PARALLEL AND SCALABLE VIDEO
DECODING FOR DISTRIBUTED
TRANSCODING APPLICATIONS

GEORGIOS GEORGAKARAKOS

MSc Thesis, Embedded Computing
Supervisors: Juha Plosila, Masoud Daneshtalab
Embedded Computer Systems Laboratory
University of Turku
Advisor: Sébastien Lafond
Embedded Systems Laboratory
Åbo Akademi University
November 22, 2013

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.
Abstract

Nowadays with the growing increase in video content over the Web, video processing and manipulation becomes an increasingly challenging process; transfer, distribution and sharing of video content is possible only if efficient transcoding solutions are introduced. Video transcoding is a wide area of research. Many initiatives are focusing on how to optimize and enhance transcoding operation by reducing redundancy. Optimization efforts often focus on the decoding part of the operation so the redundant or optimized functionality is removed from the encoding part. With the advent of multi-processor architectures the interest had shifted in efficient partitioning and parallelization of the decoding algorithm. Parallel H.264 video decoding has been already implemented in various forms. However little work has been done regarding a thorough exploration and evaluation of parallelization capabilities for performance improvements especially in multi-processing element architectures. In this thesis the focus is to explore, implement, evaluate and propose an efficient parallelization technique for H.264 video decoding in order to apply it in scalable video transcoding applications for environments with many processing elements (PEs).
## Contents

1 Introduction .................................................. 1
   1.1 Problem overview .......................................... 2
   1.2 Thesis objectives .......................................... 3
   1.3 Thesis structure .......................................... 4

2 Parallel video decoding design exploration ............. 5
   2.1 Decoding algorithm dependencies .......................... 5
   2.2 Parallelization types ......................................... 8
      2.2.1 Granularity ........................................... 11
      2.2.2 MB Parallelization techniques ......................... 15

3 Parallel video decoding architecture .................... 18
   3.1 Concept .................................................. 18
   3.2 Scheduling ................................................ 21

4 Parallel Video Decoding implementation ............. 30
   4.1 Cofluent Studio ........................................... 30
   4.2 Parallel H.264 decoding model ............................. 31

5 Experiments and results .................................. 36
   5.1 Input video streams set-up ................................ 36
   5.2 Decoding model set-up ................................... 39
   5.3 Measurements .............................................. 39

6 Conclusions .................................................. 50
6.1 Performance .......................................................... 50
6.2 Optimal allocation in multicore architecture .................. 57

7 Future work .......................................................... 62
## List of Figures

2.1 H.264 decoder .................................................. 6  
2.2 Intra prediction and de-blocking filter dependencies ........... 6  
2.3 Intra 4x4 modes .................................................. 7  
2.4 Motion estimation dependencies .................................. 8  
2.5 H.264 decoder task based parallelization ...................... 9  
2.6 Data based parallelization concept ............................... 10  
2.7 H.264 stream decomposition ...................................... 11  
2.8 Frame types in a generic H.264 stream .......................... 12  
2.9 H.264 data granularity .......................................... 12  
2.10 MB parallel decode concept ..................................... 14  
2.11 Parallel MB decoding ........................................... 15  
2.12 MB grouping for parallel decoding ............................... 15  
2.13 '3D' MB parallelization concept ................................. 17  

3.1 Parallel H.264 decode architecture .............................. 19  
3.2 Expanded H.264 parallel model .................................. 20  
3.3 H.264 decode DAG ................................................. 21  
3.4 H.264 task graph .................................................. 22  
3.5 Generic MB dependencies ........................................ 24  
3.6 Dynamic scheduling architecture .................................. 24  
3.7 H.264 graph for intra 16x16 MBs ................................. 25  
3.8 EIA Scheduler ..................................................... 27  

4.1 Parallel H.264 decoding model ..................................... 32
List of Tables

3.1 MB encoding info for scheduler input .................................. 26
5.1 Encoded input stream parameters ....................................... 38
5.2 Input H.264 video sequences ............................................ 39
5.3 Results news cif sequence .................................................. 41
5.4 Results asahi vga sequence ................................................ 41
5.5 Results madagascar 720p sequence ..................................... 42
5.6 Results pedestrian full HD sequence ................................... 42
5.7 Results foreman 4K sequence ............................................. 43
6.1 Experimental vs theoretical 2D speed-up ............................... 52
6.2 Experimental vs theoretical 2D worker count .......................... 54
6.3 Total workers used for parallel decoding ............................... 55
Definitions

**MPEG**: Moving Picture Expert Group is an experts group responsible for standardization of video compression.

**Encoding**: Is the process used to convert information from one representation to another that requires less amount of data.

**Decoding**: Is the process used to convert encoded information to its original representation.

**GOP**: Group of Pictures is an autonomous group of successive pictures inside a coded video stream.

**MB**: Macroblock is the basic unit in image/video processing, comprising a set of 16x16 luminance and 2 8x8 chrominance pixels.

**IDR**: Instantaneous Decoding Refresh is special I-type frame in a video sequence signaling that future frames cannot refer to frames prior to IDR frame.

**YUV**: Is a colour space expressing one luminance(Y) and two chrominance (U,V) components. It is used to represent raw uncompressed (or decompressed) video.

