path takes as input a frame that needs to be encoded (F_n), in units of a macroblock. Each macroblock is either inter- or intra- predicted, and the prediction P is produced. In the case of inter-prediction, the motion estimation algorithm uses the already reconstructed frame F'_{n-1} as a reference and produces a motion vector. In the case of intra-prediction, the reconstructed macroblocks of the same frame are used (uF'_n) as a base. Subsequently, the prediction is subtracted from the initial macroblock to produce the residual D_n, which is transformed (T) and quantized (Q) to produce X, the set of quantized transformed parameters. Afterwards, the elements of X are reordered to group non-zero coefficients and subjected to *Entropy encoding* to produce the final bitstream in the form of NAL units.

The reconstruction path rescales (Q^{-1}) the elements of X, and in turn the inverse transformation (T^{-1}) produces the reconstructed residual (D'_n). The residual combined with the prediction form the reconstructed frame, which is filtered with a Deblocking filter and stored as a reference. The filter is applied on the encoder as well because there is a need to have the references in exactly the same form as the decoder will, in order to produce the same prediction.

2.3.2 Decoder

The Decoder is the inverse process of the encoder (Fig. 2.3). It receives the compressed bitstream as input, in the form of NAL units and proceeds to decode the Entropy coding and inverse the reordering scan performed by the encoder. At this point, the set of quantized transformed parameters X is reconstructed and is treated in the same way as the reconstruction path of the encoder; it is rescaled (Q^{-1}) and inverse transformed (T^{-1}) to produce the residual (D'_n) which is identical to the residual D'_n of the encoder. In addition, the prediction (P) for the current macroblock is constructed from previously decoded macroblocks of either the same frame (intra-) or the reference frame (inter-prediction), according to guidelines provided by the header information. The prediction is added to the residual and the result is filtered in order to produce the reconstructed frame (F'_n).

2.4 The Baseline Profile

In this work, we focus on the Baseline Profile of the standard. As mentioned, this profile includes the most basic compression techniques and as such does not achieve the highest possible compression rate. However, due to its low complexity it is a perfect candidate for implementation on embedded systems and applications such as video-conferencing. In addition, it encompasses the core behaviour of the H.264 encoding/decoding process, therefore it can be perfectly utilized to investigate the behaviour and dynamism of a Codec pair. In this work, some details of this profile's implementation proved crucial and are thus explained in this section.

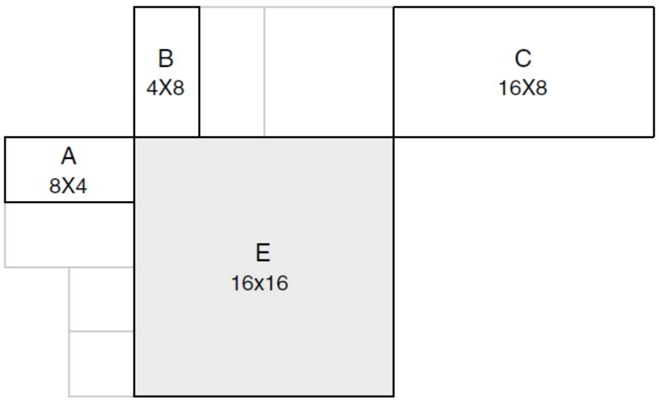


Figure 2.4: An example of dependencies for motion vector prediction [13].

In the H.264 standard, every macroblock is predicted from previously processed data and consequently heavy dependencies are introduced between them, dictating the order in which they are processed. In the baseline profile in particular, during Inter-prediction, a frame heavily depends on its reference frame. In addition, more dependencies are introduced by the need for reduced output. The motion vector produced from the Motion Estimation is also encoded to reduce the resulting bitstream, in a process called *Motion Vector Prediction*. This process predicts the motion vector of a block (macroblock or smaller size) by the motion vectors of its *left*, *top*, and *top-right* neighbours in the same frame. Fig. 2.4 shows an example of that: The macroblock E depends on its three neightbours, despite their partition size. These particular neighbours are already processed in a raster scan of the macroblocks, but nevertheless the introduced dependencies hinder a potential change of the scan order.

Intra-prediction has similar dependencies. The prediction can take place either in macroblock size or 4x4 samples sized luma partitions of macroblock (with the corresponding chroma blocks). There exist a number of intra-prediction modes for each size, each introducing its own dependencies. The largest number of dependencies are present on 4x4 luma blocks. In total, 9 modes exist for intra-predicting in this size, shown in Fig. 2.5. In this figure, the direction of each of the 9 numbered vectors shows the direction of the dependencies for each sample. We can see that for modes 3 and 7, the same dependencies as motion vector prediction are present (left, top, top-right), while in total the current block can depend on the same blocks and on its top-left one.

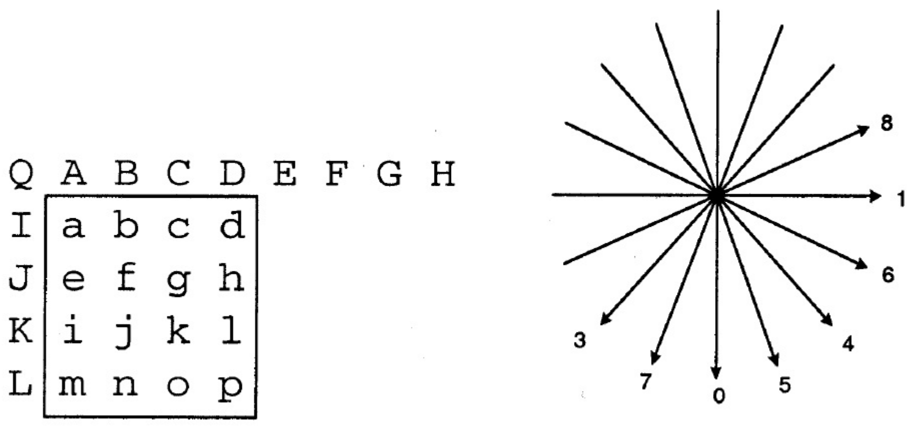


Figure 2.5: The dependencies for intra-prediction of a 4x4 block [17].

These dependencies will be revisited in the next chapter, as they are the main difficulty in parallelizing the H.264 encoder.

2.5 Summary

In this chapter, a brief introduction to the H.264 standard was presented. Some main concepts were discussed and the major functional blocks of a Codec pair are described. Finally, some aspects of the baseline profile are analysed that will prove crucial further on.

3 DAL Implementation of the Standard

As the first task of this project, an H.264 compliant Codec pair is implemented using the DAL framework. As mentioned earlier, the implementation is based on code for HOPES, provided by the Seoul National University and supports the baseline profile of the coding standard. Initially, intra-application dynamism is not considered, but a simple process network is created for each of the encoder and decoder applications. Then, focus is given on the encoder application for two reasons. Firstly, the dependencies between macroblocks are identical in the encoding and decoding process. Secondly, the encoder must be much more flexible in its execution, as it is called to make all decisions concerning the way a video is coded. As such, with the aim to both increase the degree of parallelism and allow a certain flexibility according to the input type, we explore the technique of wavefront parallelism on the encoder.

This chapter shows the methodology that was followed to achieve these goals. Section 3.1 presents the basic steps to convert an application from its CIC description in HOPES to its equivalent KPN description in DAL. Sections 3.2 and 3.3 describe how these steps are applied to create the initial implementations of the encoder and the decoder, respectively. Furthermore, Section 3.4 explores the wavefront parallelism that can be exploited on the encoder, and Section 3.5 proposes a method to increase the implementation's performance.

3.1 HOPES to DAL Transformation

As mentioned earlier, the HOPES and DAL frameworks bear similarities in the way that applications are described. However, in order to transform a CIC specification to a KPN one, emphasis must be put on certain differences.

```
static int somevar;

void process_init(){...}
int process_go(){
  ...
  somevar = ...;
  ...
  MQ_RECEIVE(port_id, buf, size);
  ...
  MQ_SEND(port_id, buf, size);
  ...
}
void process_wrapup(){...}
```

```
typedef struct _local_states {
  int somevar;
} process_State;

void process_init(DAL_process *p){...}
int process_fire(DAL_process *p){
  ...
  p->local->somevar = ...;
  ...
  DAL_read(p, port_id, buf, size);
  ...
  DAL_write(p, port_id, buf, size);
  ...
}
void process_finish(DAL_process *p){...}
```

Table 3.1: Conversion of a HOPES process specification (left) to the DAL equivalent (right).

The description of an application on the HOPES framework consists of a set of processes connected to a process network. Each process is defined by three functions: *{process name}_init()*, *{process name}_go()* and *{process name}_wrapup()*. These functions can be directly used to describe the process in DAL, as the corresponding *{process name}_init(DAL_process *)*, *{process name}_fire(DAL_process *)* and *{process name}_finish(DAL_process *)* functions (Table 3.1). In addition, the HOPES APIs *MQ_RECEIVE* and *MQ_SEND* must be replaced by the corresponding DAL APIs *DAL_read* and *DAL_write*.

The major difference between the two frameworks lies in the description of the internal state of each process. In HOPES any variable that keeps its value between calls of the *go* function can be declared as *static*, which makes it a part of the process state. In DAL such a declaration is not desirable. A process can be instantiated more than once on the process network and as such, declaration of a variable as *static* may lead to usage of the same memory space by all instances of the process, with unpredictable results. Instead, DAL provides memory for the internal state of each instance of a process in the form of the struct *_local_states*. Hence, every static variable declaration of the HOPES implementation must be moved to the struct on DAL (*somevar* in the example of Table 3.1). Since some static variables may be global on HOPES, a pointer to the state struct is given to each function that needs to access it in DAL, thus ensuring a different instance of the variable for each process instance.

Finally, the way the application terminates changes between the two frameworks. HOPES uses a predefined maximum number of firings, known at design time, for each process. DAL, on the other hand, provides higher flexibility by giving the application the ability to signal the run-time controller for a scenario change when it has been completed or needs to pause. Therefore, it can terminate when the entirety of its input is consumed, regardless of its size, which is the termination method we will use in this work's applications.

3.2 Encoder

Using the transformation technique presented above, the base implementation for the encoder application on the DAL framework was created. In this section, the specifics of this implementation are presented.

The encoder implementation uses task division on the macroblock level. The set of functions is divided among five processes: *Init*, *ME*, *Encode*, *Deblock*, and *VLC*. The division of the application's function set is shown in Fig. 3.1.

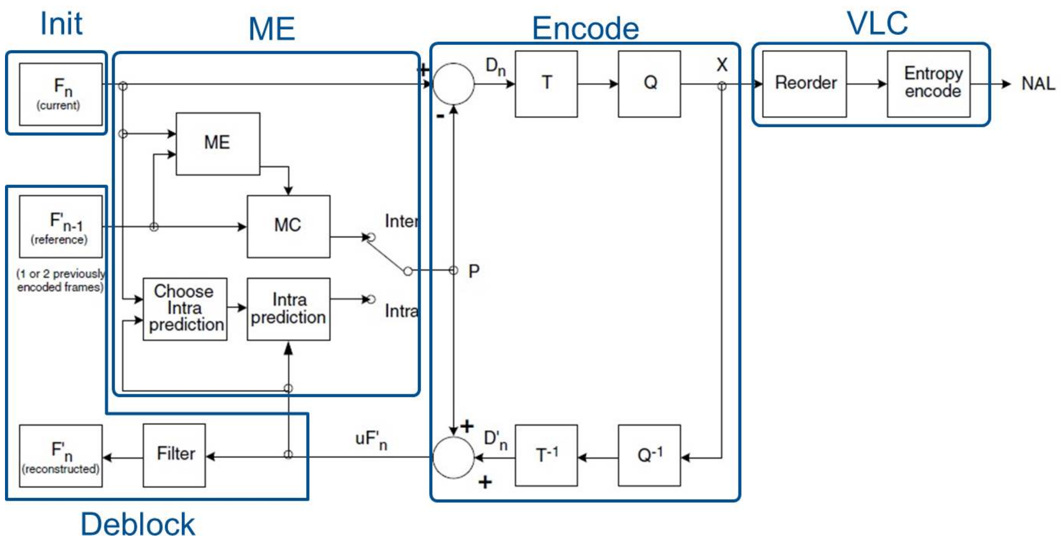


Figure 3.1: Division of the encoder functions in processes.

The application accepts the input video in raw YUV sequence file format and the *Init* process reads one frame of the input file at a time. Frames are specified as I or P ones, based on the GOP size value; the first frame of each GOP is specified as I and the rest as P. Subsequently, each frame is divided in macroblocks, which are scanned in raster order and fed to the ME process.

The *ME* process reads one macroblock in each firing sent from the *Init* process and performs inter- and intra-prediction[1], using the macroblock's reference and neighbouring data that is communicated from the *Deblock* process. The results of the two predictions are compared and the best one is used as the macroblock's predictor, which is sent to the *Encode* process.

In turn, the *Encode* process uses the predictor and the raw macroblock to calculate the residual, which is subsequently transformed and quantized to form the coded macroblock. This result is sent to the *VLC* process and is also reconstructed, with the reconstruction being passed onto the *Deblock* process.

The *Deblock* process stores the reconstructed macroblocks to form a database containing the reference frame's macroblocks used for inter-prediction, and the current frame's macroblocks used for intra-prediction. Thus, when it receives a reconstructed macroblock, it saves it in the database and loads macroblocks that are needed for inter- and intra-prediction of the next macroblock in line. The *Deblock* process also performs the task of deblocking filtering. Since intra-prediction requires unfiltered macroblocks, and inter-prediction requires the filtered ones, frames that are used as references are filtered when they are fully processed. When the final macroblock of such a frame is received, the process performs the deblocking filter to the whole frame and saves it as a reference.

Finally, the *VLC* process reorders the elements of the received coded macroblock and produces the final bitstream by applying variable length coding on these elements and every other necessary information needed for reconstructing the macroblock (e.g., the macroblock type, possibly the motion vector etc.). It also creates the headers of the NAL units and writes all this data on the output file.

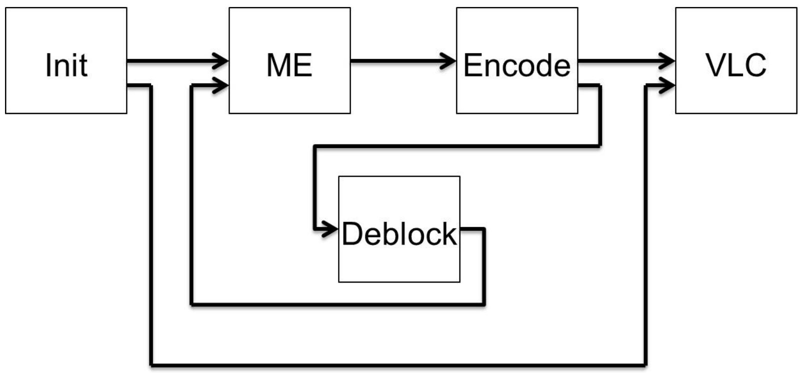


Figure 3.2: The process network of the base DAL implementation of the encoder.

[1]Only intra-prediction is performed if the macroblock belongs to an I-frame.

The process network of the encoder is shown in Fig. 3.2. The communication channels follow the flow of data as described earlier. The only exception is the channel from the *Init* process to the *VLC* process. This channel is added to signal the end of the input file to the *VLC* process, which in turns signals the run-time controller to terminate the application. For signalling between the two processes we exploit the fact that they both fire once for each macroblock of the input video. On each firing, the *Init* process sends a token with a predefined value indicating that the end of the input file is not reached. When the input is fully read, the *Init* process sends a different predefined value and terminates its own execution. At this point, the processes *ME*, *Encode* and *Decode* will not receive any inputs and will remain inactive, with the possibility of being terminated. The *VLC* process reads one token from the *Init* process on each firing and interprets its value, thus knowing when the input is over and the run-time controller needs to be signalled.

3.3 Decoder

In accordance to the encoder application, the DAL implementation of the decoder application was derived from the corresponding HOPES implementation.

For the decoder implementation, task parallelism is used as well. The set of functions is now divided among seven processes: *Readfile*, *Decode*, *PredY*, *PredU*, *PredV*, *Deblock*, and *WriteFile*, as shown in Fig. 3.3.

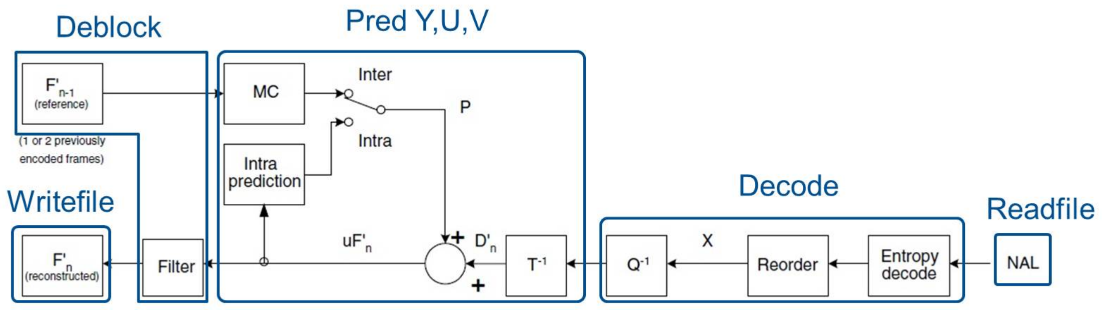


Figure 3.3: Division of the decoder functions in processes.

The *Readfile* process reads the input bitstream, checks it for errors, and decodes any header information. This information, along with the still encoded payload, is sent to the *Decode* process. Each NAL unit corresponds to one slice of the encoded video, and since one slice per frame is used, the *Readfile* process fires once per frame.

The *Decode* process, as the name suggests, decodes the variable length coding, reorders and rescales the NAL units' data to form the transformed residuals. These data are sent to one of the three *Pred* processes, according to the luma or chroma component they represent. However, macroblocks are transmitted in 4x4 blocks, so that the necessary computations are pipelined.

Since the luma and the two chroma components are independent, the tasks of inverse transformation, motion compensation and reverse intra-prediction for each component can be calculated in parallel. As such, the processes *PredY*, *PredU* and *PredV* perform these tasks concurrently for each 4x4 block they receive from the *Decode* process, by using as reference the data received by the *Deblock* process. The results of the *Pred* processes are the reconstructed blocks, which are sent to *Deblock*.

The *Deblock* process once again acts as a database for references, similar to its counterpart of the same name from the encoder application. Once each frame is completed, it performs the deblocking filter to enhance the quality, and sends it to the *Writefile* process, which simply writes it to the output file.

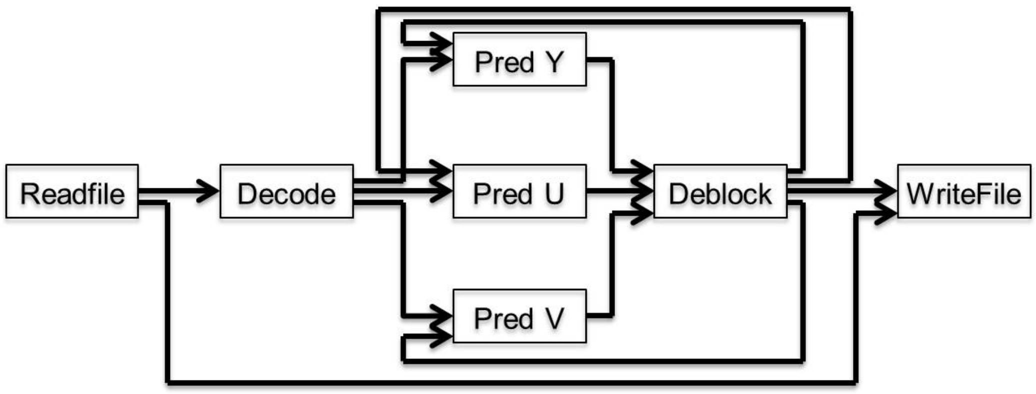


Figure 3.4: The process network of the DAL implementation of the decoder.