**I frame**: Is a frame for which prediction is made only using pixels within the frame itself.

**P frame**: Is a frame for which prediction is made using pixels from preceding frames of the stream.
**B frame**: Is a frame for which prediction is made using pixels from preceding and succeeding frames (in display order) of the video stream.

**Task graph**: is a representation of a problem solution, algorithm or process in a series of independent operations connected together in a way that reflects the dependencies among them.

**Scheduling**: Is the process of deciding how and when to allocate several tasks of a main process in available resources.

**Thread**: Is a general term expressing a (small) group of instructions forming a code that can be executed independently and concurrently with others similar elements.

**FPGA**: Field Programmable Gate Array is a popular electronic circuit device designed to be programmable after fabrication.

**DSP**: Is a processor specially optimised for signal processing applications.

**PE**: Processing Element is a generic term used to reference a hardware component that executes a group of instructions. In modern systems that feature many elements and complex architectures, the context defines what unit of hardware is considered a processing element (computer, CPU, DSP, FPGA, IP core etc).
από τότε που ἔνας ἄνεμος μ’ εμπόδισε, με ὅλους τοὺς ἄνεμους ταξιδέων...

Οδυσσέας Ελύτης
Chapter 1

Introduction

Video transcoding is a technology that gathers growing interest in ICT applications. With the high increase in video content over the Web, video processing and manipulation becomes an increasingly challenging process. The challenges rely mainly on the significant diversity of video content creation from different devices supporting different formats, quality levels, bitrate constraints etc. This diversity makes transfer, distribution and sharing of video content possible only if efficient transcoding solutions are introduced.

Video transcoding can be defined as a process that converts the video signal representation in a different form based on a set of features and attributes [1], [2]. Several different forms of transcoding exist depending on video application needs: from simply changing the video signal’s container headers to converting the video signal’s frame rate, quality factor or resolution up to converting the whole video according to a different encoding algorithm. Regardless of its form, transcoding essentially features a decoding and an encoding stage. In case of a container header change, the process is rather simple. However in other cases the process features computationally intensive signal processing operations. Early transcoding implementations feature direct decoder-encoder cascading. This approach, although simple in realization, had two significant drawbacks: First, continuous decoding and re-encoding caused video quality decrease when using lossy encoding algorithms. Secondly, significant redundancy between decoding and encoding stages was
not exploitable, thus some part of signal processing operations was consuming resources from the system’s time and energy budget without practical reason. Thus, optimization efforts focused on optimization by removing this redundancy between decoding and encoding stages [3]. Examples of this redundancy are the prediction part of the video coding standards which can be present in both encoding and decoding processes. When a video signal is first decoded in a transcoding system the prediction part is already applied, so there is no need to be re-launched in the encoding stage, but the same prediction results can be used. In general, redundant processing varies with the transcoding operation i.e. what attribute of the video signal has to change.

Currently video transcoding is a wide area of research. Many initiatives are focusing on how to optimize and enhance transcoding operation by reducing redundancy. In any transcoding system decoding is applied first to the video signal so the redundant or optimized functionality is removed from the encoding part. This drives the efforts of optimization to the decoding part of the operation.

1.1 Problem overview

Video decoding has been a very popular research area for more than a decade. Video coding standards introduced by early 90s have defined complex signal processing algorithms whose implementation created a whole new range of video-based services and products. However, at the same time it included significant challenges especially when integrated in embedded environment. MPEG video decoding algorithms were implemented in software, either in general purpose CPUs or in special DSPs or even directly in hardware on FPGA or ASIC. With the advent of multi-processor architectures the interest had shifted in efficient partitioning and parallelization of the decoding algorithm in several cores. Parallelizing video decoding algorithm is a challenging process mainly due to the complexity of the algorithms and the tight dependencies present among their data structures. Essentially, efficient parallelization becomes a scheduling problem: the decoding
tasks must be properly grouped and efficiently allocated to multiple processors with respect to specific performance goals including application speed-up, scalability, energy savings, resource utilization etc.

The current state of the art industry standard in video coding is MPEG4 Part 10 or H.264/AVC (Advanced Video Coding) [4]. As a successor of MPEG2 it has leveraged video coding quality and bitrate savings, enabling a set of modern applications to become mainstream including Blue-Ray video storage, real time HD video streaming, IPTV etc.

Parallel H.264 video decoding has been already implemented as presented in [5], [6], [7]. However little work has been done regarding a thorough exploration and evaluation of parallelization capabilities for performance improvements. For example, the scheduling algorithms used for decoding tasks execution are working on worst- case task dependency scenarios thus causing a limit on the possible performance outcome. Additionally, most of existing parallelization approaches are customized for specific multi-processor architectures with fixed number of processors and are not designed for scalability [8], [9], [10]. Thus, current parallel video decoders have a clear inability to exploit emerging multiprocessor architectures that feature a large pool of processing resources.

1.2 Thesis objectives

In this thesis the focus is to explore and propose an efficient parallelization technique for H.264 video decoding in order to apply it in scalable video transcoding applications for environments with many processing elements (PEs). The H.264 decoder’s algorithm is explored in order to identify, organize and group several parallelization possibilities. Then the goal is to propose an efficient scheduling logic that would exploit those capabilities and implement enhanced parallel and scalable H.264 decoding on many-PE environment. Finally, after the evaluation of the results, an architecture featuring many PEs that would implement
an enhanced H.264 parallel decoder can be proposed.

This work is part of the AMEBA project [11]. Among others, AMEBA focuses on the design and exploration of high density 3D packaged processor architectures for massive, efficient data storage and retrieval. For such platforms, AMEBA proposes parallel, distributed software ‘agents’ organised in a hierarchical manner for fast data access and processing. An agent-based system can exploit the presence of a high density pool of processing cores in a way that future data and memory access intensive applications and services can be realised in markets including multimedia services, health and well-being, smart spaces etc.

1.3 Thesis structure

This thesis is organized as follows: Chapter 1 introduces the basic thematic area of video transcoding and related challenges and presents the specific topic of parallel scalable video decoding on which the work attempts to contribute. Chapter 2 presents the concepts of H.264 decoding related to this work and gives a thorough overview of the existing parallelization types and techniques together with their granularity levels. In Chapter 3 the selected parallel video decoding concept is introduced, together with the proposed scheduler for efficient scalable decoding in many-PE environments. In Chapter 4 the parallel video decoding model featuring the proposed scheduler implementation is presented in detail. Chapter 5 lists the details of the experimental set up and the set of measurements made using the developed decoding model. Chapter 6 discusses the evaluation of the results gathered as well as the proposed many-PE architecture for efficient parallel video decoding. Finally, in Chapter 7 several topics that have been identified appropriate for future work in terms of extending, improving and/or upgrading the work performed are proposed.
Chapter 2

Parallel video decoding design exploration

In order to propose an efficient parallel video decoding scheme, the video decoding algorithm has to be first analysed and evaluated in terms of parallelism. The tight dependencies featured in H.264 specification create challenges when trying to implement systems that include parallel, distributed H.264 video decoding operation. Thorough design exploration must also consider proper data granularity level to which several parallel approaches can be applied, as well as the types of parallelization that can be applied to dataflow algorithm applications like video signal processing.

2.1 Decoding algorithm dependencies

The generic H.264 decoder block diagram is shown in figure 2.1: As presented in detail in [4], H.264 decoding algorithm features a lossless and a lossy part. First, the compressed video enters the lossy stage in entropy decoding either under CAVLC or CABAC methods. Entropy decoder outputs the transformed and quantised coefficients of each frame’s macroblock in the video sequence. Those coefficients are then entering the lossy part of the algorithm: inverse quantisation and inversr DCT output the original residual of each macroblock. The prediction of each block, which is able to be constructed after the entropy decoding stage using data from
previously decoded macroblocks of the same (intra) or different frame (inter), is then added to the residual in order to reconstruct each original macroblock. Finally, the decoded macroblocks are input to the de-blocking filter in order to compensate for the artifacts introduced by the block-based frame (image) data processing.

In more detail, for each or the decoder’s component it can be noted that:

- **Entropy decoding**: Decoding is being done based on tables that are selected based on the tables of the previous macroblock. So each Macroblock is entropy decoded sequentially in raster scan order.

- **Inverse quantization**: The quantization value Q might be decided based on the value of the previous macroblock. The input coefficients are received from entropy decoder output.

- **IDCT** Input data are received from inverse quantization. There is a control dependency based on the encoded type of the macroblock. If the macroblock is coded as intra16x6 or chroma, an extra ‘DC’ step is performed before normal input idct.

- **Intra prediction**: Each macroblock needs pixels from the left, left top, top and
top right neighbor macroblocks as shown in Figure 2.2. They have to be already decoded before the current macroblock can be decoded. However, not all the pixels from the neighbouring macroblocks are needed in each type. Intra prediction has 13 possible encoding types; intra 4x4 modes are shown in Figure 2.3 taken from [12]. Since the encoding intra mode is known prior to decoding, the dependencies can be different in each case, so less amount of neighbours would be needed.

- **Motion compensation:** For motion compensation data from the previous or future frame(s) are needed. Those data are pointed from the motion vector of each macroblock. The motion vector is decoded from the stream, based on motion vectors of neighbour macroblocks as shown in Figure 2.4. Also, since the motion vector can be fractional, i.e. not showing exact pixel borders (can be quartel-pel), interpolation is done in order to produce those data. Interpolation requires extra pixels around the ones that are going to be interpolated, so additional pixels from the reference frame are needed. The amount of data needed from the reference frame(s) and thus the dependency density in this case, is specified from the length of motion vectors. An upper limit of motion vector length is used in order to constraint the dependency from motion compensation.