The process network of the decoder application is shown in Fig. 3.4. Once again, an extra channel is introduced from the *Readfile* process to the *Writefile* one to signal the end of the input file for the application's termination. The same principle as in the encoder application is used.

3.4 Increasing the Degree of Parallelism

From the two applications that were implemented, the encoder exhibits greater potential for exploring its intra-application dynamism. This is due to the fact that the encoder has an endless range of possible inputs, decides all the parameters concerning the way they are encoded (e.g., selecting the GOP size, the types of frames) and also contains the Motion Estimation task which can be the subject of a whole different work by itself. The decoder, on the other hand, can also have a very diverse set of inputs, but is limited by the way it processes them, because it is bounded by the decisions of the encoder. Thus, for the rest of this work, we will focus our efforts on the encoder, with some of our ideas being applicable to the decoder, as well.

In the process network of the encoder (Fig. 3.2), we can see that the *Init* process can run independently from the rest, and so does the *VLC* one, as long as it receives input from the *Encode* process. However, the feedback loop between the other three processes prohibits their parallel execution. This loop is imposed by the dependencies of the encoding algorithm; the *ME* process has to use previously encoded macroblocks as references. Therefore, in this implementation, the *ME*, *Encode*, and *Deblock* processes are always serialized. To alleviate this problem, we must change the order in which the macroblocks of a frame are being processed, so that the *ME* process does not always depend on the previously processed macroblock. The first step towards this goal is to identify all macroblock dependencies.

As already mentioned in Section 2.4, a macroblock depends on its neighbouring macroblocks in order to be processed (Fig. 3.5). In the case of inter-prediction these neighbours are the left, top, and top-right macroblocks of the currently processed one. In the case of intra-prediction the dependencies include the top-left neighbour, as well. Thus, in the general case, for a macroblock to be processed, its left and top-right macroblocks must have been already processed. If this condition is met, the rest of the dependencies are resolved recursively.

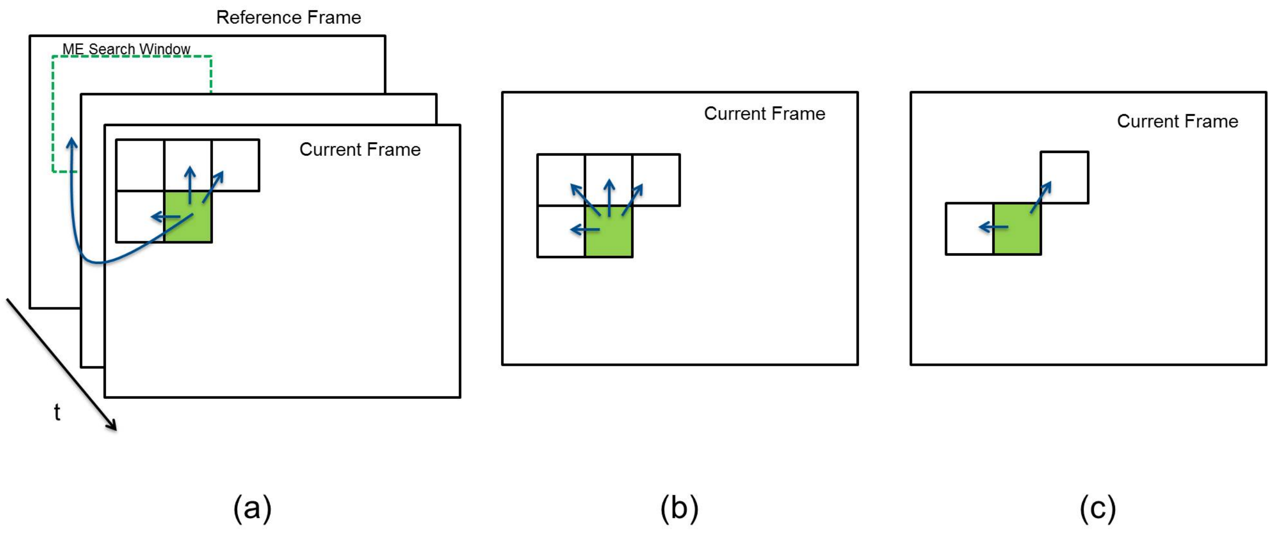


Figure 3.5: Dependencies of the currently processed macroblock: (a) during inter-prediction, (b) during intra-prediction, (c) in the general case.

It is evident that when the macroblocks are issued by the *Init* process in raster order, the above mentioned dependencies do not allow for more than one macroblock to be processed at a time. However, we can change the processing order and increase the number of macroblocks that can be processed in parallel, using for the *ME* process a popular technique called *wavefront parallelism* [7, 9]. This technique guarantees that all macroblocks that have their dependencies resolved are issued on every step and the degree of parallelism increases from one to a maximum value, and then decreases back to one. An example of wavefront parallelism can be seen in Fig. 3.6, for a frame consisting of 4x7 macroblocks.

In this example, we see that the first two macroblocks are processed sequentially, because of their dependencies. However, on the third step, the degree of parallelism increases by one, because there are two macroblocks with resolved dependencies. In the same manner, the degree keeps increasing until a maximum value of four, and then gradually decreases back to one.

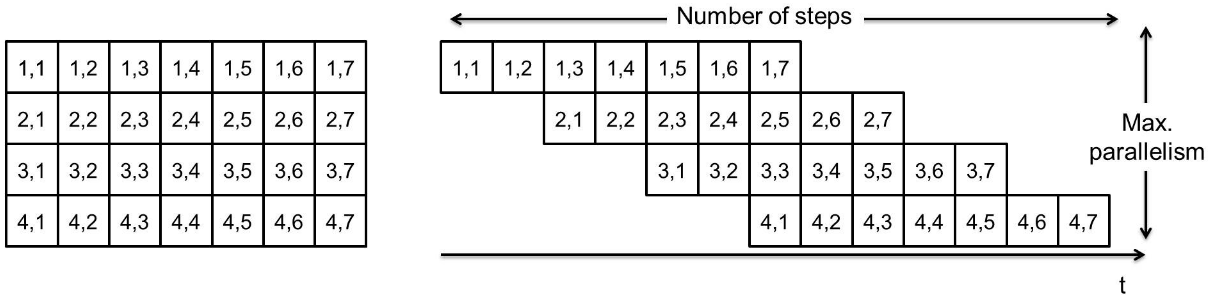


Figure 3.6: Example of the order that macroblocks are processed when using wavefront parallelism.

In this technique, the maximum achievable degree of parallelism depends on the size of the input video frame. A frame with $w * h$ resolution, has $MB\_width = (w + 15)/16$ [2] macroblocks in the vertical direction and $MB\_height = (h + 15)/16$ macroblocks in the horizontal one. The maximum achievable degree of parallelism is:

$$max\_par = min((MB\_width + 1)/2, MB\_height) \tag{3.1}$$

The number of steps required for processing that frame are:

$$\#steps = MB\_width + 2 * (MB\_height - 1) \tag{3.2}$$

[2] / symbolizes integer division, fractions symbolize regular division.

The average degree of parallelism is:

$$avg\_par = \frac{MB\_width * MB\_height}{\#steps} \tag{3.3}$$

Using wavefront parallelism on the *ME* process enhances the encoder in two ways: multiple macroblocks can be predicted at the same time and the firings of the *Encode* and *Deblock* processes can overlap. As such, we improve the encoder implementation by multiplying the *ME* process as many times as max_par for the given input resolution. For spawning multiple instances of the process on the process network specification, the iterators of the DAL framework are used. Fig. 3.7 shows how the process network is altered for a frame with $max\_par = 4$.

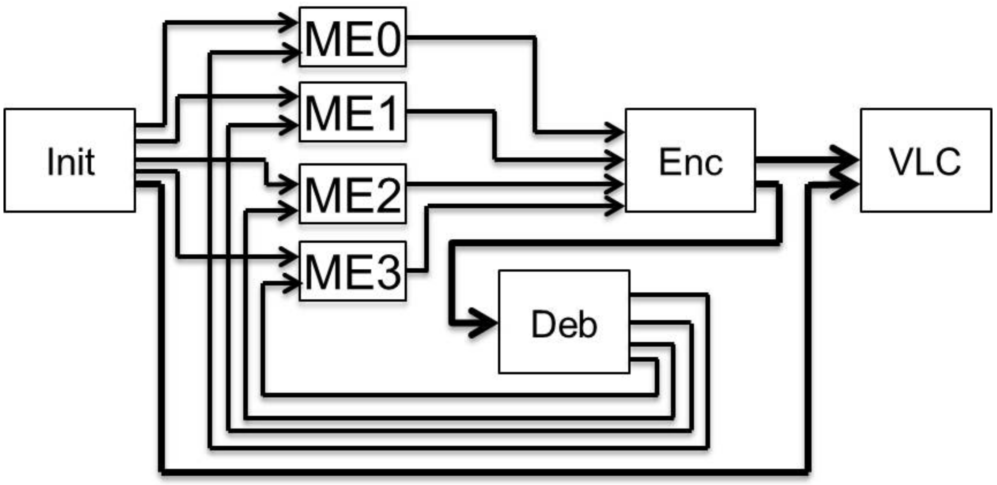


Figure 3.7: The process network of the improved encoder application for 4 ME instances.

The code for the *ME* process does not need to change, as it does not keep any internal state between consecutive firings; it just processes any input it is given. On the other hand, the code of the *Init*, *Encode*, *Deblock* and *VLC* processes has to be modified. The *Init* process is changed so as to issue, on each step, all macroblocks that have their dependencies resolved. Each macroblock line of the input frame is sent to a different *ME* process: a macroblock with coordinates (i, j) is sent to the *ME* process with index $(j\%\#ME\_processes)$. The order in which the macroblocks are issued can be easily computed by the frame's resolution, and so it is known by every process of the application. As such, the *Encode* process is changed to read from the correct *ME* on each firing, in the same order as they were issued by the *Init* process.

The most important changes were made to the *Deblock* and *VLC* processes.

Deblock does not fire for every macroblock, as it used to, but rather fires for each step. It waits to receive all the reconstructed macroblocks of the previous step, and saves them one by one as they are received. After they have all been collected, it sends the data needed for the next step to the *ME* processes, which start their execution at almost the same time. The *VLC* process has to change because of the fact that the output bitstream describes the macroblocks in raster order. Previously, when the macroblocks were processed in the same order, each one that reached the *VLC* process was written directly to the output. However, now that the order has changed, partial bitstreams for each macroblock must be stored until the correct ones are received and they can all be written to the output.

In this improved implementation, we have managed to increase the degree of parallelism of the encoder, but most importantly, we have shown that this degree is a function of the input video resolution. We have managed to introduce some dynamism on the encoder application, moving one step towards the goal of this project.

3.5 Improving the Performance

Using the wavefront parallelism technique on the encoder in the previously described manner, the average degree of parallelism is lower than the maximum achievable one. In this section a method is proposed for increasing the average parallelism, therefore increasing the performance. We need to point out that this method was not implemented, since it exceeded the goals of this project.

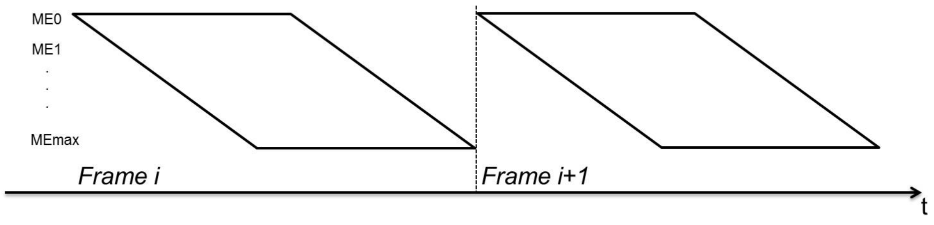


Figure 3.8: Execution of two successive frames in the current implementation, as described in the previous subsection.

Fig. 3.8 shows the execution of two consecutive frames in time. As shown, a frame can only start being processed when all the macroblocks of its previous frame have been completed. This restriction is imposed because, during inter-prediction, a macroblock depends on macroblocks that have already been processed by the deblocking filter. Since the filtering is performed on the whole frame after it is completed, the next frame can only start being processed afterwards. This results in periods in time where each *ME* process remains inactive, as shown in the figure. Thus, the average parallelism achieved throughout the encoding of the whole video equals the average parallelism for one frame, which is inherently less than the maximum.

In order to address this problem, we propose a pipelined scheme of frame processing, by modifying the implementation of the deblocking filter. Using this scheme, the execution of two successive frames will be as shown in Fig. 3.9. In this figure we see that the gaps between consecutive firings of the *ME* processes are no longer present. Hence, the average degree of parallelism will become almost equal to the maximum.

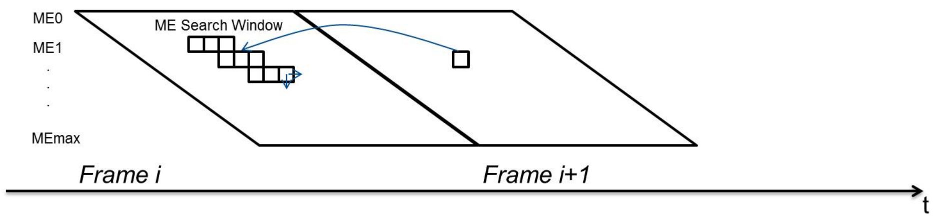


Figure 3.9: Pipelining two successive frames.

In the previous implementation we have considered that when a frame is processed, all previous frames are already processed and can be directly used as references. However, this is no longer the case in the described approach as at some time intervals, the processing of two successive frames overlap. Two main concerns arise: the deblocking filtering and the inter-prediction dependencies.

The deblocking filter implementation must change so that filtering is performed on each macroblock separately as soon as possible, because this macroblock may be needed as an inter-prediction reference before its frame's execution is over. The dependencies of a macroblock for the filtering are its *bottom* and *right* neighbours [8]. This means that a macroblock can be deblocked only after the next time step, when its right neighbour will be processed. Therefore, after receiving the macroblocks of the current step, the new implementation of the *Deblock* process should deblock the macroblocks of the previous step.

In addition, the inter-prediction's dependency from a previous frame must be revisited. When a macroblock from the current frame is processed, all macroblocks from its reference frame that belong in its Motion Estimation window must be already processed (Fig. 3.9). This could potentially cause a problem for the frame directly succeeding the reference. Adding the dependency of the deblocking filter, the current macroblock can be processed at least after $ME\_search\_range + 1$ time steps after the macroblock in the same position on the reference frame. In case this cannot be done, gaps must be inserted in the pipeline, resulting again in a decreased average degree of parallelism. For frequently used search ranges, even in small resolutions like QCIF (11*9 macroblocks) there should not be a problem. However, if the search range is increased for quality improvement, attention must be given in the dependencies of the pipeline.

The proposed method is expected to increase the performance of the encoder significantly, by increasing its degree of parallelism.

3.6 Summary

In conclusion, in this chapter the implementation of the Codec pair for the DAL framework is presented. First, the methodology for converting a HOPES application specification to its DAL equivalent is discussed. Afterwards, using this methodology, the initial implementations of the encoder and decoder are created, from the code provided by the SNU. Deciding to focus on the encoder for the rest of the work, its main problem for achieving parallel execution is discussed and a solution is implemented, which enables some static dynamism on the application as a reaction to the input video resolution. Finally, a technique is proposed for improving the overall performance of the encoder.

4 Run-Time Dynamism

The implemented H.264 encoder, described in the previous chapter, adjusts its degree of parallelism according to the input video frame size. This is a form of static intra-application dynamism, since it is unlikely that the resolution changes during a video encoding process. Nevertheless, streaming applications like this one often have strict throughput constraints. For example, real-time video transmission requires a constant frame rate of at least 24 frames per second (fps) for a frame sequence to be perceived as a video by the human visual system, while HDTV standards require 60 fps. For large video resolutions, i.e., 720p and 1080p, achieving the necessary framerate can be challenging. Furthermore, the workload that a video introduces does not only depend on the video's resolution, but also on the visual data; quick motion and scene changes increase the computational requirements. Thus, the encoder must react to high-workload input video at run-time whenever there is a risk of not meeting the constraints.

As a means of introducing run-time dynamism on the encoder application, we propose balancing the tradeoff between the achieved fps and the video compression rate. The encoder can easily monitor the fps by measuring the elapsed time between the processing of the last macroblocks of two successive frames. In case the achieved fps drops and gets close to the lower bound imposed by the constraints, compression rate can be sacrificed in favour of better performance. Conveniently enough, the H.264 standards provides opportunities to balance this tradeoff by adjusting encoding parameters at run-time. In this chapter we propose three such parameters that can be used to provide run-time dynamism.

4.1 I to P Frames Ratio

The first parameter that is proposed for run-time dynamism is the ratio between I and P frames. Since the first frame of each GOP is specified as an I frame and the rest as P frames, the ratio can be adjusted by increasing or decreasing the GOP size. Macroblocks belonging to I frames are only intra-predicted, while macroblocks from P frames are both inter- and intra-predicted. Thus, I frames are encoded faster than P frames, which means that higher fps can be achieved. However, the compression rate of P frames is higher, since both prediction techniques are used on their macroblocks. The most accurate of the two predictions is selected, which in turn produces a residual with less energy, needing less bits to be encoded.

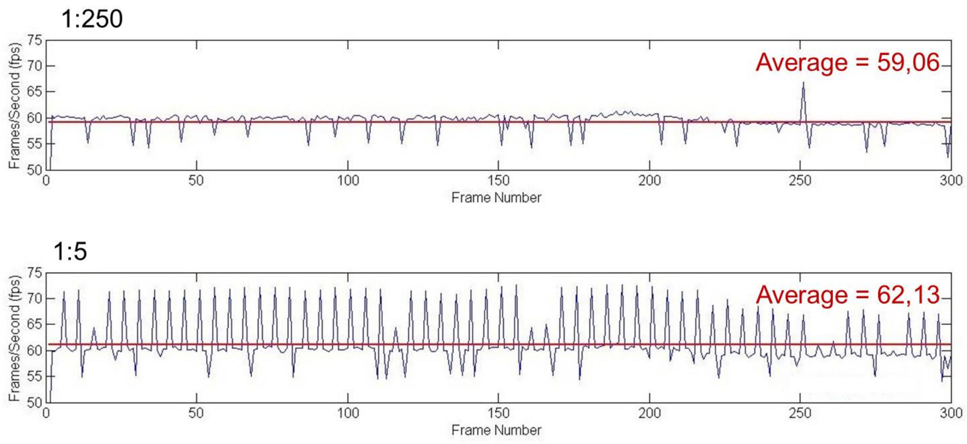


Figure 4.1: Fps per frame for 1:250 (top) and 1:5 (bottom) I to P frames encoding of a 300 frames, CIF resolution video.

Fig. 4.1 shows the fps achieved for each frame for encoding 300 frames of a CIF resolution (352*288) video using the implemented encoder and two different I to P frames ratios [1]. For the 1:250 rate, it is evident that the fps for the P-frames is just below 60, but spikes on the sole I-frame (frame 251) to approximately 70 fps. By increasing the rate to 1:5, more spikes are introduced, which effectively increases the average fps of the encoding process by approximately 5.1%. The input 45.5 MB video is compressed to 3.82 MB for the lower ratio and to 4.04 MB for the higher ratio. This means that along with the speedup, a drop of approximately 5.4% is witnessed in the compression rate, as expected. These results show that it is indeed possible to balance, within a certain range, the achieved fps and output compression rate tradeoff by adjusting the I to P frames ratio at run-time.

[1]The experimental setup for these measurements corresponds to the setup described in 5.2.