- **De-blocking filter:** The filtering operation receives the reconstructed macroblock as this is added from the outputs of idct and prediction (intra or MC). Then it needs up to 4 pixels from the left and top macroblock of the current macroblock (see Figure 2.2). The filtering operation also updates pixels of the neighbor macroblocks.
2.2 Parallelization types

In principle, when attempting to parallelize by partition an application in a multiprocessor environment, the parallelization is based on data-based or task-based partition [5]. In task-based partition, different tasks are identified and assigned to different processing elements (PEs) in a pipeline manner. In case of H.264 decoding, a reasonable minimum task split (based on the decoding algorithm flow) can be done as shown in Figure 2.5. Four task partitions can be defined:

1) Entropy decoding (includes also parsing)
2) IQ-IDCT
3) Prediction (intra or inter)
4) Deblocking

Tasks 2) and 3) can execute in parallel since they are independent from each other but they depend only on the parsing/entropy decoding (except from Intra 4x4 prediction case). When tasks 2) and 3) execute for macroblock n, entropy decoding can execute for macroblock n+1, and deblocking for macroblock n-1. Inherently, entropy decoding is a highly sequential task so it is decoupled from the other tasks. Further task partition in more fine-grain layer is possible, where the above tasks are similarly split in sub-tasks. The subsequent parallelization gain depends on the parallel architecture used i.e. the availability and suitability of PEs that would be allocated to tasks. Task level partitioning on its own achieves limited parallelism [5] as it is based only on the pipelining of tasks executing in parallel. Although it is possible to increase parallelization by splitting further the tasks to be
assigned in different PEs, this comes with a cost on inter-processor communication, while the scalability is still limited.

In data-based parallelization, several data structures (macroblocks in case of H.264 algorithm) are assigned to different PEs which can execute some or all the processing tasks. The concept is shown in Figure 2.6, taken from [5]. The critical factor in this case is to identify appropriate data structures that can be scheduled for concurrent processing. Assuming H264 decoding as the use case, a scheduling logic would assign Macroblocks (MBs) to PEs. Scheduling can be done either with static or dynamic approach. For H.264 decoding, in the general case MB decoding time varies according to input parameters of the encoded stream such as MB encoding types. Therefore it’s beneficial for the scheduler to be aware of these encoding parameters and decide dynamically for MBs assignment rather than assuming a worst case decoding time for all MBs. In the later case MB scheduling is static and simpler to implement, but non optimal, since significant input parameters such as the MB encoding type are not exploited, and essentially all slices or frames of a stream are treated in the same way by the scheduling algorithm.
In general, the main goal of algorithm parallelization is to speed-up its operation. This is measured by the respective speed-up factor against the original sequential version of the algorithm. For H.264, assuming that the time to process each MB is constant, that there is no overhead for parallel working modules synchronization and that there is an unlimited number of PEs available, the theoretical maximum performance can be estimated when the parallelization focuses solely in intra-frame/slice case. The maximum speed-up can be calculated according to the following equations:

\[
T_{seq} = mb\_width \times mb\_height \\
T_{par} = mb\_width + (mb\_height - 1) \times 2 \\
Max\_speedup = \frac{T_{seq}}{T_{par}}
\]

Maximum theoretical parallelization numbers for various resolutions can be found in [13].
2.2.1 Granularity

According to its specification, the H.264 video stream features an hierarchical structure that comprises Group of Pictures (GOP), frames, slices and macroblocks as shown in Figure 2.7, taken from [13]. Furthermore, macroblocks have one 16x16 pixel luma component and two 8x8 chroma components. The luma component can be further divided up to 16 4x4 sub-blocks when applying macroblock prediction. GOPs are completely independent and upon parsing they can be scheduled to be processed by free PEs, which can be alternatively called 'workers', in parallel. Parallelization in GOP-level is straightforward [14], however non-optimal as the actual decoding algorithm is not evaluated at all, while the GOP size buffering might introduce large delays in real-time applications. Each GOP features a different set of frames and each frame might be further divided in slices. Frames and slices can be also parsed directly from the stream. However frames are not independent from each other unless the H.264 prediction is done according to Intra mode always, which is not the general case. So, each frame depends on the previous and perhaps also on the next frame in display order (B frames). This can be seen in Figure 2.8 of [13]. In this example, 2 B frames are present for every P frame.
I frames do not require other frames for decoding while P frames require only previous ones in display order. Thus, once I and P frames are decoded, multiple B frames could be scheduled for decoding concurrently as they don’t depend on each other but only on I and P frames. In some cases though, B frames might not be present at all in the stream (Baseline H.264 profile, [15]), so independent frame decoding is not possible except from I frames. Therefore, a generic solution for efficient parallel decoding should be explored on a lower level of granularity as shown in Figure 2.9.

Figure 2.8: Frame types in a generic H.264 stream

![Diagram of H.264 data granularity]

Figure 2.9: H.264 data granularity
The next level to attempt parallelization is slice level. Slice decoding depends on slices from previous or future frames. Slices of the same frame can be processed (decoded) completely independently from each other; even in the slice borders the H.264 algorithm foresees static prediction mode selection that can prevent dependencies between slices to occur. Additionally, the motion estimation vectors can be set not to exceed the borders of a slice region. This can simplify the dependencies with previous slices. After each slice decoding finishes, the slice covering the same region in the next frame(s) can start.

Slice parallelization can obviously achieve better decoding speed-up as opposed to frame level parallelization, since slices in following frame(s) can be scheduled for processing before the previous frame is completely decoded. A large slice number per frame means higher speed-up. However it has 2 drawbacks: the bitrate/quality is slightly increased/decreased since the prediction types of the MBs in slice borders are pre-selected, thus they don’t yield the most efficient result. Indeed, in order to preserve slice independency, MBs that are located in slice borders cannot be predicted with prediction types that use pixels from neighbouring MBs. So it is possible that the final prediction is not the most efficient one in terms of bitrate or quality. According to [13] and [8] this overhead can be from 3 up to 30 percent in large frames with many slices per frame (64). The second drawback is that actual slicing is not an encoder independent operation; it has to be assumed that the encoder always splits a frame in specific number of slices. Additionally, the motion vectors have to have an upper limit in their range, so that they don’t exceed slice boundaries. Then in slice borders only specific intra modes are used. However, for the transcoding applications to which this exploration refers, it can be assumed that there is control over the encoder, thus it is possible to instruct specific slice partitioning on the frames of the video stream. Under this assumption, a parallel slice-level decoding solution would be able to process efficiently streams encoded under a set of pre-defined specifications. In fact, for applications where some visual quality degradation can be tolerated while achieving good speed-up factors, slice level parallelization is a good practice. Still for a more efficient solution, lower level
parallelization is worth to be evaluated.

The next level for parallelization is the macroblock level. In this level, there is no inherent parallelization, but all dependencies of paragraph 2.1 must be resolved before several MBs can be scheduled to be decoded in parallel. Another important issue in this level is that MB parsing is not a straightforward operation. There are no delimiters in the stream that can identify MB borders easily (like in slice case) which forces entropy decoding task to be executed before macroblock data can be scheduled for decoding. This concept is shown in Figure 2.10.

In addition to this, entropy decoding in algorithmic terms is referred as highly sequential while not much parallelization is able to be achieved [13]. So entropy decoding is the sequential constraint of the decoding algorithm in the parallelization effort. Due to this sequential behaviour, it is good practice to decouple it from the rest of MB decoding, for example CABAC entropy decoding can be performed for all the MBs in a frame/slice and the results to be stored in an intermediate buffer. Once entropy decoding is performed, the rest of MB decoding is executed in parallel using some kind of parallelization. Whether the entropy decoding is performed fully in a parallel region or not is a matter of implementation, but the decoupling of entropy task from the rest of the decoding tasks is essentially beneficial. This leads most of the parallelization techniques found in literature to focus on MB scheduling after entropy decoding. Consequently, a hybrid task-based and data-based parallelization scheme is adopted; it features task-based parallelization between entropy decoding and the set of prediction-reconstruction-filtering tasks, while for the latter set data-based parallelization is applied. Figure 2.11 from [9]

Figure 2.10: MB parallel decode concept
explains graphically the MB parallelization concept where the different colors reflect the task level parallelization between entropy (CABAC or CAVLC) decoding and the rest of decoding functions in MBs within the same spatial parallel group.

### 2.2.2 MB Parallelization techniques

In macroblock parallelization level, considering grouping MBs together and allocating them to a specific worker, several options are possible. Besides the speed-up factor, several additional criteria have to be considered including inter-worker communication, which has to be kept small, utilization of each worker (measured by the number of stalls in its operation), the scalability factor of each approach etc. Some of the techniques known as '2D wave' and presented in [10] are shown in Figure 2.12.
From several measurements done in [10], [16], 'MC' (several columns per worker) and 'NBSR' (slice per worker) techniques seem the most efficient ones. This is more evident especially in streams with higher resolution pictures where the scalability factor is more important. An alternative of NSBR is also an option (better in high resolution), where the slice partitions are not allocated to the same workers, but switching per frame so that a worker processes different regions of the frame, resulting in a more balanced overall workload.

The above techniques try to minimize the spatial dependencies of each MB. Thus the focus is to organize the MB scheduling for processing in order to exploit the fact that several MBs inside a frame can be executed in parallel. When there are also temporal dependencies i.e. dependencies in previous frames (or future for B frames) for a MB, they are assumed to be resolved already. Therefore dependence frames have to be decoded before the current one is decoded. The scalability of those techniques depends on the frame size. Parallelization and speed-up can happen only within each parallel region (i.e. frame, slice etc.)

To increase parallelization and scalability, each time MB dependencies are checked spatially, it is also useful to check if the data needed for the motion compensation process is available from a previous (or future) frame, without that frame necessarily having already been decoded totally. This is referred as ”3D wave” parallelization [13], [7], [6]. 3D wave achieves better speed-up at the cost of more complexity. The scalability in this case depends on the range of motion vectors of the MBs predicted using motion estimation. Indeed, if a motion vector range is small, then the area in the image to which it points will be decoded faster, thus it will become available faster for motion compensation of MBs in future frames. Also, it is beneficial when multiple frame references are used, where some of those frames might have long distance from the current one. In that case, decoding can start as soon as the reference area in the reference frame is decoded without having to wait for all the intermediate frames to be decoded as in '2D wave'. Figure 2.13 taken from [7] shows the ”3D wave” concept.
This technique raises some concerns regarding how efficiently it can be implemented. Indeed, using '3D wave' the number of MBs that can be processed concurrently might become so high that it will not be possible to efficiently implement parallel decoding in standard many-PE systems. This is because the number of MBs available for parallel decoding exceeds by far the number of available PEs. Furthermore, a large number of working PEs would provide better speed-up but would not exploit optimally the cycle budget of PEs. A study of '3D wave' parallelism having an upper limit in the number of parallel decodable frames or MBs shows that applying a limitation increases the decoding time but contributes to a more efficient utilization (close to 100%) of cores [6], [13].