4.2 Motion Estimation Search Range

Another parameter that influences both the framerate and compression rate of the encoder is the range of the search window for the Motion Estimation algorithm. This algorithm is part of the inter-prediction process and therefore only affects P frames. Still, it is one of the most computationally intensive parts of the encoder, and its behaviour can influence the behaviour of the whole application. This algorithm calculates the vector of a macroblock's motion between the current and reference frames. In order to perform this calculation, the algorithm calculates the macroblock of the reference frame that best matches the macroblock currently being encoded, with the comparison between two macroblocks being determined by some metric, e.g., the Sum of Absolute Transformed Differences (SATD).

Usually, a full search algorithm that compares the current macroblock with all macroblocks of the reference frame is too slow and is not used, despite the fact that it calculates the optimal matching. Instead, heuristic methods are used, which, in the general case, only compare the current macroblock with macroblocks within a certain window in the reference frame. The search window is centered at the macroblock with the same coordinates in the reference frame (Fig. 4.2). The shape of the window and the starting point of the search varies according to the algorithm used. Frequently, early termination techniques are used in the search, when a macroblock that matches better than its neighbours is found. In our implementation a hexagonal search window is used, the starting point of the search depends on the motion vectors of the neighbours, and early termination is used to speed up the execution.

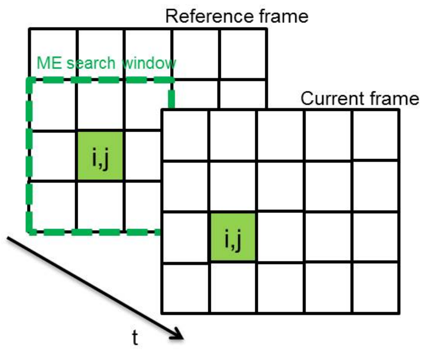


Figure 4.2: Example of a Motion Estimation search window.

The range of the search window can determine the speed of the Motion Estimation algorithm. Using a larger range, more comparisons are possible, resulting in increased execution time. However, more comparisons means that there is a higher probability of a better matching, which in turn produces a residual containing less energy, thus requiring less bits to be encoded. Therefore, adjusting the Motion Estimation search window range at run-time can balance the fps-compression rate tradeoff, as desired.

4.3 Slices Per Frame

On both the encoder and decoder implementations in this work, it is assumed that each frame is encoded in a single slice, which is a popular method in small and medium resolutions. However, it is possible to split each frame in more than one slices, and the number of slices per frame can vary between frames. Thus, this property can be used as another parameter that is adjusted at run-time for achieving dynamism.

As mentioned earlier, slices are encoded and decoded independently. Hence, it is possible to process them in parallel, by multiplying the corresponding parts of the process network on different cores. This slice-level parallelism decreases the processing time of each frame which in turn increases the fps of the application. However, the higher the number of slices per frame, the lower the compression rate. This is attributed to two main reasons. Firstly, there is less flexibility in encoding a single macroblock, because only data within the same slice can be used. This means that suboptimal predictors are calculated, resulting in suboptimal compression. Secondly, each slice requires separate header data and as such, higher slice numbers result to higher data overhead on the encoded video.

Therefore, the number of slices per frame can be dynamically adjusted at run-time and increase or decrease the degree of parallelism of the encoder application, providing either increased fps or better compression rate, according to the application's constraints.

4.4 Summary

In this chapter, we proposed exploiting run-time dynamism on the encoder application, by balancing the tradeoff between time and compression performance. In order to do so, three encoding parameters are presented that can be adjusted at run-time. The effects of each of these parameters is shown in Table 4.1. The I to P frames ratio is the most easily controlled of the three and the search range of the Motion Estimation algorithm is a parameter which greatly affects the application's behaviour. Finally, adjusting the number of slices per frame can enable frame-level parallelism, which cannot be exploited by the current implementation.

Parameter	Options	Computation Time	Compression Rate
I to P frames ratio	I frames	+	-
	P frames	-	+
Motion Estimation Search Window Size	Large Window	-	+
	Small Window	+	-
Number of Slices per Frame	Multiple Slices/Frame	+	-
	One Slice per Frame	-	+

Table 4.1: Summary of the effects of different encoding parameters.

5 Experiments

In this chapter, the evaluation of the wavefront parallelism technique used on the implemented encoder is presented. Focus is given on the execution time of the application and the speedup achieved by implementation-specific parameters. Other metrics, like the compression rate and quality of the coded video, are not considered, as they depend on the encoding parameters of the H.264 standard which remain unchanged throughout all experiments. First, the theoretical model of the application is presented in Section 5.1, according to which the maximum theoretical speedup achieved with wavefront parallelism is calculated. Then, in Section 5.2, the experimental setup is discussed and Sections 5.3 and 5.4 explore the effect of the mapping and number of ME processes on the encoder, respectively.

5.1 Model of the Application

As explained in Section 3.4, the behaviour of the application depends on the *ME*, *Encode* and *Deblock* processes, due to the dependencies that are introduced by the encoding algorithm. The wavefront parallelism technique changes the execution timings of these processes and as such, the model only considers them to calculate the theoretical speedup.

As also shown in that section, the maximum degree of parallelism is a function of the input video resolution (Eq. 3.1). The average degree of parallelism depends on the resolution as well, and is lower than the maximum parallelism (Eq. 3.3). Thus, for each input video resolution, a (max_par,avg_par)

pair is defined.

A (max_par,avg_par)-encoder is equivalent to a theoretical encoder with a constant degree of parallelism equal to avg_par. This theoretical encoder will be represented as (avg_par)-th.encoder, and will be used as a model for calculating the maximum speedup.

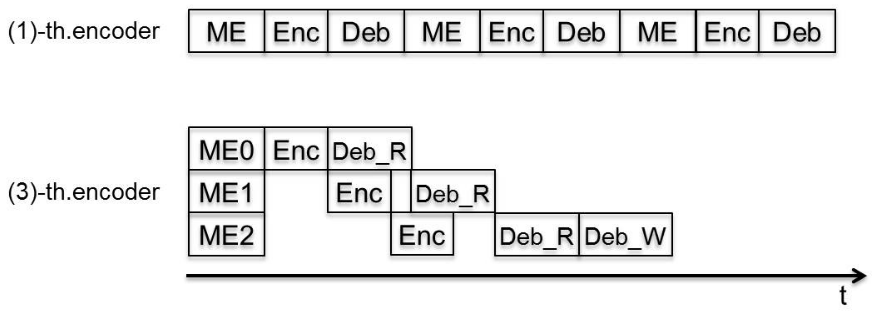


Figure 5.1: Example of the timings of the processes for the theoretical encoder.

Fig. 5.1 shows an example process pipeline for encoding three macroblocks on a (1)-th.encoder and a (3)-th.encoder. On the (3)-th.encoder, the *Deblock* process is represented as two separate blocks. *Deb_ R* represents the reading and saving of the coded data of each macroblock, while *Deb_ W* represents the time needed for loading and communicating the macroblocks needed for the next step, along with the time needed for performing the filtering. On the (1)-th.encoder, the *Deblock* process is approximated as a single *Deb* block per macroblock. The execution time of this approximation is calculated as the sum of the time of one *Deb_R* (t_{Deb_R}), plus the time for one write. *Deb_ W* represents the writing of avg_par macroblocks, thus one writing requires $\frac{t_{Deb\_W}}{n}$ time. This means that: $t_{Deb} = t_{Deb\_R} + \frac{t_{Deb\_W}}{n}$.

This model shows that we can calculate the execution time for encoding n macroblocks for the two theoretical encoders, using the following formulas:

$$t_{(1)-th.encoder} = (t_{ME} + t_{Enc} + t_{Deb}) * n$$

$$t_{(avg\_par)-th.encoder} = \big(t_{ME} + t_{Enc} + t_{Deb\_R} + t_{Deb\_W} + (avg\_par - 1) * max(t_{Enc}, t_{Deb\_R})\big) * \frac{n}{avg\_par}$$

Subsequently, when using an average parallelism of avg_par, the speedup gained compared to the sequential execution of one *ME* process is calculated as follows:

$$speedup(avg\_par) = \frac{(t_{ME} + t_{Enc} + t_{Deb}) * avg\_par}{t_{ME} + t_{Enc} + t_{Deb\_R} + t_{Deb\_W} + (avg\_par - 1) * max(t_{Enc}, t_{Deb\_R})} \quad (5.1)$$

In order to calculate the maximum theoretical speedup for a given input video resolution, first the average degree of parallelism is calculated from Eq. 3.3. Then, the average execution time of the processes *ME* and *Enc*, and the two parts of the *Deblock* process are measured, and t_{Deb} is calculated. These values in turn give the maximum theoretical speedup of this implementation through Eq. 5.1.

5.2 Experimental Setup

All experiments were conducted on a machine with two 8-core Intel Xeon E5-2690 CPUs and 16 GB of shared memory. Two input video resolutions where used, QCIF and CIF. The properties of these two formats are shown in Table 5.1. The average and maximum degrees of parallelism, are calculated from Equations 3.1 and 3.3 respectively, and the maximum theoretical speedup is calculated as explained in the previous section. The execution time of each process' firing is measured as the user and kernel CPU time of the corresponding thread, to avoid measuring waiting and context-switching time. However, every measurement of the encoder's total execution time regards the elapsed time, to present the real capabilities of the application.

Format	Resolution	Max. parallelism	Avg. parallelism	Max. Theoretical Speedup
QCIF	176*144	6	3,67	2,37
CIF	352*288	11	7,07	2,25

Table 5.1: Properties of the formats used as inputs.

The process network of Fig. 3.7 is used, with as many *ME* processes as needed on every experiment. The goal of the experiments is to explore the effects of two important parameters of the implementation: the mapping and the number of *ME* processes.

5.3 Effect of Mapping

The first experiment explores the effect different mappings have on the encoder implementation. A 300 frames, QCIF video is used as input to an encoder with the maximum number of *ME* processes (6).

Four different mappings are compared:

1. For the first mapping, all processes are mapped on the same core.
2. The second mapping keeps all processes on the same core as the previous mapping, except the *ME* ones, which are mapped in pairs on different cores. In this case, four cores are utilized.
3. A greedy mapping scheme is considered next. Each process is mapped on a different core. In total, ten cores are used.
4. For the last mapping, information on the application's timings is used to reduce the number of cores that are needed. It is easy to deduce from Fig. 5.1 that the *Encode* and *Deblock* processes' execution never overlap with the *ME* processes' execution. Hence, *Encode* is mapped on the same core as *ME0* and *Deblock* is mapped on the same core as *ME1*. In addition, *Init* and *VLC* are mapped on the same core, even though their executions overlap, since they do not have high computational requirements. Seven cores are needed for this mapping.

The execution times of the encoder for each of these mappings are shown in Fig. 5.2. We can see that mapping 2 improves the execution time compared to mapping 1, since the degree of parallelism increases. However, when more that one *ME* processes are mapped on the same core, the maximum degree of parallelism cannot be exploited, as their executions overlap. This is the main reason why mapping 3 performs better than mapping 2. The final mapping is the best among the four, since it performs the best, on less cores that mapping 3. In the last two mappings, no processes' executions overlap on any core, but mapping 4 performs better. The reason for that is the fact that processes with high communication between them are mapped on the same cores, so this communication can benefit from the core's cache memory.

This experiment shows that a good mapping can affect the application's performance. The last mapping is most likely not the optimal one. It can be improved by creating the performance model of the application which would enable usage of the DAL framework's mapping optimizations. However, this model must be extremely accurate to achieve the optimal mapping. The optimal mapping would be given by an exhaustive search of the different mappings, but this would be very time consuming and expensive. Nevertheless, it is evident that utilizing information on the processes' execution can increase the effectiveness of the mapping .

Kim et al. [7] use the same input video to test the effects the wavefront parallelism in their implementation. Even though our base implementation was derived from the base implementation of that work, they achieve 1.56x speedup compared to a single core execution, while we achieve 2.18x.

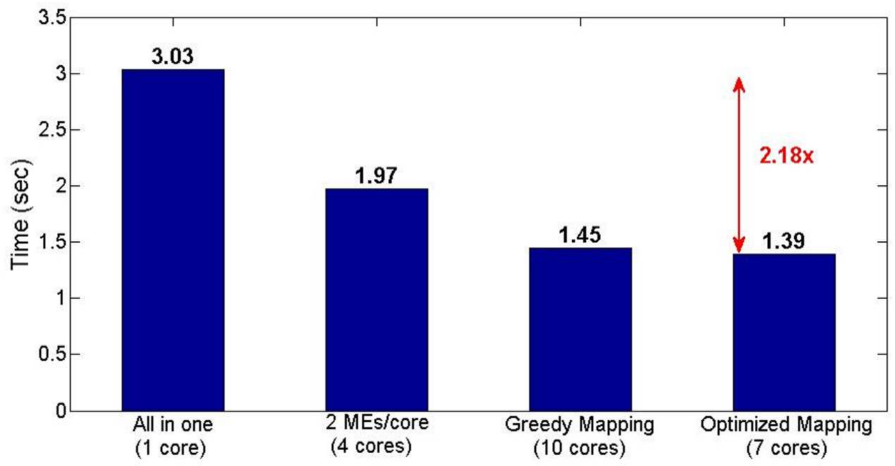


Figure 5.2: Execution time for a 300 QCIF video using four different mappings.

5.4 Effect of Number of *ME* Processes

The next experiment showcases the effect of varying the number of *ME* processes on the application's performance. A 1374 frames, CIF resolution video is used as input, in which a maximum of 11 *ME* processes can be utilized. The total elapsed time of the application's execution is measured, for different numbers of *ME* processes, ranging from 1 to the maximum 11. The fourth mapping from the previous experiment is used, therefore the number of cores used is equal to the number of *ME* processes plus one.

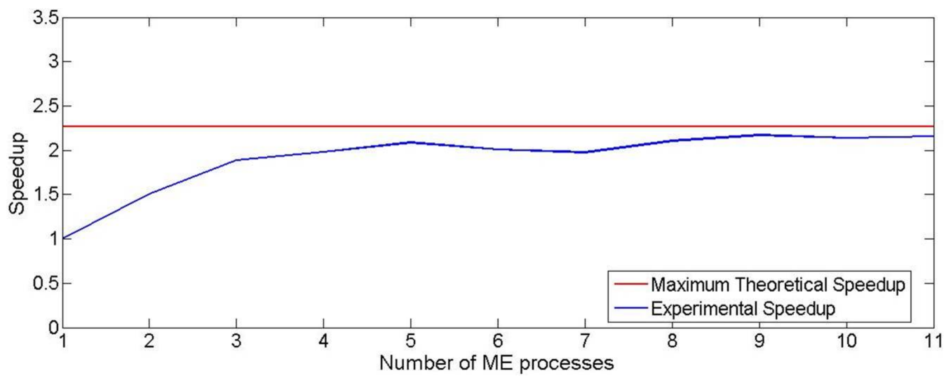


Figure 5.3: Speedup achieved for a varying number of *ME* processes, compared to the single *ME* process encoder.

Fig. 5.3 shows the speedup achieved with each *ME* processes count, compared to the execution time of the application with one such process. It is evident that the speedup increases as the number of processes increases. This is due to the fact that the predictions of more macroblocks are executed in parallel. As shown in the figure, the speedup is sub-linear. This is attributed to the fact that only a part of the application is parallelized, and even though some processes are pipelined, there are still parts of the encoder that are executed sequentially, i.e., the *Deblock* process always follows an *Encode* process. In addition, diminishing returns are witnessed as the number of processes increases, as is usually the case when increasing the parallelism of an application. This phenomenon can be attributed to the fact that the processes share the same system resources, such as the L2 cache memory and the memory bus, which can become congested as the communication traffic increases.

Nevertheless, using the maximum number of processes reaches close to the maximum achievable speedup. The divergence between the maximum theoretical and experimental speedups is attributed to the fact that the theoretical model does not incorporate some aspects of real-world systems, such as context switches and waiting time for memory accesses.

In addition, from the figure one can notice that the execution with 5 *ME* processes achieves a speedup very close to the one achieved with 11 processes. Therefore, using 5 processes gives a more efficient encoder, as half the cores are used.

The final remark on this experiment concerns the subject of Chapter 4. The adjustment of the number of *ME* processes can be another means of introducing run-time dynamism on the application. This way, the tradeoff between performance and number of utilized cores can be balanced at a frame-to-frame basis, without the need for reduced compression rate. For example, on a motion heavy scene, the encoder can use 5 *ME* processes. On another scene with less computational requirements, 3 processes can be used if the constraints can still be met, with the remaining two processes remaining inactive, which temporarily reduces the core count by two.

5.5 Summary

In conclusion, this chapter evaluates the performance of the wavefront parallelism-enhanced encoder application. A theoretical model of execution is created, from which the maximum achievable theoretical speedup is calculated. We performed two experiments that show how two different implementation parameters affect the performance. The first experiment regards the mapping of the encoder's processes and showed that a mapping that considers the application's characteristics is essential. It also shows that the achieved speedup is higher than that of related work [7]. The second experiment explores the number of *ME* processes that are needed for an efficient execution. It is shown that the maximum number of processes does not produce a cost-optimal execution and that the experimental speedup reaches close to the theoretical maximum.

6 Conclusion and Outlook

6.1 Conclusion

This project provides a first outlook to exploit intra-application dynamism on streaming applications, using an H.264 compliant encoder/decoder pair as a case study.

Initially, the main aspects of the H.264/AVC standard are summarized to provide a brief introduction and rudimentary understanding of the coding process for the reader. The Codec pair was implemented on the DAL framework, based on code for the HOPES framework, provided by the SNU. At first, non-dynamic implementations are created in the form of static process networks for the two applications. Focusing on the encoder application, the popular technique of wavefront parallelism is utilized in order to increase its degree of parallelism. This technique also enables static dynamism on the number of processes of the process network, which now depends on the resolution of the input video. The performance of the application increases with this technique, but we propose a pipelining scheme that can be used to further boost the performance.

In our implementation, only static dynamism is introduced. However, the coding standard provides opportunities for balancing the tradeoff between execution time and compression rate at run-time. We propose three encoding parameters that can be dynamically adjusted towards this goal: the I to P frames ratio, the search window of the Motion Estimation algorithm and the number of slices per frame.

Finally, the implemented encoder, enhanced with wavefront parallelism is evaluated. The methodology for calculating the maximum theoretical parallelism for each input video resolution is presented. Then, our experiments show the effect that the mapping and the varying number of *ME* processes have on the application's performance. Results show that a mapping that utilizes information on the processes' execution can benefit the performance, but also that less than the maximum number of *ME* processes can provide almost the maximum speedup. Furthermore, this implementation's achieved speedup is higher than that of related work [7] that uses the same technique .

6.2 Outlook

This project was only a first step towards an efficient intra-application dynamic Codec pair. Several enhancements of this first implementation have been proposed throughout this report and can be explored on any follow-up work.

Techniques for achieving run-time dynamism have been described and can be incorporated in the application in the future. By doing so, we can uncover the extend to which the dynamism improves the execution.

As far as performance is concerned, a frame pipelining scheme has been proposed, which can increase the average degree of parallelism on the encoder application. In addition, code and communication optimizations can be performed towards the same goal.