After the evaluation of the H.264 algorithm it became evident that a hybrid task/data-based partition scheme is needed for efficient parallel decoding. It was also found that granularity up to MB level would be needed to be exploited for maximum efficiency. Between the MB parallelization techniques, this work will focus on 2D-wave case, with an emphasis on proposing an architecture with advanced scheduling that optimizes performance.
Chapter 3

Parallel video decoding architecture

Based on the previous chapter, an H.264 parallelization concept can be developed. The goal for this model is to be able to analyze H.264 streams up to a parallel region and then invoke a scheduling algorithm in order to expand parallel processes to be executed in a multiprocessor/multicomputer/multi-PE environment. Thus a high level architecture has to be defined first, followed by an efficient scheduling algorithm.

3.1 Concept

The proposed parallel architecture is shown in Figure 3.1. It features several blocks that express different functional parts of H.264 decoding algorithm and buffers between them for data storage and therefore enabling parallel operations in a pipeline style. At the input (far left), encoded H.264 streams are received from which a parsing (separation) in GOPs is made and separate independent GOPs are buffered. From a single GOP level, a slice parser can be invoked so that slices can be extracted and stored. Here the assumption is that an H.264 stream is encoded under a configuration set that applies slice partitioning inside frames, otherwise the slice parser operation just parses complete frames by default (1 slice/frame).
As mentioned before, slice level is the last level where direct parsing can be applied from the H.264 stream, so multiple slices can be spawned and executed in parallel with respect to some possible limitations directed by the parallelization method (2D, 3D, etc). In order to be able to parse also MBs so that they can be parallelized with previously discussed methods, the architecture features a slice parser working in parallel "fork" style with the entropy decoder. As already discussed previously, simple stream parsing is not enough for data extraction in MB level. Thus, this operation is necessary in order for the MB coefficients from CABAC (or CAVLC) decoding processes to be output and buffered. Furthermore, this last step results in availability of both MB coefficients and MB related encoded information (extracted by slice parsing) to the scheduling component.

At the point where the above information becomes available, H.264 decoding parallelization becomes essentially a scheduling problem. A scheduler must assign each available MB to a free worker for further decode processing. The term 'worker' is used in order to include every PE capable and available to process MBs in a multi-PE environment. Those workers are enabled by the scheduling operation which is described in the following section. Upon availability of multiple eligible workers, the MB reconstruct process can be further task-partitioned, for example pipelined in separate sub-processes including MB prediction, inverse quantization/inverse DCT and deblocking. Finally, after the MB reconstruct
operation, the decoded data per MB are available and H.264 decoding finalizes.

It is important to note that the discussed model is covering (up to) MB parallelization inside a frame, so the parallel region can be a custom set of MBs, a slice, or a complete frame. Expanding this model to cover complete streams can be done by replication of its components and assuming a shared memory architecture topology where components can have access to previous or future frame information when needed. Figure 3.2 shows an example of an expanded H.264 decoding parallelization case, where a stream features 2 GOPs and 4 slices. Indeed, in this example, a hybrid task parallelization (horizontally) and data parallelization (vertically) is applied.
3.2 Scheduling

Despite which of the previously mentioned or other emerging parallelization approaches is selected, the parallelization in MB level is inherently a scheduling problem. In our approach the basic ‘task’ the scheduler works on is the MB decoding operations after entropy decoding has finished. This task has been named ‘MB reconstruct’. At that point, the scheduler has available MB coefficients and MB encoding information. Additionally, the scheduler’s task dependence logic is actually defined by the dependencies that various MBs have. Having in mind those dependencies (presented in section 2.1), H.264 decoding for one parallel region (for example, a frame) can be represented by a Directed Acyclic Graph (DAG) like in Figure 3.3 which is presented in [17].

In this figure, the boxes represent the basic MB decoding ‘task’ and the arrows represent the dependencies between each other, while the coordinates of each MB inside the frame on a 2D X-Y axis are also listed. Similarly, with respect to a frame (slice) the respective task dependency graph will look like shown in Figure 3.4. As it can be observed, any H.264 ‘MB reconstruct’ task in the graph will have at most 2 previous tasks that it depends on. This is due to the dependencies that –in the general case- the MB has to respect in order to be decoded.

Scheduling for H.264 decoding has been approached either in static or dynamic mode [13], [9], [17], [18]. In static mode, there is a basic (but not realistic)
assumption that the MB decoding time is the same for all MBs which is not the case in real decoding. Typically, MB execution times are variant due to the different amount of residual information in different I and P-macroblocks, and (only for P-macroblocks) due to global memory/cache access times. This means that static schedulers, although easy to implement, cannot exploit full parallelization options since MB decoding order is pre-defined so if at any point there are more MBs available to be decoded they are not identified. Thus, it is evident that dynamic scheduling is more appropriate choice for parallel H.264 decoding.

In dynamic scheduling [9], a MB scheduler should keep track of information for each MB, taken from parsing and entropy decoding process. When the information about a MB is available from parsing and entropy decoding processes, the dependencies of the MB become known. Dependencies at this level can be just the number of macroblocks that must be decoded before the current MB can de
decoded. This identifier should be stored in a table.