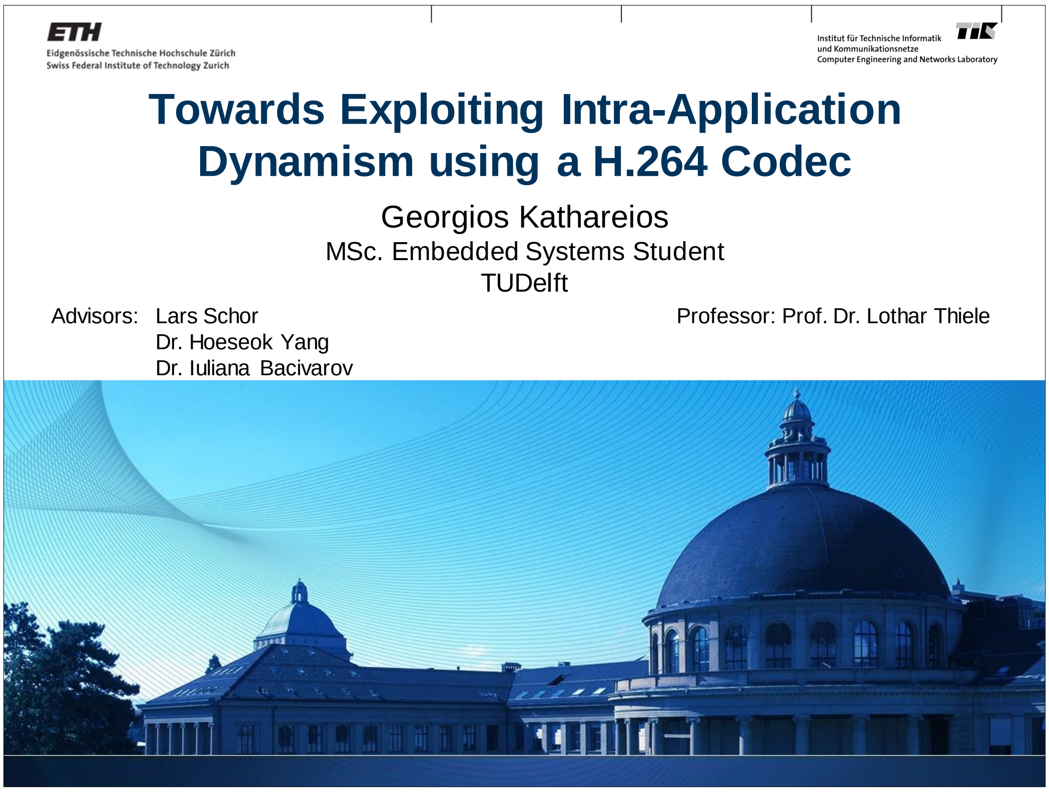
In addition, the wavefront parallelism technique and the previous proposals (when applicable) can also be used on the decoder, in order to determine whether the results are consistent on a different streaming application.

Finally, it would be very interesting to implement the main or extended profile of the H.264 standard, in order to provide more efficient compression rates, and an even more complex application for study.

A
List of Acronyms

API	Application Programming Interface
AVC	Advanced Video Coding
CAVLC	Context Adaptive Variable Length Coding
CIC	Common Intermediate Code
CIF	Common Intermediate Format
Codec	Encoder/Decoder
DAL	Distributed Application Layer
DCT	Discrete Cosine Transformation
EPN	Expandable Process Network
FPS	Frames Per Second
FSM	Finite State Machine
GOP	Group Of Pictures
HDTV	High Definition Television
HOPES	Hope Of Parallel Embedded Software development
KPN	Kahn Process Network
MB	Macroblock
MC	Motion Compensation
ME	Motion Estimation
MPEG	Moving Picture Experts Group
MPSoC	Multi-Processor System-on-Chip
NAL	Network Abstraction Layer
QCIF	Quarter Common Intermediate Format
SATD	Sum of Absolute Transformed Differences
VLC	Variable Length Coding

B
Presentation Slides



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

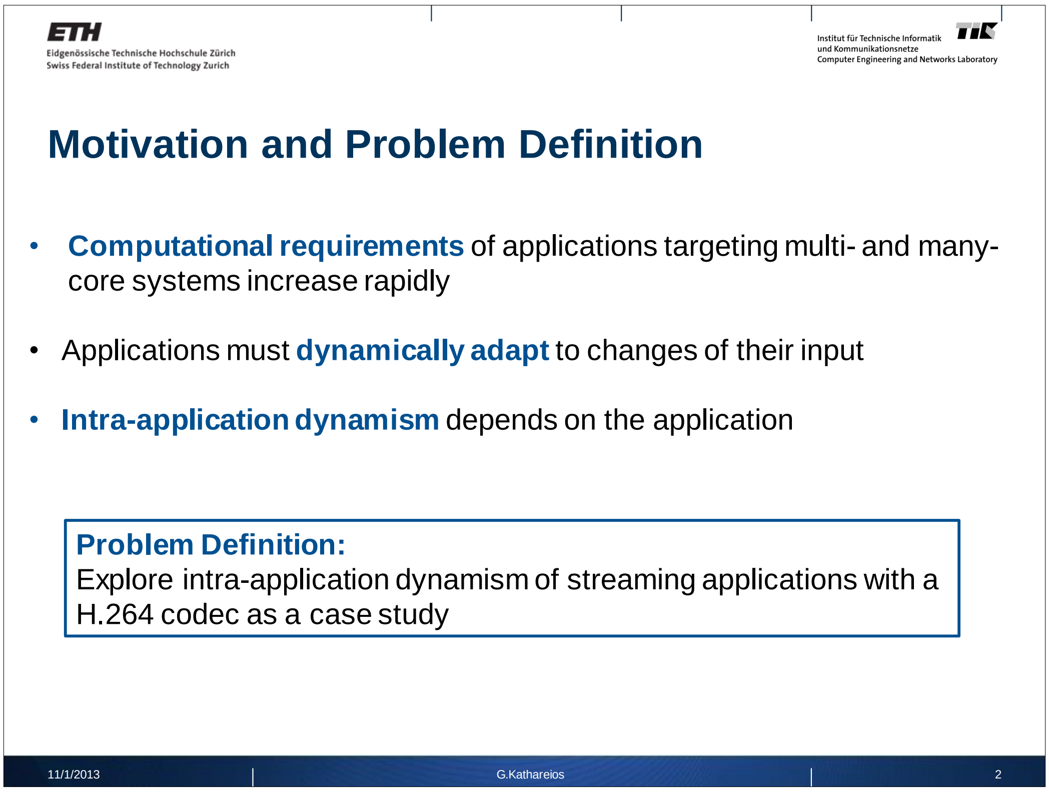
Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Towards Exploiting Intra-Application Dynamism using a H.264 Codec

Georgios Kathareios
MSc. Embedded Systems Student
TUDelft

Advisors: Lars Schor
Dr. Hoeseok Yang
Dr. Iuliana Bacivarov

Professor: Prof. Dr. Lothar Thiele



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

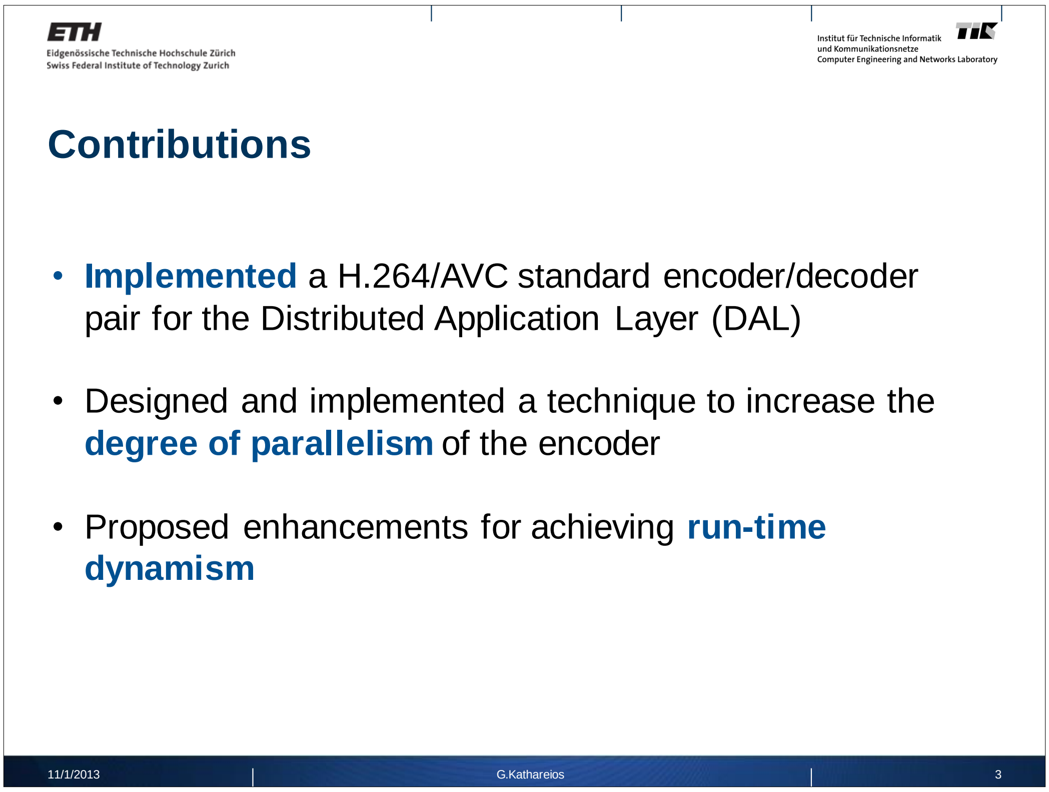
Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Motivation and Problem Definition

- **Computational requirements** of applications targeting multi- and many-core systems increase rapidly
- Applications must **dynamically adapt** to changes of their input
- **Intra-application dynamism** depends on the application

Problem Definition:
Explore intra-application dynamism of streaming applications with a H.264 codec as a case study

11/1/2013 | G.Kathareios | 2

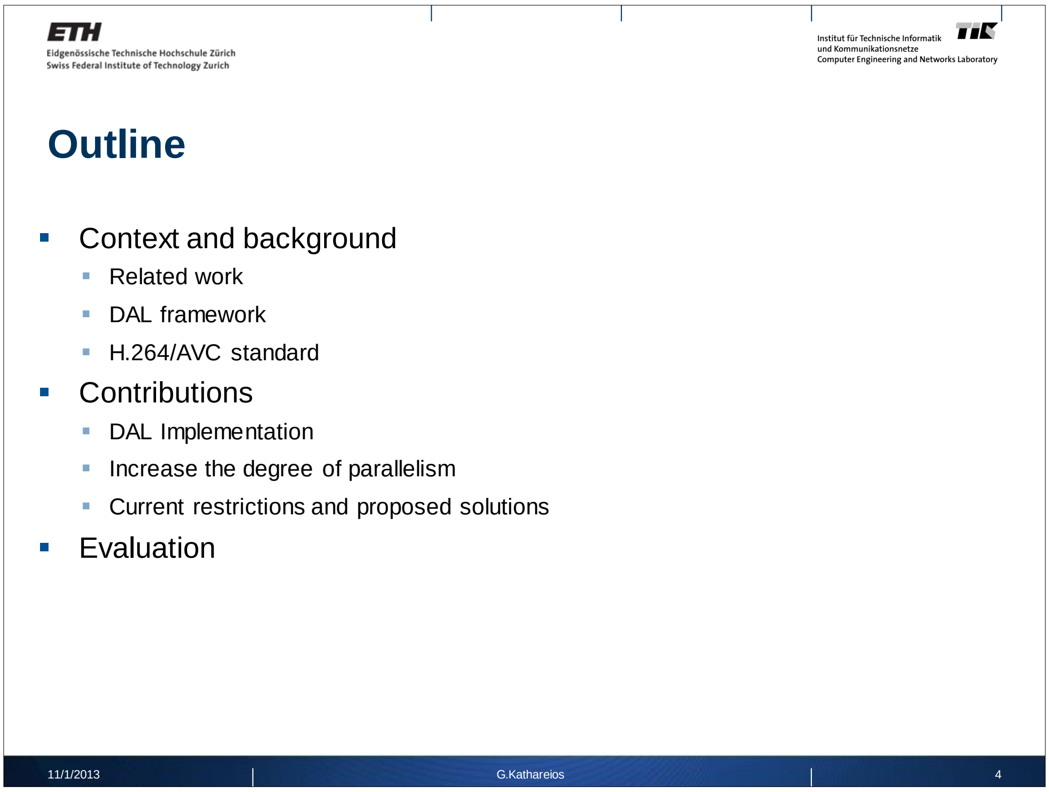

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contributions

- **Implemented** a H.264/AVC standard encoder/decoder pair for the Distributed Application Layer (DAL)
- Designed and implemented a technique to increase the **degree of parallelism** of the encoder
- Proposed enhancements for achieving **run-time dynamism**

11/1/2013 | G.Kathareios | 3

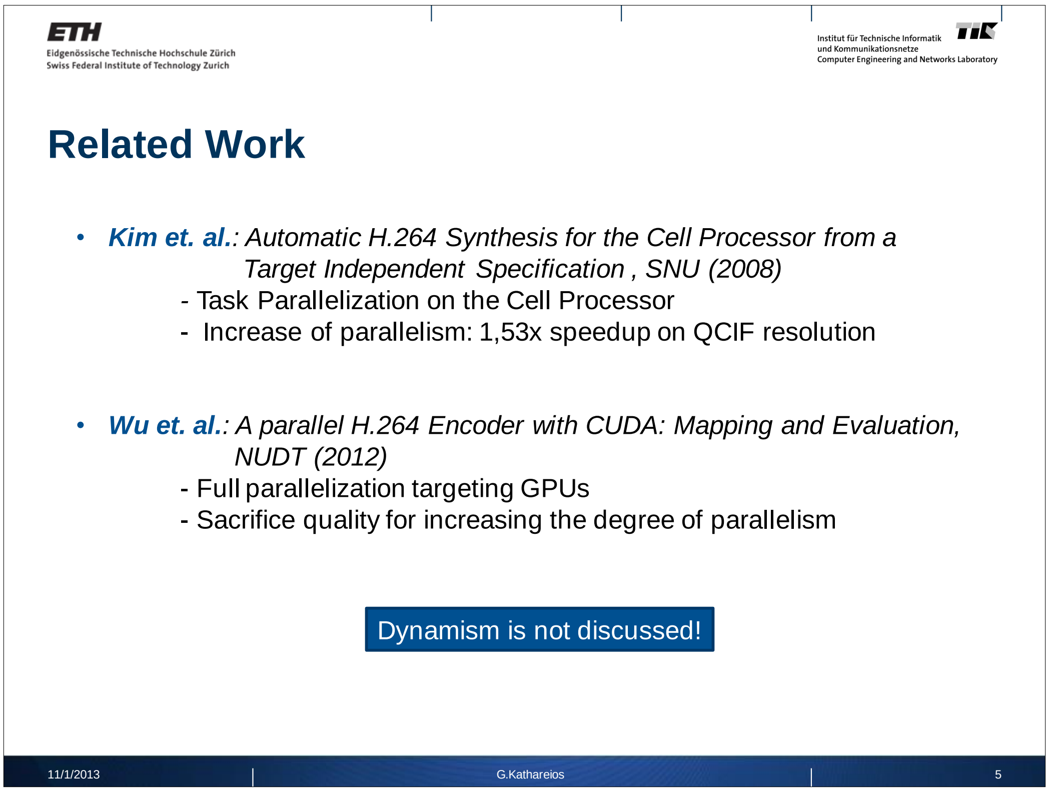

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Outline

- Context and background
  - Related work
  - DAL framework
  - H.264/AVC standard
- Contributions
  - DAL Implementation
  - Increase the degree of parallelism
  - Current restrictions and proposed solutions
- Evaluation

11/1/2013 | G.Kathareios | 4



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

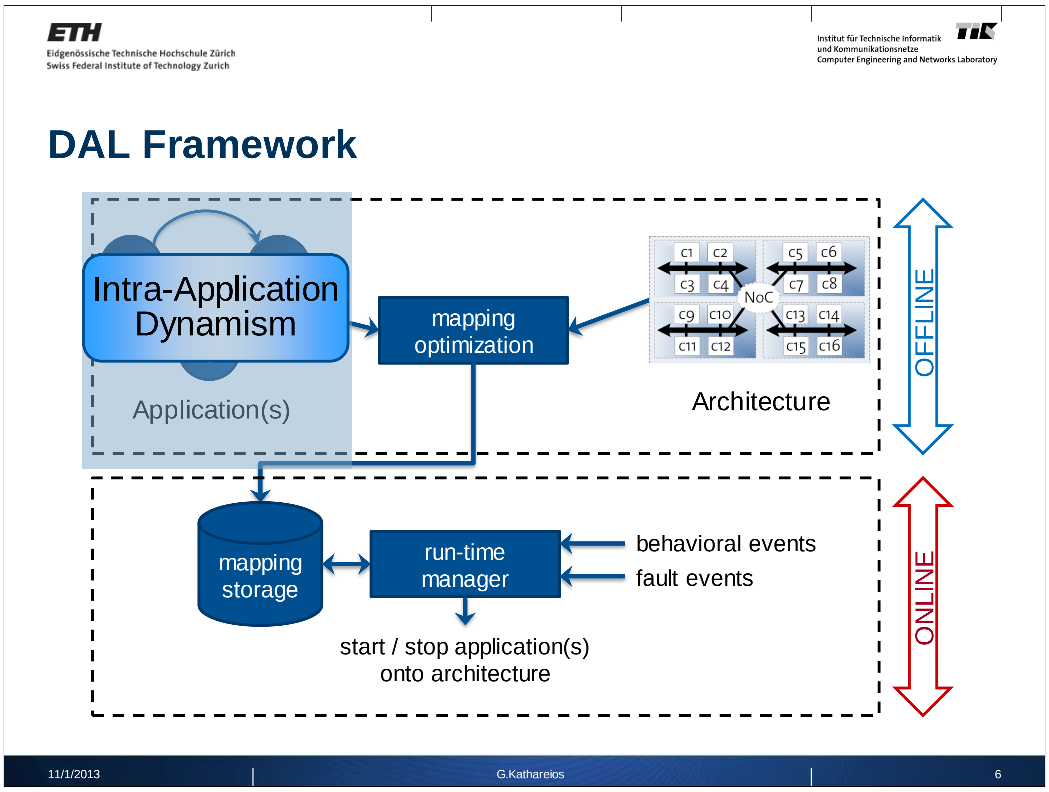
Related Work

- ***Kim et. al.***: *Automatic H.264 Synthesis for the Cell Processor from a Target Independent Specification , SNU (2008)*
  - Task Parallelization on the Cell Processor
  - Increase of parallelism: 1,53x speedup on QCIF resolution

- ***Wu et. al.***: *A parallel H.264 Encoder with CUDA: Mapping and Evaluation, NUDT (2012)*
  - Full parallelization targeting GPUs
  - Sacrifice quality for increasing the degree of parallelism

Dynamism is not discussed!

11/1/2013 | G.Kathareios | 5

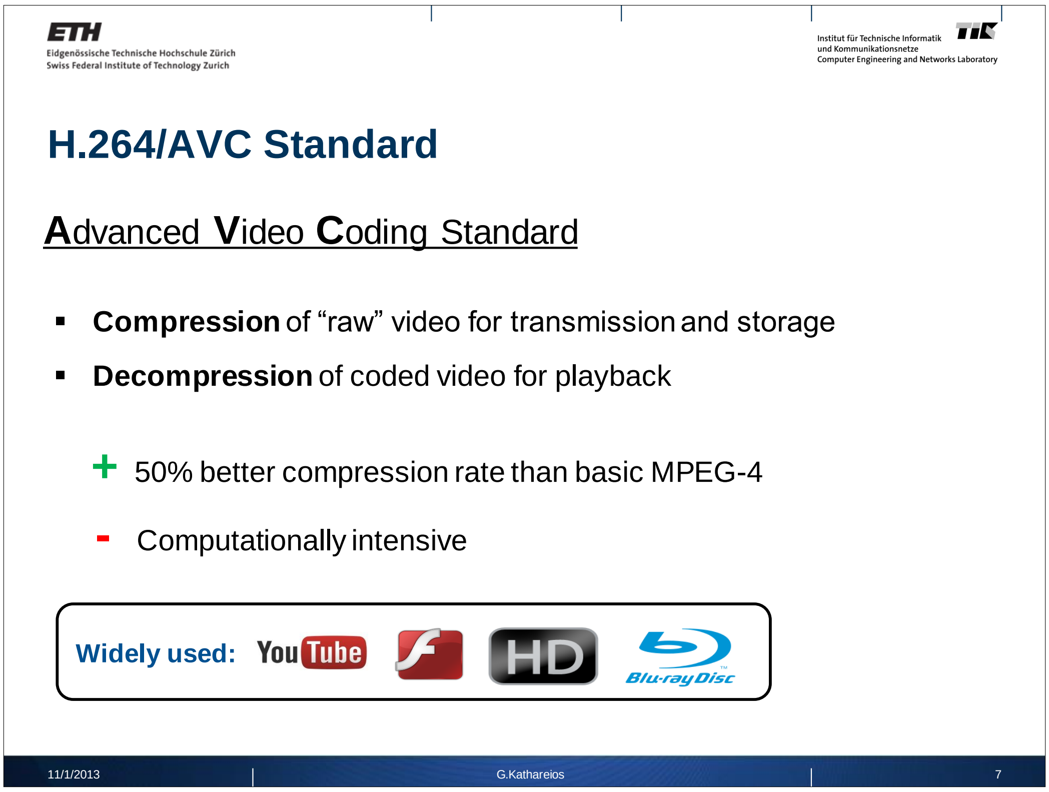


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

DAL Framework

11/1/2013 | G.Kathareios | 6



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

H.264/AVC Standard

<u>**A**dvanced **V**ideo **C**oding Standard</u>

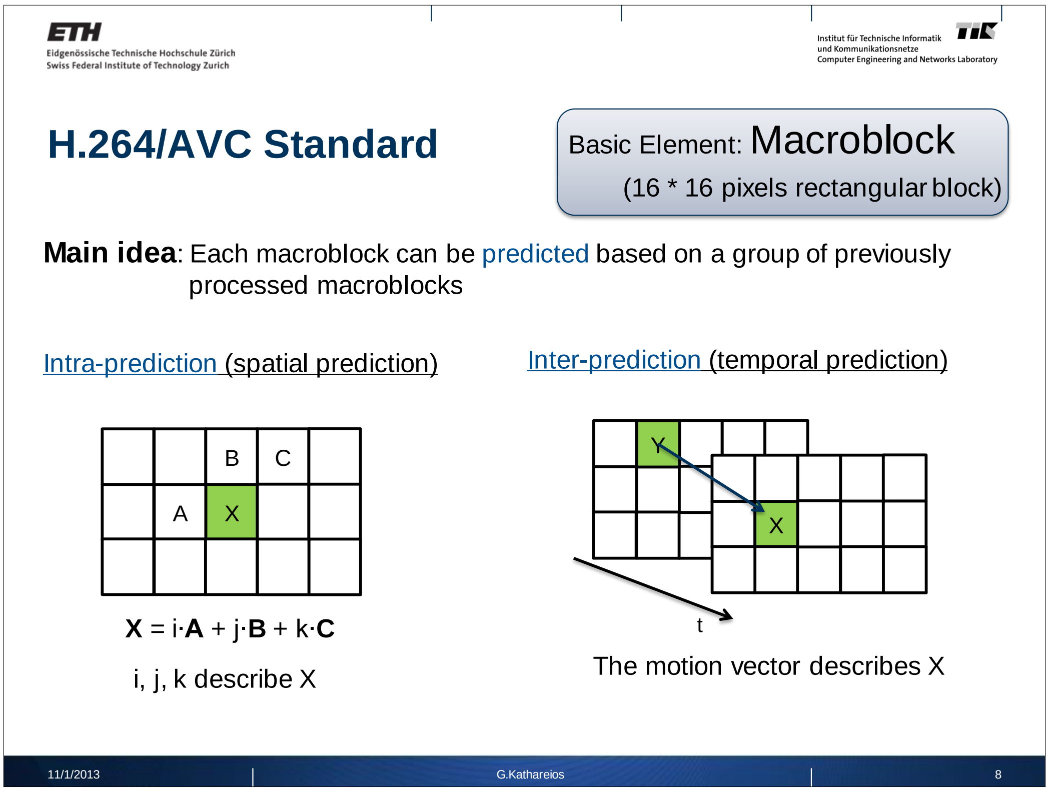
- **Compression** of "raw" video for transmission and storage
- **Decompression** of coded video for playback

+ 50% better compression rate than basic MPEG-4

- Computationally intensive

Widely used: YouTube HD Blu-ray Disc

11/1/2013 | G.Kathareios | 7



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

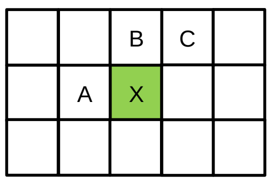
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

H.264/AVC Standard

Basic Element: Macroblock
(16 * 16 pixels rectangular block)

Main idea: Each macroblock can be predicted based on a group of previously processed macroblocks

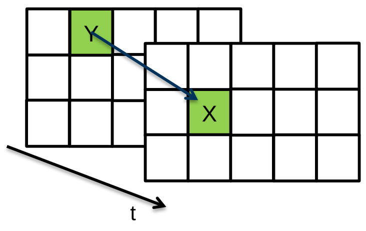
<u>Intra-prediction (spatial prediction)</u>



X = i·**A** + j·**B** + k·**C**

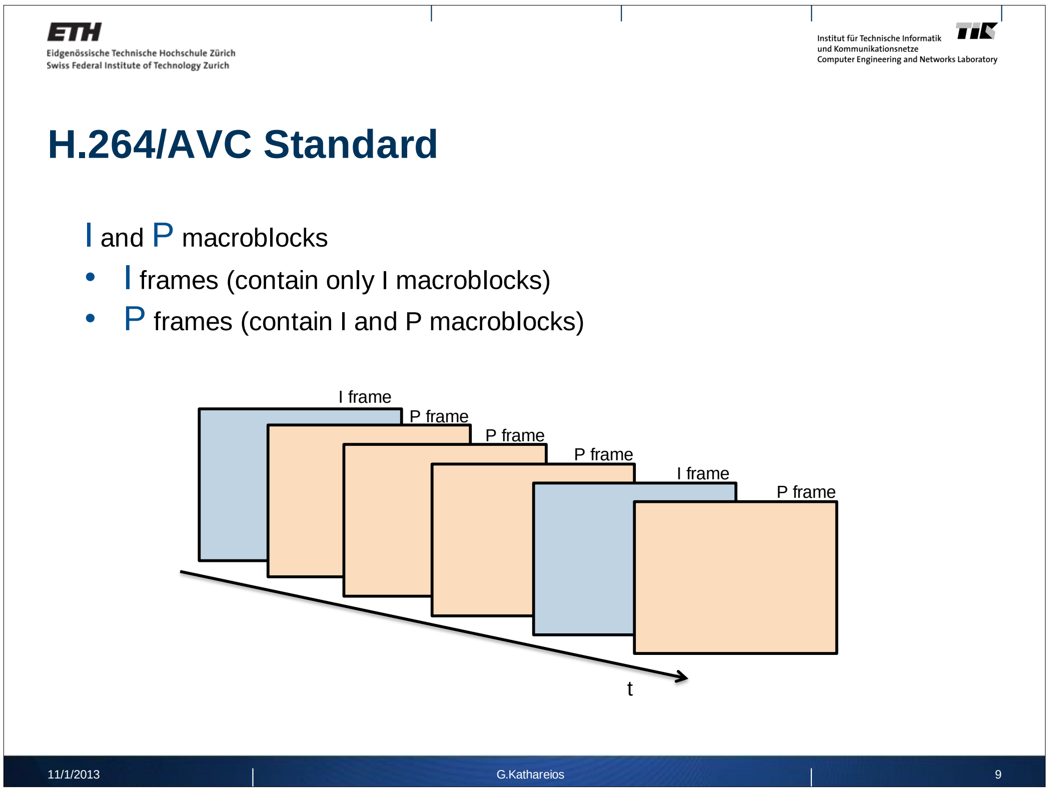
i, j, k describe X

<u>Inter-prediction (temporal prediction)</u>



The motion vector describes X

11/1/2013 | G.Kathareios | 8



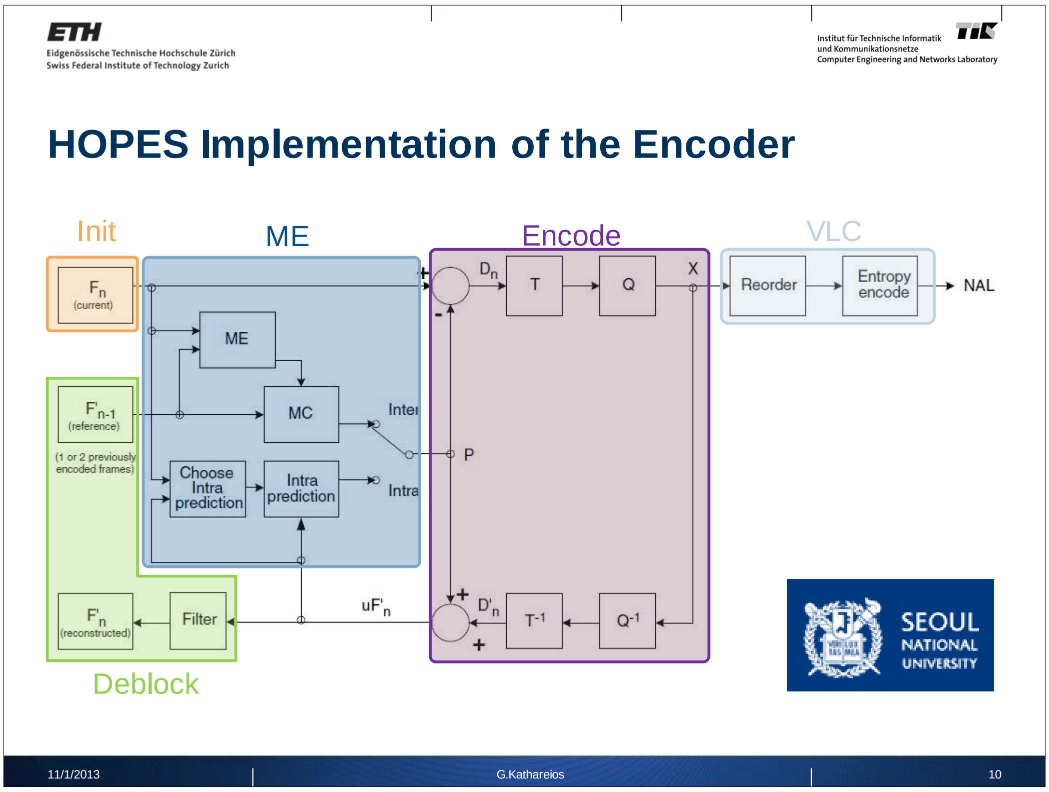
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

H.264/AVC Standard

I and P macroblocks
- I frames (contain only I macroblocks)
- P frames (contain I and P macroblocks)

11/1/2013 | G.Kathareios | 9

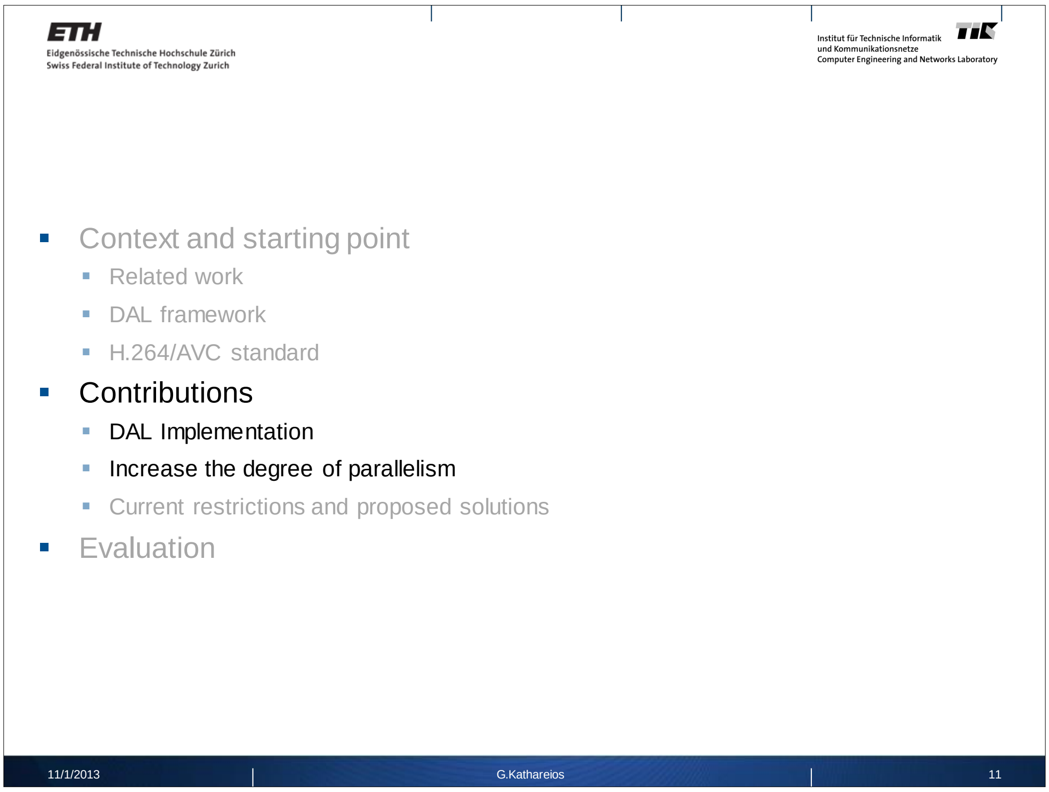


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

HOPES Implementation of the Encoder

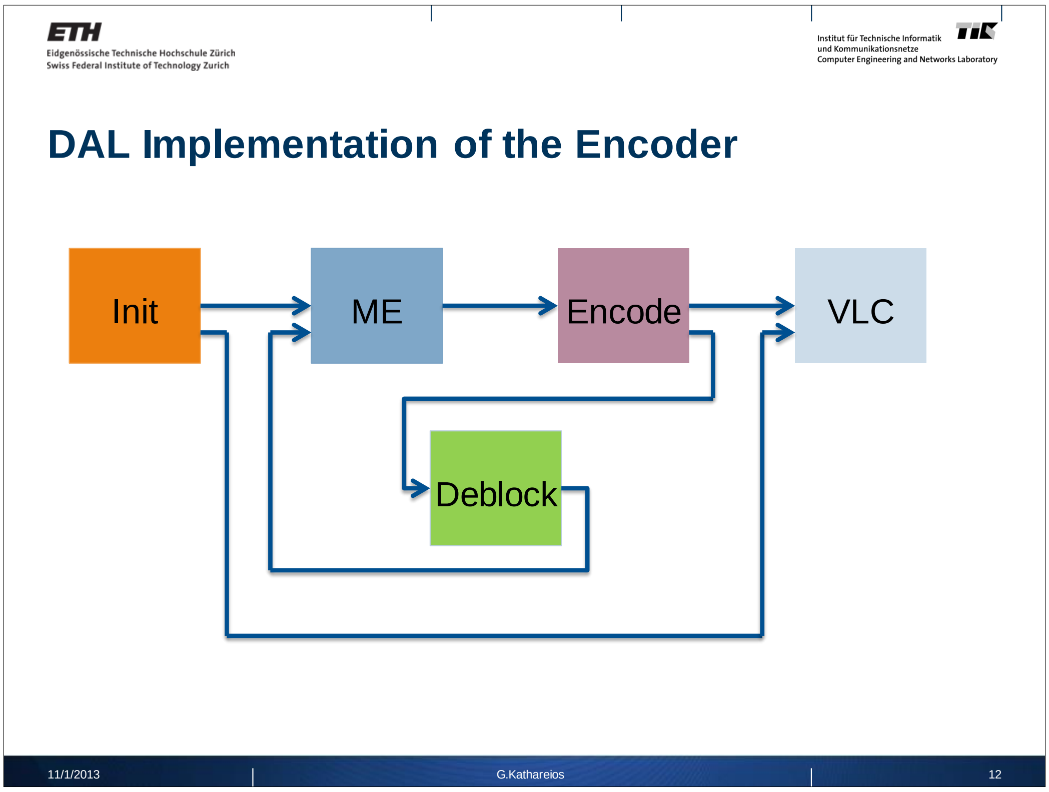
11/1/2013 | G.Kathareios | 10


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

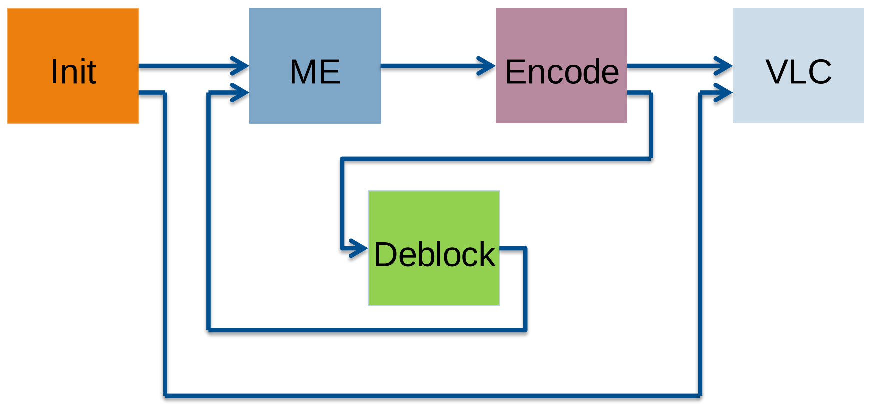
- Context and starting point
  - Related work
  - DAL framework
  - H.264/AVC standard
- Contributions
  - DAL Implementation
  - Increase the degree of parallelism
  - Current restrictions and proposed solutions
- Evaluation