Assuming that a task is defined as 'MB reconstruct', a task queue will specify which MBs are next to be decoded. The available MBs can be forwarded to any free worker (CPU, core, thread). This means that MB coefficients are copied by the entropy decode buffer to the workers using information from the task queue. When the task is processed, the table of dependencies should be updated in order to highlight the MBs that can be forwarded for execution. This update can be a decrement in the number of MBs that a MB needs to wait for before being ready to be decoded. When this identifier becomes 0, a MB can be signaled ready for processing. Figure 3.5 shows the possible spatial dependendies of a MB. The dynamic scheduling concept can be described as follows:

```plaintext
if X < FrameWidth-1 then
  decrement(dependencies[X+1][Y]);
  if dependencies[X+1][Y]=0 then
    ready_to_execute(macroblock[X+1][Y]);
  end
end

if X > 0 and Y < FrameHeight-1 then
  decrement(dependencies[X-1][Y+1]);
  if dependencies[X-1][Y+1]=0 then
    ready_to_execute(macroblock[X-1][Y+1]);
  end
end
```

In Figure 3.6 the block diagram of a dynamic scheduler architecture is shown. Each free worker checks the task queue for the next MB to process. Some research works try to optimize this latter issue by making the worker thread to check the dependency table immediately and identify a ready MB without having to check the queue [9]. This implies a dynamic behavior as the ready to be decoded
MBs are executed immediately. Other works try to optimize the thread to queue communication by introducing local queues instead of one global \cite{18}.

Although all above dynamic scheduling techniques improve parallelization, still towards parallelizing H.264 decoding they are not efficient enough. Indeed, in those cases the scheduler does not have input from the MB encoding parameters, so it stalls until all pre-defined MB dependencies are resolved even when this is not needed, adopting a 'worst-case' operation. However, a significant issue in the performance of a dynamic scheduler for H.264 decoding, would be the exploitation of encoding information of MBs during scheduling as considered in \cite{19}. If this information is used the performance of parallel decoding should be improved. The exploitable information that a scheduler can have available before assigning MBs to workers includes:

![Figure 3.5: Generic MB dependencies](image)

![Figure 3.6: Dynamic scheduling architecture](image)
- Slice type
- Quantization parameter
- Filter enable/disable and filter strength configuration values
- MB skip value
- MB type
- Motion vectors
- Intra 4x4 modes
- Chroma intra prediction modes
- Weighted prediction, bi-prediction information

The most important from these values is the MB type value which informs about the encoded type of the MBs. This information can alter the MB scheduling. Considering for example an intra encoded slice and all MBs encoded as intra 16x16 the task graph would look like in Figure 3.7, as opposed to the graph of Figure 3.4.

![Figure 3.7: H.264 graph for intra 16x16 MBs](image)
This is due to the change in the maximum number of dependencies of a MB with respect to intra 16x16 modes (DC, plane, horizontal, vertical). Furthermore:

- In MBs encoded with type intra 4x4, IDCT function is simpler than in intra 16x16 (no Hadamard processing), so decoding time changes
- In MBs encoded with type inter (motion estimation) different neighbor MBs might be needed according to motion vector extraction (8x16, 16x8, 8x8 partitions)
- In MBs encoded with type “B direct temporal”, no neighbours need to be checked for vector extraction (as in the general case)
- If skip mode is enable for some MBs, no further info exists in the stream, so IQ and IDCT tasks are not invoked at all
- If deblocking filter is switched off for a MB, the respective task is not invoked at all

The following table summarizes the sources of schedule altering in H.264 dynamic decoding.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra 16x6 mode</td>
<td>not max neighbor dependency</td>
</tr>
<tr>
<td>Intra 4x4 mode</td>
<td>IDCT sub-task faster, not max neighbor dependency</td>
</tr>
<tr>
<td>Inter 16x8, 8x16 mode</td>
<td>not max neighbor dependency</td>
</tr>
<tr>
<td>Inter B direct temporal mode</td>
<td>No neighbor dependency</td>
</tr>
<tr>
<td>Deblocking filter off</td>
<td>No filtering</td>
</tr>
<tr>
<td>Skip MB mode</td>
<td>No IDCT execution</td>
</tr>
</tbody>
</table>

In case of “3D wave” scheduling, a significant issue that is not covered by the previous techniques is how to have available MB information from previous worker threads in case of motion compensation. Either a thread should check a large number of local dependency tables or some information should reside in global
tables, available to all threads at any time in a parallel region. For example, if there are up to 5 reference frames possibly a global dependency table for these 5 frames can exist. Another issue in implementing the 3D wave lies in the frame buffer access limitation when multiple MBs in different frames will have to access reference macroblocks. Thus a worker arbitration/synchronization scheme is necessary.

In this work, a dynamic local (per-slice/frame) MB Encoding Information Aware (EIA) scheduler is proposed, that supports advanced 2D wave H.264 decoding parallelization. The scheduler’s functional flow diagram is shown in Figure 3.8.