11/1/2013 | G.Kathareios | 11

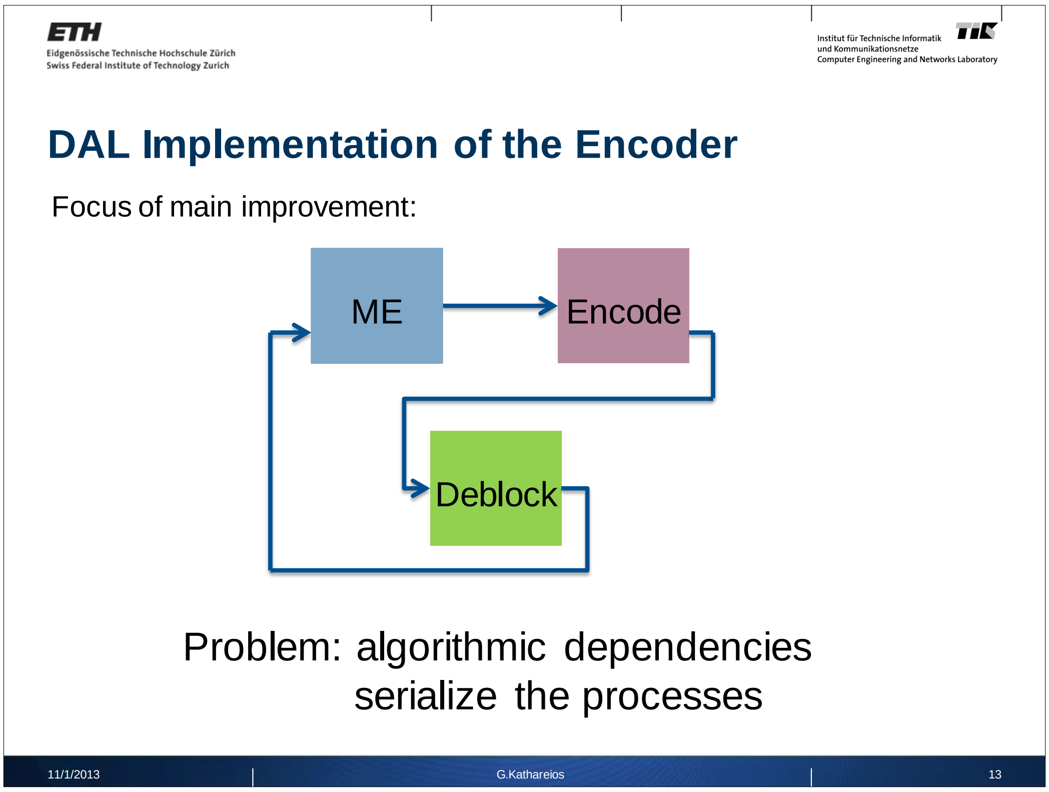

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

DAL Implementation of the Encoder



11/1/2013 | G.Kathareios | 12



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

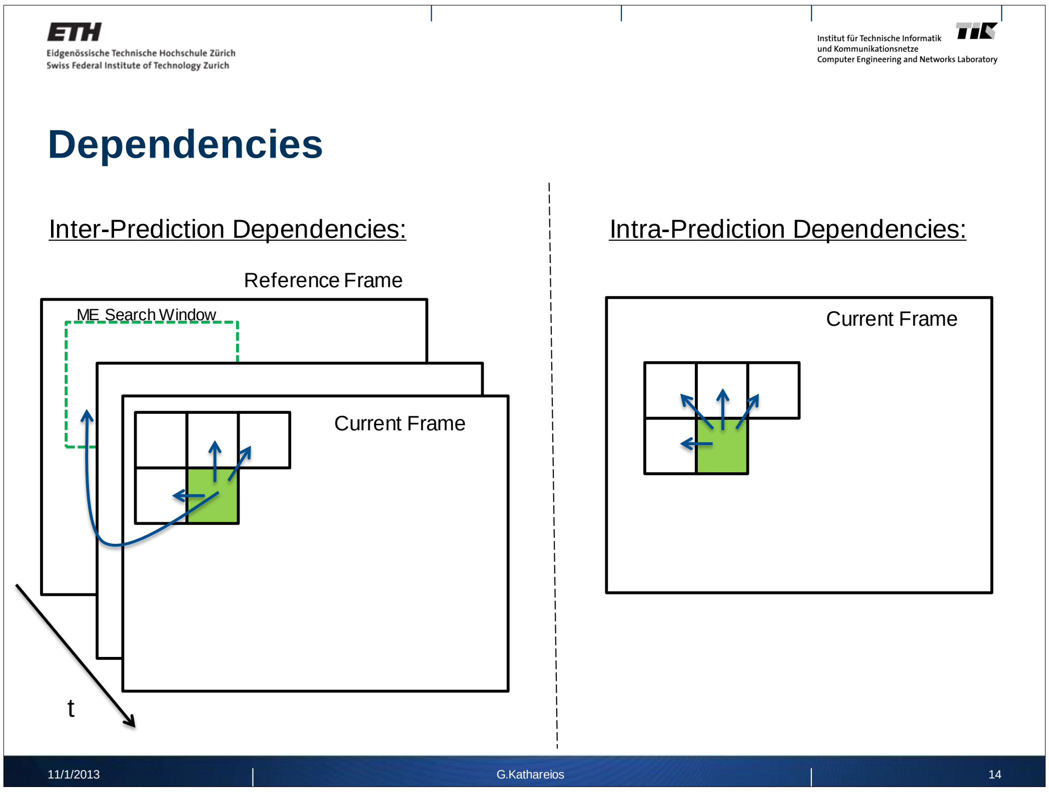
DAL Implementation of the Encoder

Focus of main improvement:

ME
Encode
Deblock

Problem: algorithmic dependencies serialize the processes

11/1/2013 | G.Kathareios | 13



ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

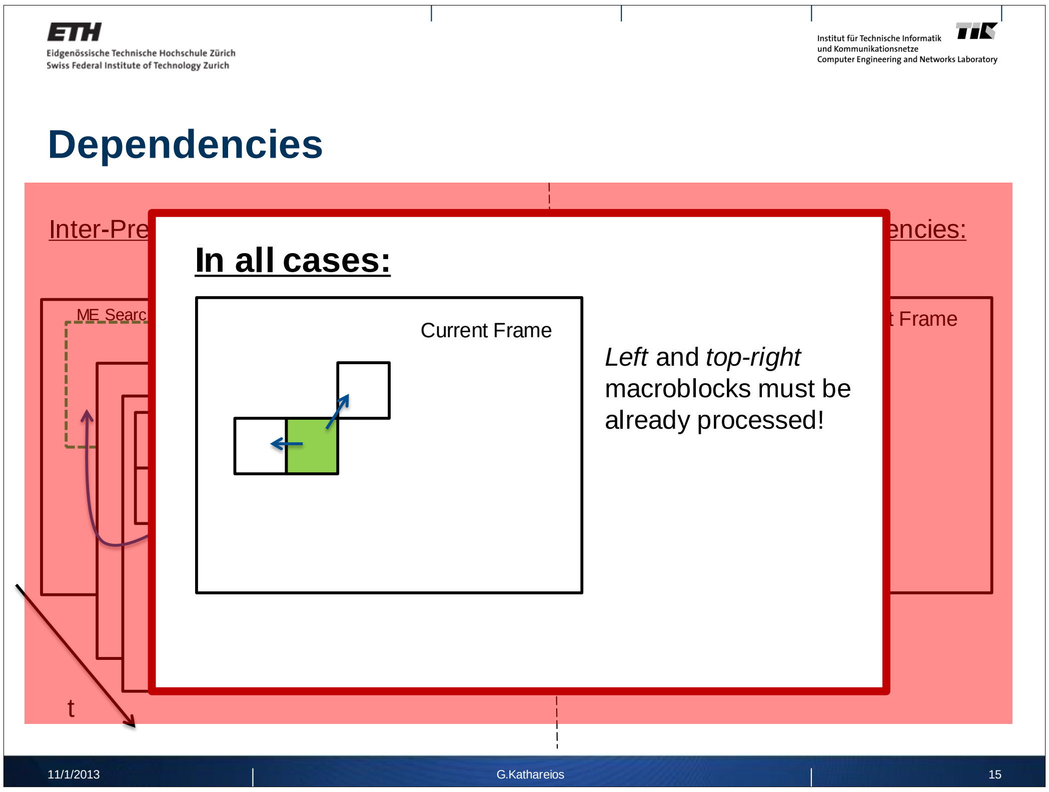
Dependencies

Inter-Prediction Dependencies:

Intra-Prediction Dependencies:

Reference Frame
ME Search Window
Current Frame
t
Current Frame

11/1/2013 | G.Kathareios | 14



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Dependencies

Inter-Pre

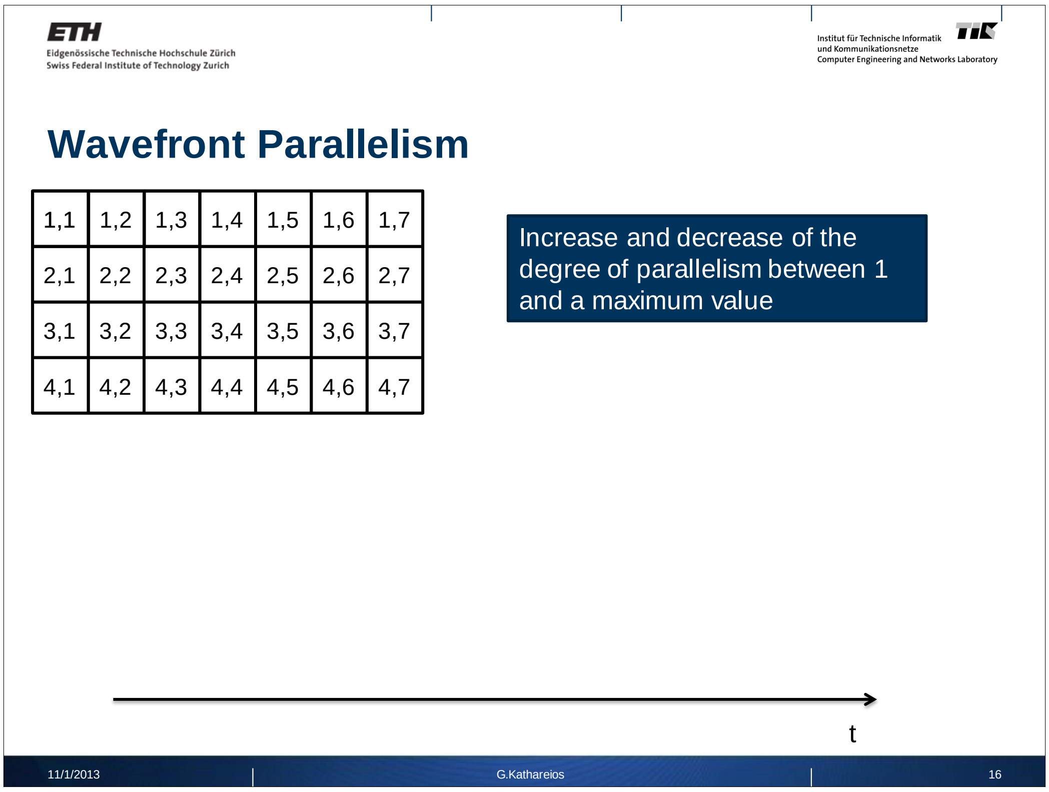
encies:

In all cases:

Left and *top-right* macroblocks must be already processed!

ME Searc
Current Frame
t Frame
t

11/1/2013 | G.Kathareios | 15



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

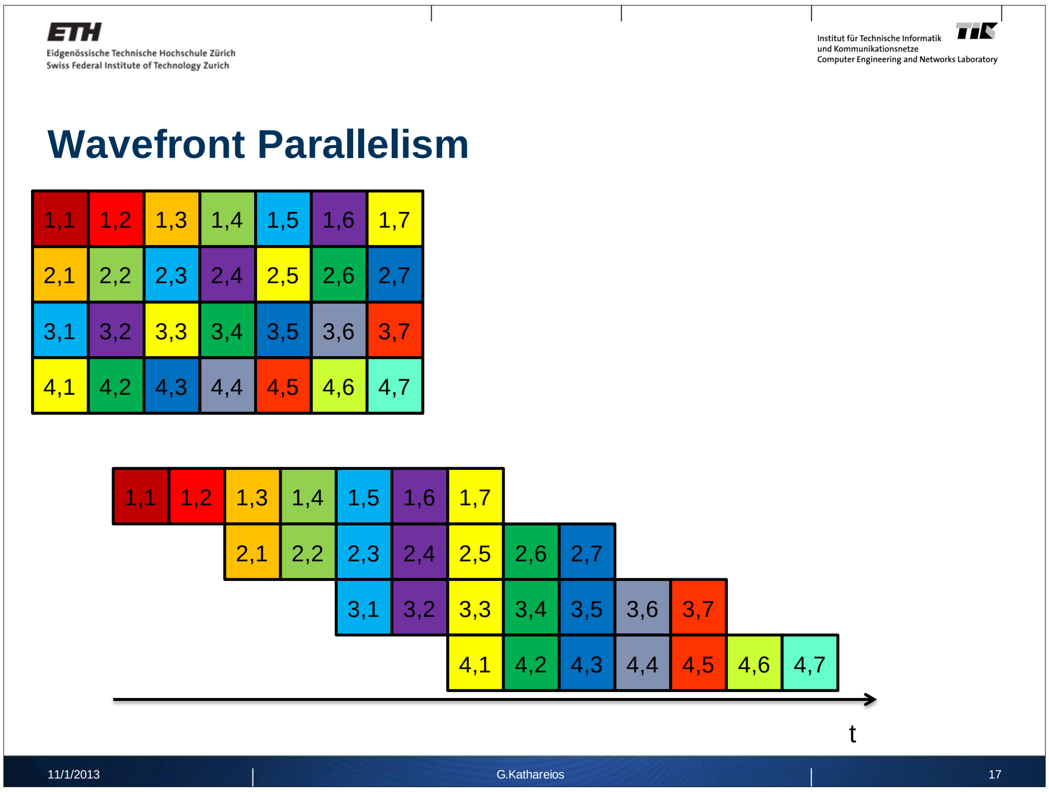
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Wavefront Parallelism

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7

Increase and decrease of the degree of parallelism between 1 and a maximum value

11/1/2013 | G.Kathareios | 16

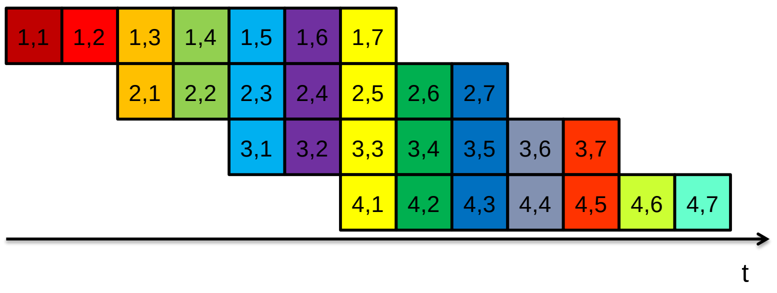


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

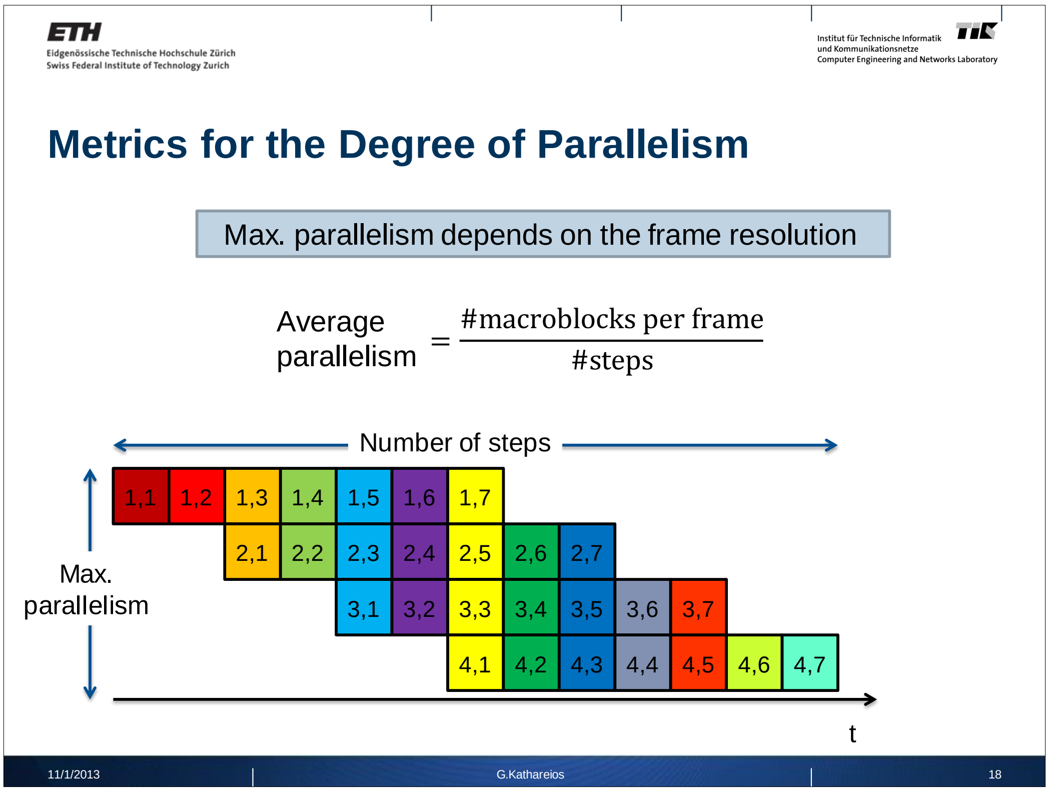
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Wavefront Parallelism

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7



11/1/2013 | G.Kathareios | 17



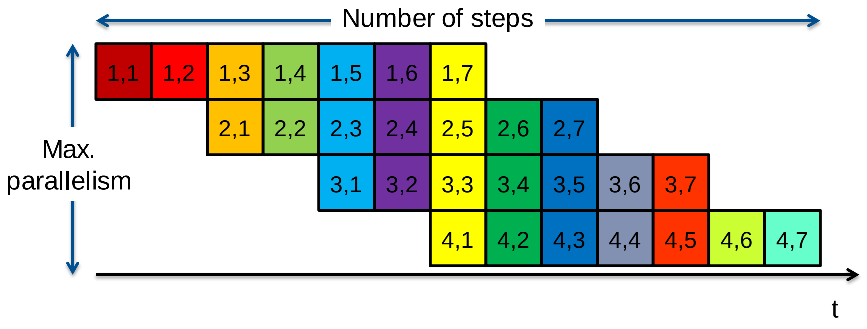
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

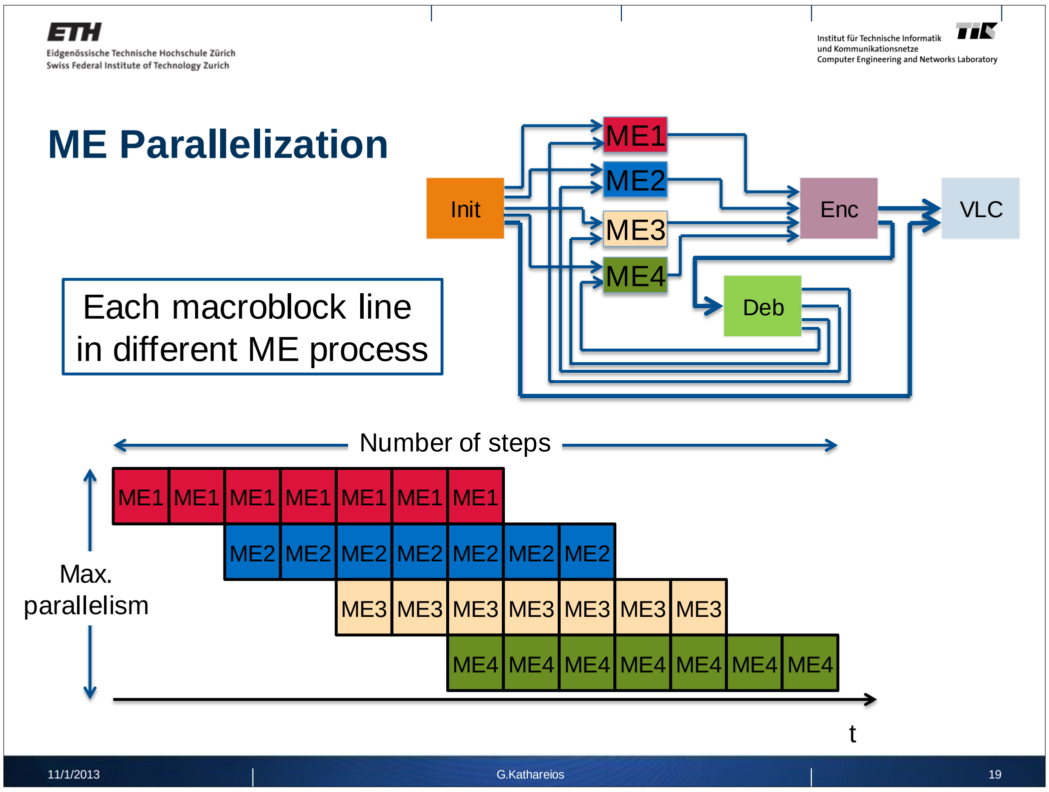
Metrics for the Degree of Parallelism

Max. parallelism depends on the frame resolution

$$\text{Average parallelism} = \frac{\#\text{macroblocks per frame}}{\#\text{steps}}$$



11/1/2013 | G.Kathareios | 18



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

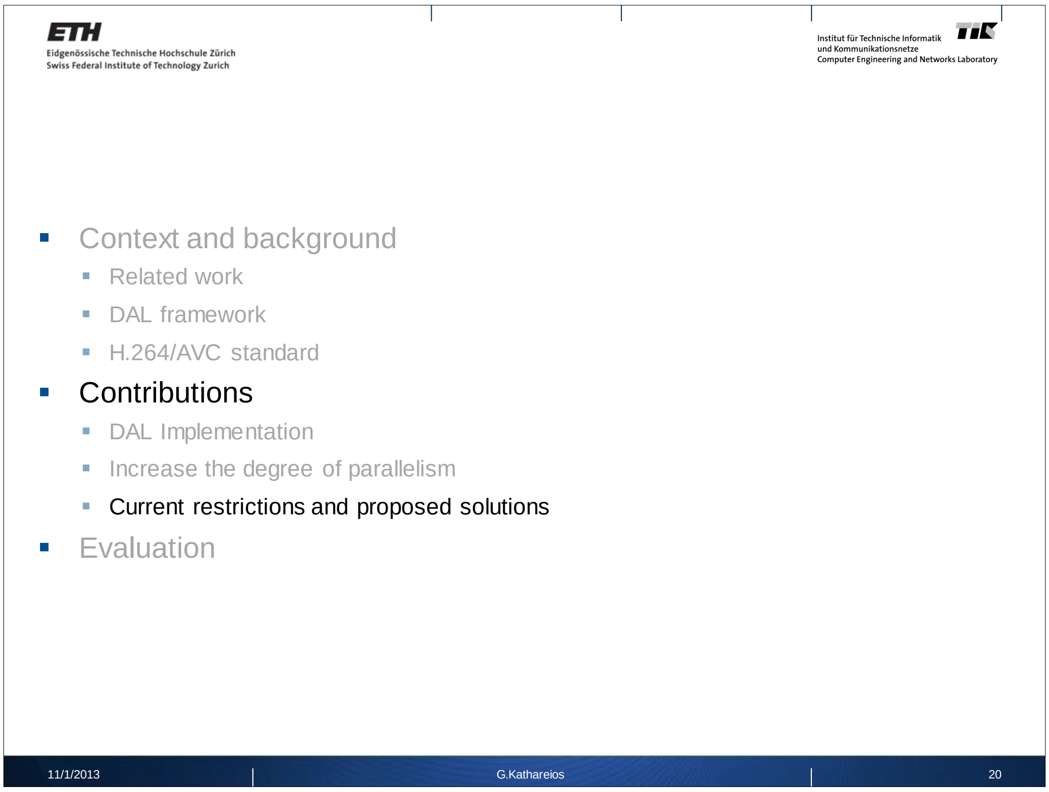
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

ME Parallelization

Each macroblock line in different ME process

Init
ME1
ME2
ME3
ME4
Enc
VLC
Deb
Number of steps
ME1 ME1 ME1 ME1 ME1 ME1 ME1
ME2 ME2 ME2 ME2 ME2 ME2 ME2
ME3 ME3 ME3 ME3 ME3 ME3 ME3
ME4 ME4 ME4 ME4 ME4 ME4 ME4
Max. parallelism
t

11/1/2013 | G.Kathareios | 19

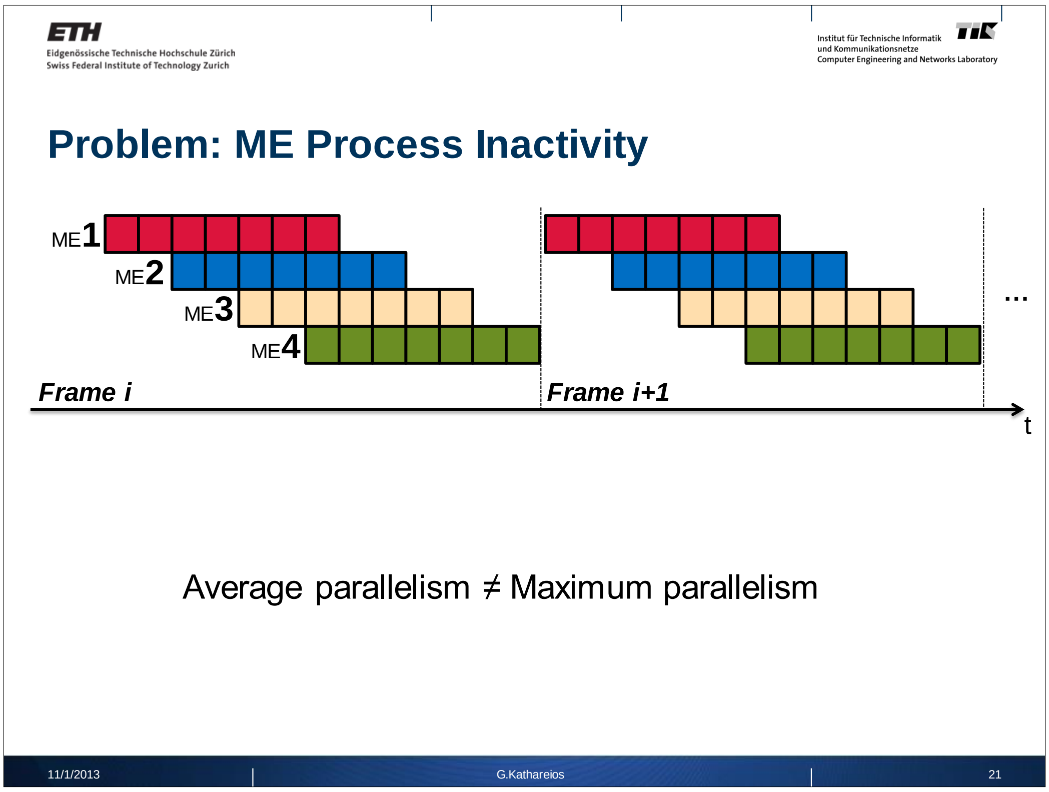


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

- Context and background
  - Related work
  - DAL framework
  - H.264/AVC standard
- Contributions
  - DAL Implementation
  - Increase the degree of parallelism
  - Current restrictions and proposed solutions
- Evaluation

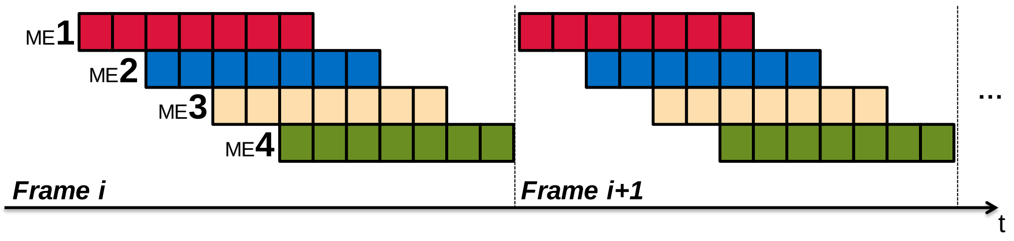
11/1/2013 | G.Kathareios | 20



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

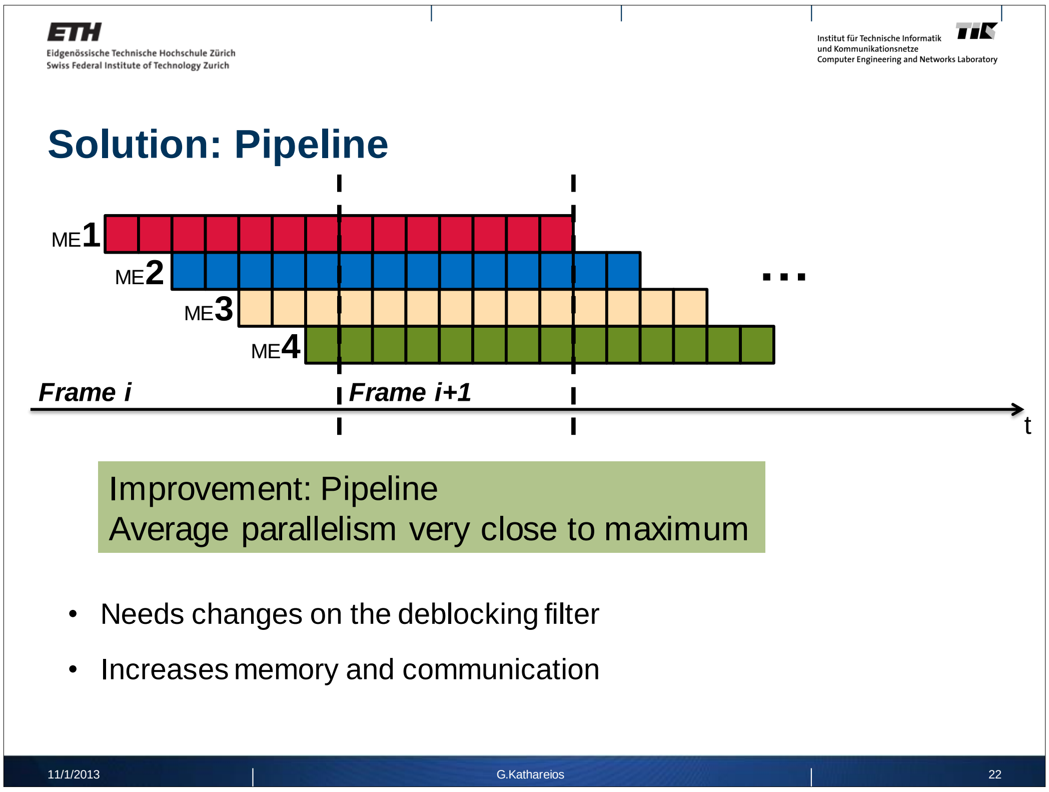
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Problem: ME Process Inactivity



Average parallelism ≠ Maximum parallelism

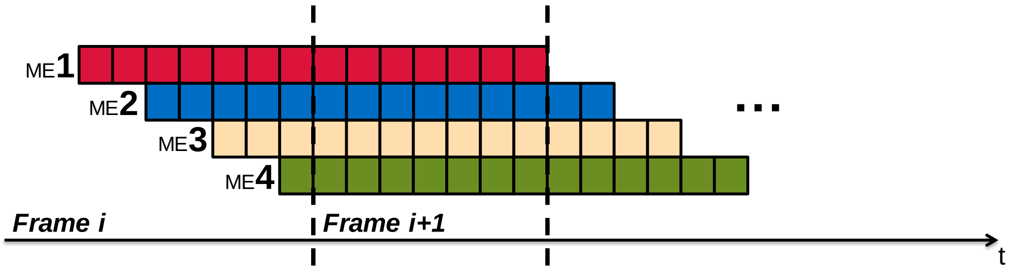
11/1/2013 G.Kathareios 21



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

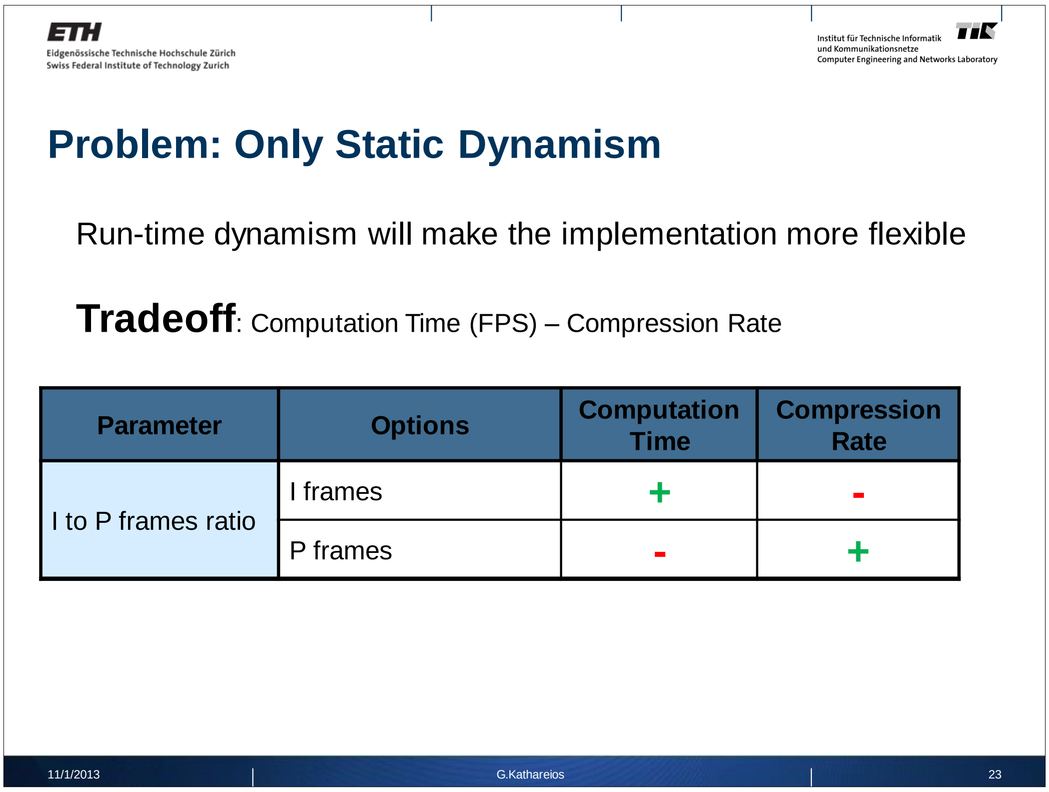
Solution: Pipeline



Improvement: Pipeline
Average parallelism very close to maximum

- Needs changes on the deblocking filter
- Increases memory and communication

11/1/2013 G.Kathareios 22



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

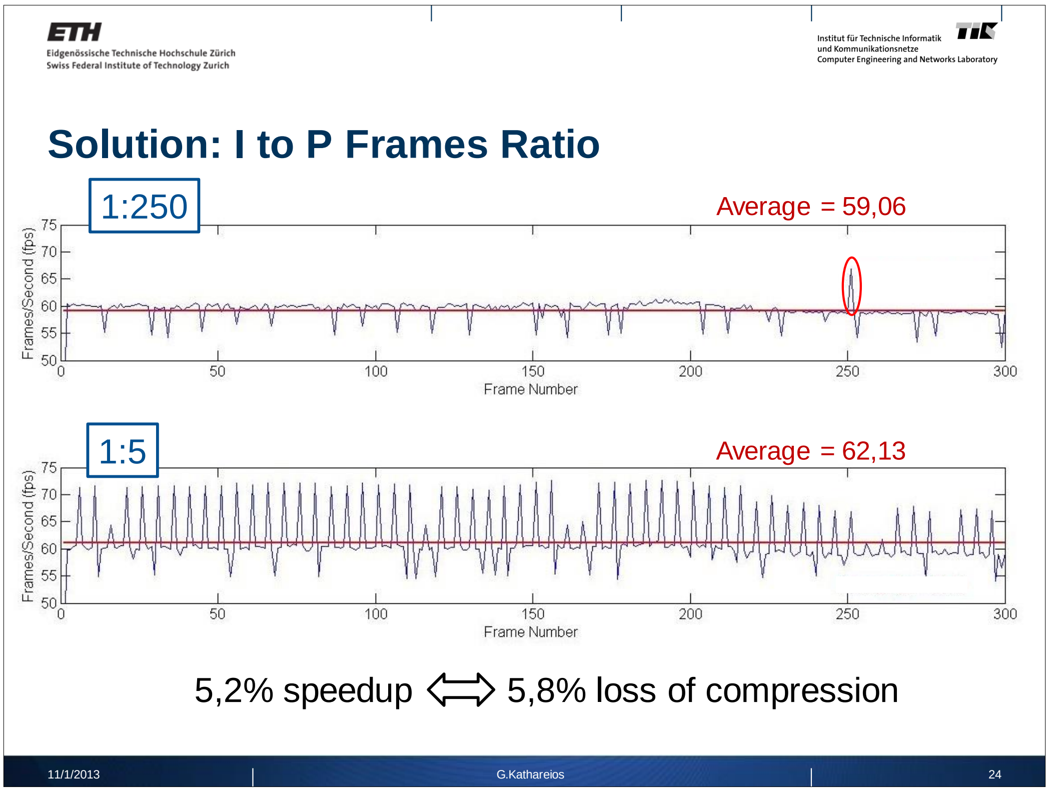
Problem: Only Static Dynamism

Run-time dynamism will make the implementation more flexible

Tradeoff: Computation Time (FPS) – Compression Rate

Parameter	Options	Computation Time	Compression Rate
I to P frames ratio	I frames	+	-
	P frames	-	+

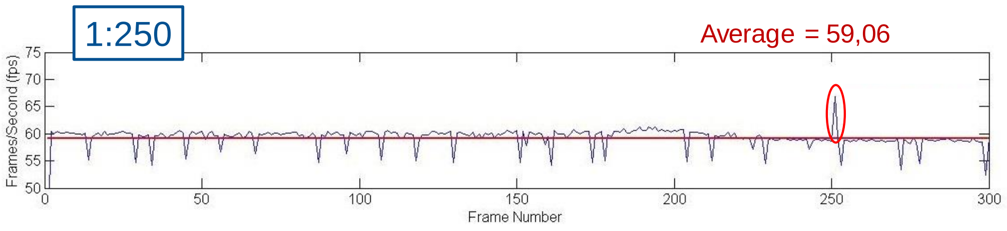
11/1/2013 | G.Kathareios | 23

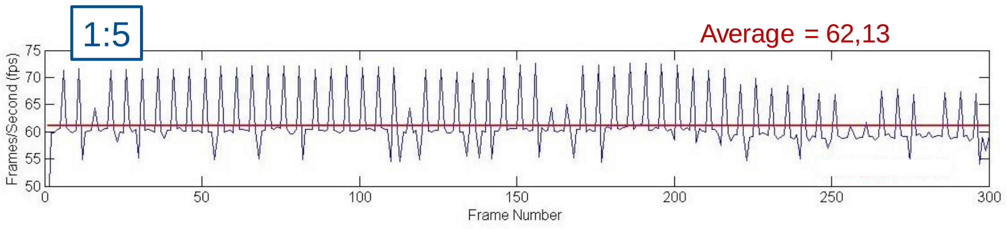


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

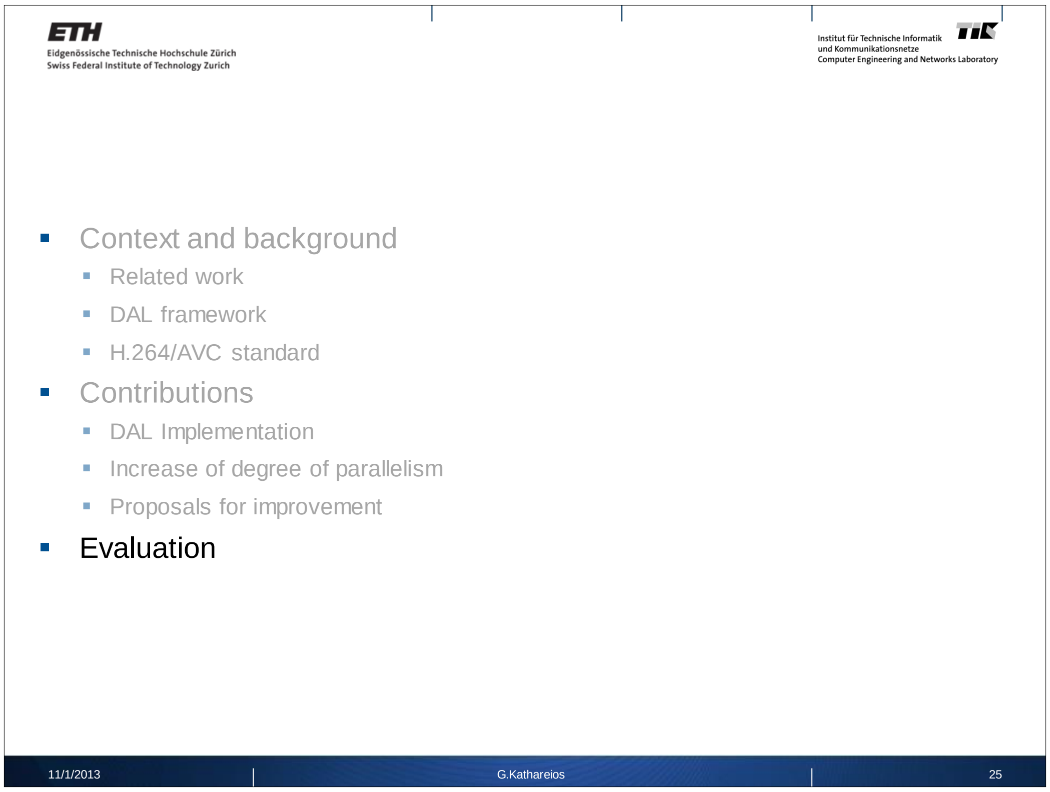
Solution: I to P Frames Ratio





5,2% speedup ⟺ 5,8% loss of compression

11/1/2013 | G.Kathareios | 24

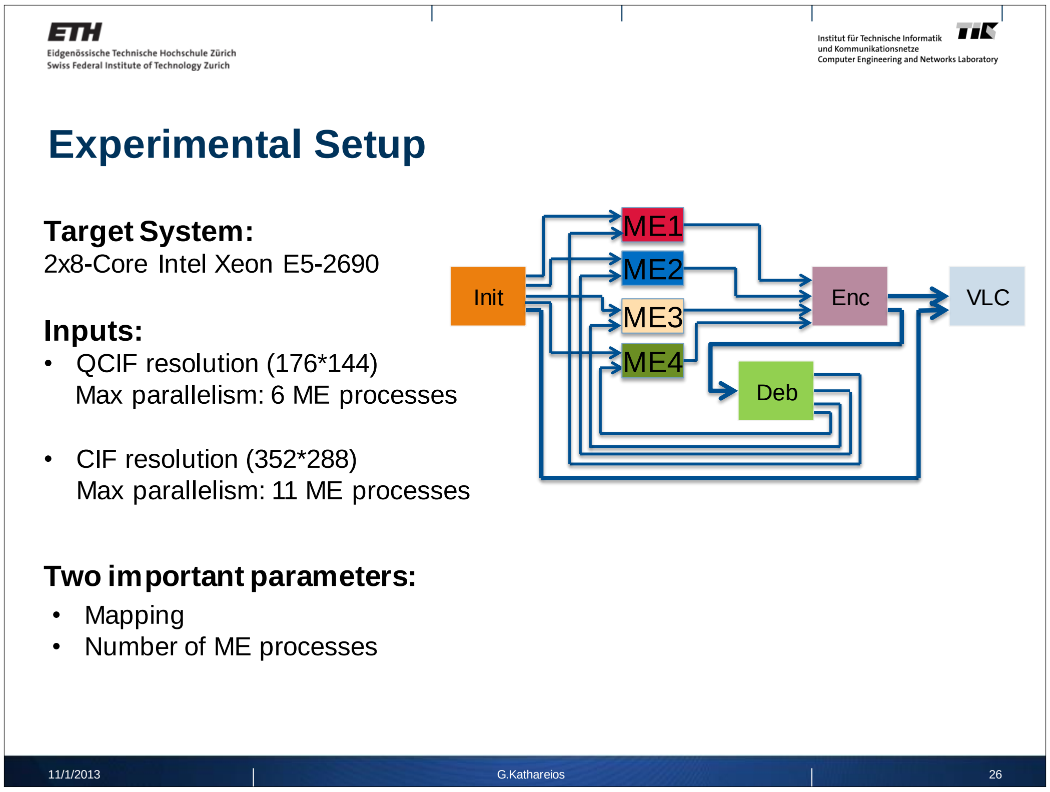


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

- Context and background
  - Related work
  - DAL framework
  - H.264/AVC standard
- Contributions
  - DAL Implementation
  - Increase of degree of parallelism
  - Proposals for improvement
- Evaluation

11/1/2013 | G.Kathareios | 25



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

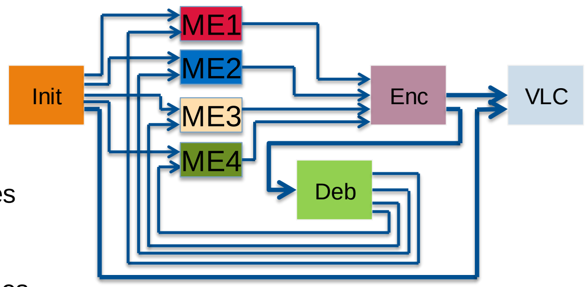
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Experimental Setup

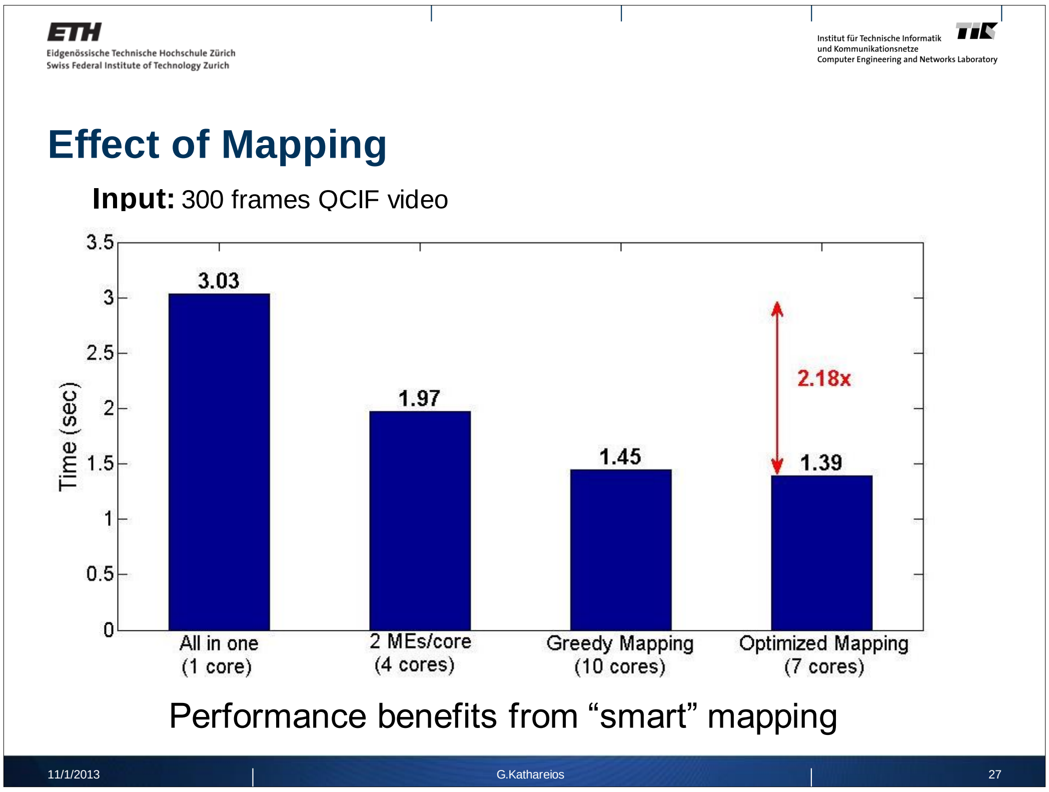
Target System:
2x8-Core Intel Xeon E5-2690

Inputs:
- QCIF resolution (176*144)
  Max parallelism: 6 ME processes
- CIF resolution (352*288)
  Max parallelism: 11 ME processes

Two important parameters:
- Mapping
- Number of ME processes



11/1/2013 | G.Kathareios | 26

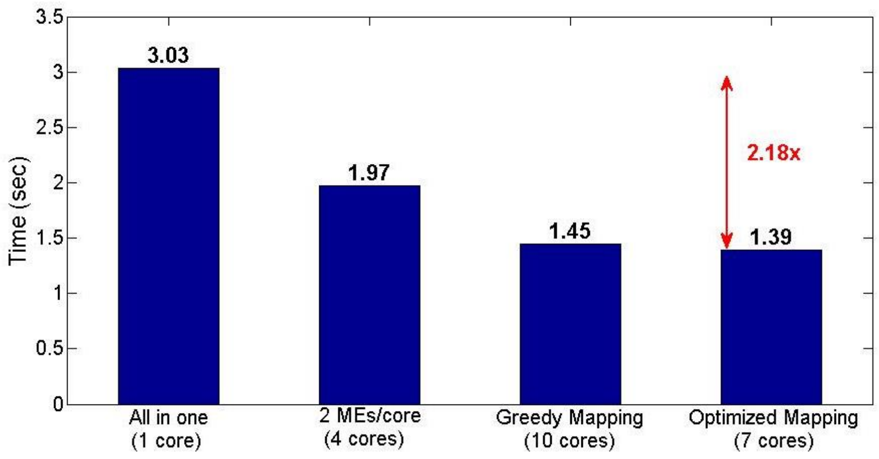


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

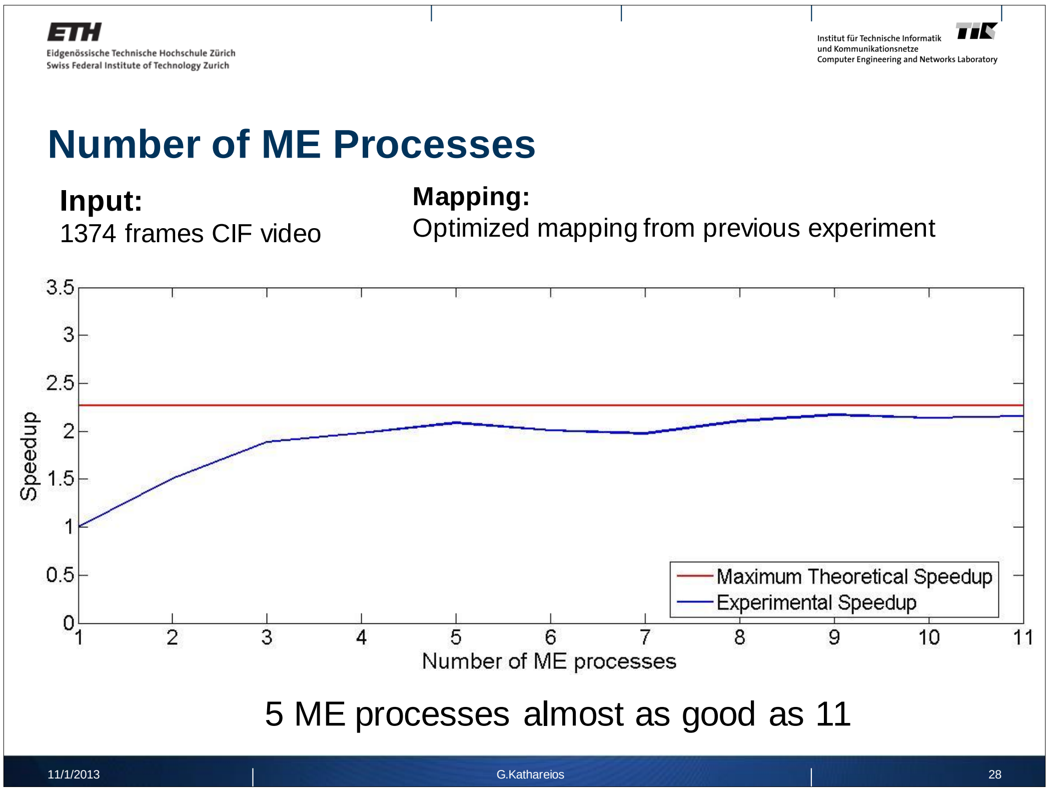
Effect of Mapping

Input: 300 frames QCIF video



Performance benefits from “smart” mapping

11/1/2013 | G.Kathareios | 27



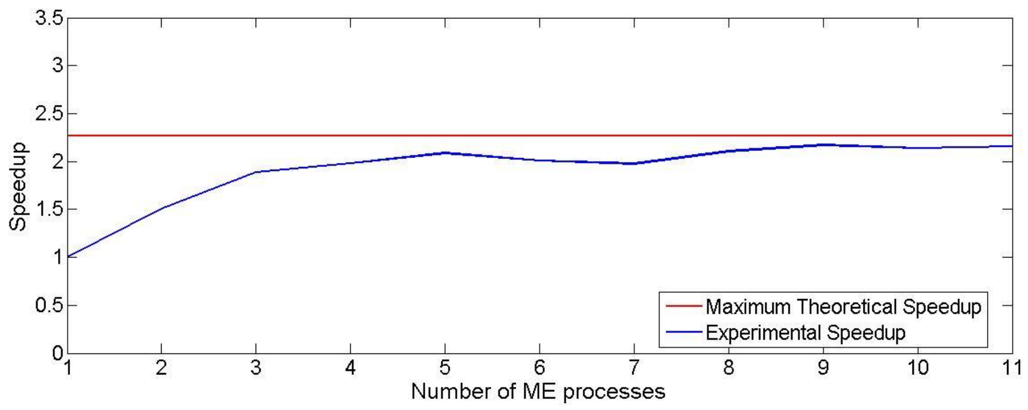
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Number of ME Processes

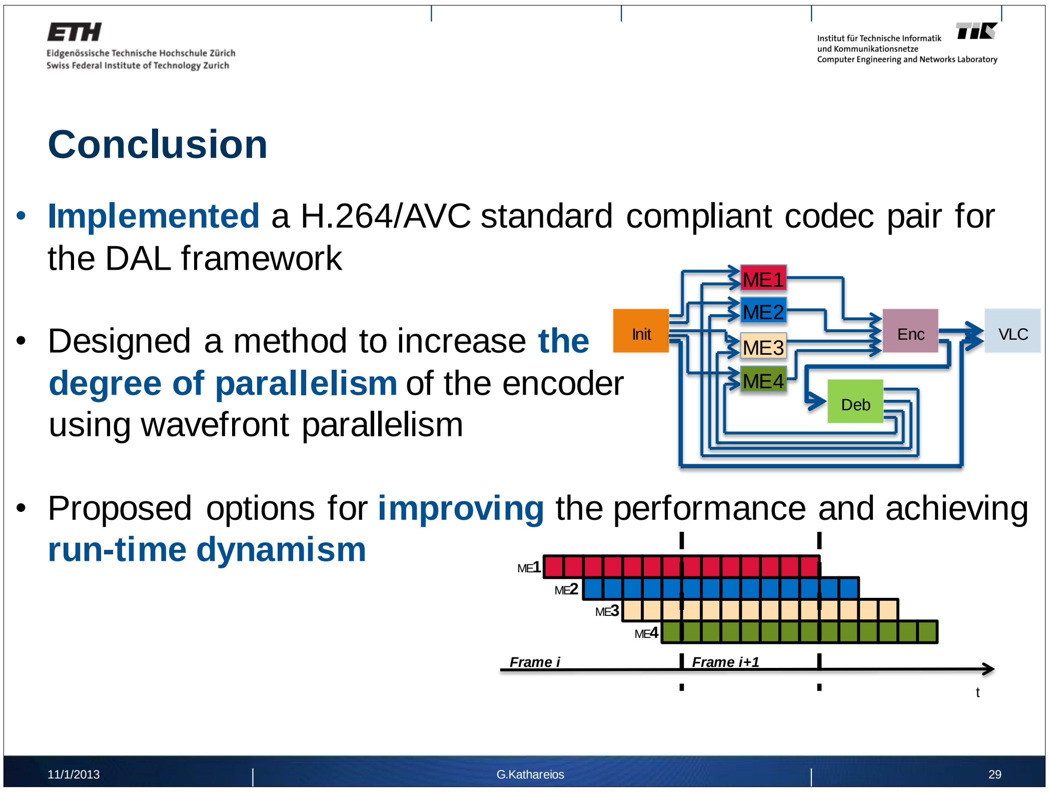
Input:
1374 frames CIF video

Mapping:
Optimized mapping from previous experiment



5 ME processes almost as good as 11

11/1/2013 | G.Kathareios | 28

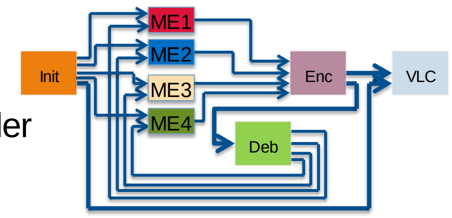


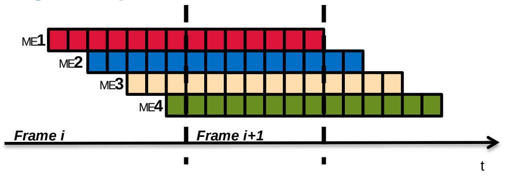
ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Conclusion

- **Implemented** a H.264/AVC standard compliant codec pair for the DAL framework
- Designed a method to increase **the degree of parallelism** of the encoder using wavefront parallelism
- Proposed options for **improving** the performance and achieving **run-time dynamism**





11/1/2013 | G.Kathareios | 29

Bibliography

[1] x264. `http://www.videolan.org/developers/x264.html`.

[2] ISO/IEC 14496-10 and ITU-T Recommendation. H.264 : Advanced Video Coding for Generic Audiovisual Services. 2003.

[3] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[4] Shin-Haeng Kang, Hoeseok Yang, Lars Schor, Iuliana Bacivarov, Soonhoi Ha, and Lothar Thiele. Multi-objective mapping optimization via problem decomposition for many-core systems. In *Proc. IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 28–37, Tampere, Finland, Oct 2012. IEEE.

[5] Youngsub Ko, Youngmin Yi, and Soonhoi Ha. An Efficient Parallelization Technique for x264 Encoder on Heterogeneous Platforms Consisting of CPUs and GPUs. *Journal of Real-Time Image Processing*, pages 1–14, 2013.

[6] Seongnam Kwon, Yongjoo Kim, Woo-Chul Jeun, Soonhoi Ha, and Yunheung Paek. A Retargetable Parallel-Programming Framework for MPSoC. *ACM Trans. Design Autom. Electr. Syst.*, 13, 2008.

[7] Hae-woo Park Kyunghyun Kim, Jaewon Lee and Soonhoi Ha. Automatic H.264 Encoder Synthesis for the Cell processor from a Target Independent Specification. In *ESTImedia*, pages 95–100, 2008.

[8] P. List, A. Joch, J. Lainema, G. Bjontegaard, and M. Karczewicz. Adaptive deblocking filter. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):614–619, 2003.

[9] Wu Nan, Mei Wen, Huayou Su, Ju Ren, and Chunyuan Zhang. A Parallel H.264 Encoder with CUDA: Mapping and Evaluation. In *ICPADS*, pages 276–283. IEEE Computer Society, 2012.

[10] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi. Video Coding with

H.264/AVC: Tools, Performance, and Complexity. *Circuits and Systems Magazine, IEEE*, 4:7–28, 2004.

[11] Jonghan Park and Soonhoi Ha. Performance Analysis of Parallel Execution of H.264 Encoder on the Cell Processor. In *ESTImedia*, pages 27–32. IEEE, 2007.

[12] Pier Stanislao Paolucci, Iuliana Bacivarov, Gert Goossens, Rainer Leupers, Frédéric Rousseau, Christoph Schumacher, Lothar Thiele and Piero Vicini. EURETILE 2010-2012 Summary: First Three Years of Activity of the European Reference Tiled Experiment. *CoRR*, abs/1305.1459, 2013.

[13] I.E. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia.* Wiley, 2003.

[14] A. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical Parallelization of an H.264/AVC Video Encoder. In *Proceedings of the international symposium on Parallel Computing in Electrical Engineering*, PARELEC '06, pages 363–368, 2006.

[15] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems. In *CASES*, pages 71–80. ACM, 2012.

[16] Lars Schor, Hoeseok Yang, Iuliana Bacivarov, and Lothar Thiele. Expandable process networks to efficiently specify and explore task, data, and pipeline parallelism. In *Proc. International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES)*, Montreal, Canada, Oct 2013. IEEE.

[17] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13:560–576, 2003.

[18] Hajer Krichene Zrida, Abderrazek Jemai, Ahmed C. Ammari, and Mohamed Abid. High level H.264/AVC Video Encoder Parallelization for Multiprocessor Implementation. In *DATE*, pages 940–945. IEEE, 2009.