Figure 3.8: EIA Scheduler
The scheduler accesses the MB encoding information produced by the stream parser and the CABAC/CAVLC module. Those modules are assumed already available from previous stages of the parallel decoding architecture. Based on this information, the table of dependencies is defined. The next step is to signal the first task for execution, as it always have dependency of value 0, i.e. it can immediately be forwarded to an execution queue for execution by an available worker. When MB decoding finishes, the table of dependencies is updated and the MBs that are ready for processing (dependency value of 0) are identified. Those new MBs are forwarded to the queue and when there are workers available, they are processed. Based on the worker type and availability several additional functionality can be added, for example if there are several workers that can undertake a MB decoding task, a selection for the optimal worker or a split in MB decoding subtasks can be done. The process described above is repeated until all the MBs of the parallel region (slice/frame) are decoded. The novel characteristics of the described EIA scheduler rely on the knowledge of the encoding information of the MBs. With respect to that feature the following functionality is added:

- The initialization process of the dependencies table assigns different dependency value to some MBs compared to the normal ‘worst-case’ dynamic scheduler. The dependency value per MB is now specified not only from its location inside the slice, but also from its type (i.e. intra16x16, P16x8, intra4x4). This is done in order to reflect the change in the task graph generation. For example, the intra16x16 MBs are further checked with respect to the 16x16 type. If those MBs are of type ‘vertical’ or ‘horizontal’ their ‘intra’ related dependency is always 1. A scheduler that does not check this encoding info would assign always an ‘intra’ dependency value of 2. Thus, for different encoding information there is a different decoding task graph produced per parallel region.

- When the dependencies table is updated, in addition to the MBs to the right and down left (see figure 3.5), also the MBs directly below the current MB is checked. If this MB is of type intra16x6, intra4x4, or P16x8, its dependency table is also
updated. This can lead to some dependencies to be resolved earlier, i.e. a task graph that allows faster execution is implemented.

- When a MB is submitted to the queue for execution, a range from 0 up to 3 MBs can be submitted for processing at the concurrently.

- When the deblocking filter is disabled, the dependency table is created based only on prediction dependencies of the MBs either in inter or intra prediction.

Upon availability of workers, a MB decoding task can be split into its sub-tasks a) iq/idct, b) prediction c) deblocking, and depending on the characteristics of each worker, a targeted optimal assignment can be done. For example, a computational intensive task such as IDCT could be assigned to a DSP among the workers. The proposed scheduling approach can improve the efficiency of parallel H.264 decoding. Efficiency primarily concerns increased speed-up, however additional issues can be exploited including number of resources needed, and resource utilization. Furthermore, evaluation of the encoding features of the video streams can be done in order to identify a possible efficient set of encoding features that enhances parallel decoding. In order to practically evaluate the proposed scheduling approach, a H.264 parallel decoding model is implemented featuring an implementation of the proposed EIA scheduler. The model is used in order to simulate parallel decoding of H.264 encoded streams, measure the results and draw conclusions. This is presented in detail in the next chapter.
Chapter 4

Parallel Video Decoding implementation

In this chapter a parallel H.264 video decoding model is presented. The preference on developing a decoding model over a platform-specific implementation was based on the need for rapid verification of the model’s proof of concept as well as for more thorough exploration of the various parameters of the parallel decoding process. This exploration would be better to be performed without any constraints that a platform specific implementation may include. Indeed, the experiments based on the model would have to assume a big pool of available workers connected in a communication-efficient manner. Since it is difficult to have such a platform available for experiments and simulation trials, a model implementation becomes an attractive alternative. Thus, the model can be used to evaluate various scenarios of parallel decoding, draw conclusions and finally recommend optimal manycore architecture for distributed video transcoding (featuring scalable video decoding).

4.1 Cofluent Studio

The H.264 parallel decoding model is implemented based on Cofluent Studio [20] software. Cofluent Studio is a model-driven engineering (MDE) software for modeling and simulating complex multi-core hardware/software systems. It generates transaction-level modeling SystemC code for architecture exploration
and performance analysis. Design in Cofluent studio features Timed-Behavioral modeling, Platform modeling and Architecture modeling. Since our implementation is a parallel decoding model that would explore a scheduler implementation performance, we focus on Timed-Behavioral modeling. Timed-behavioral modeling describes the functional characteristics of the system to be developed together with the communication properties:

- A graphic editor is used to define the building components and processes of the system, the communication properties (queues, shared variables, events, and connections) and the logic of each process either with source code or a combination of source code and graphical symbolic elements (loops, I/O, wait conditions etc.)

- The software allows also setting specific attributes for each component. Attributes can define function types, cycle periods, channel modes and capacities, access times for read/write operations etc.

- The algorithm section of Timed-Behavioral modeling allows invoking any c/c++ based source code and related libraries in order to describe the functionality of a component.

Once the model is ready it can be simulated and verified. Timing and component utilization values can be gathered from simulation. For verification, several charts can be constructed to show control flow and data transactions.

4.2 Parallel H.264 decoding model

A generic view of the H.264 decoding model is shown in Figure 4.1. The model emulates parallel ‘2D’ H.264 decoding, as described previously. This means that intra-frame parallelization is explored. In order for a frame to start decoding, all frames that must be decoded prior to that frame have already been decoded.
As input the model reads the MB encoding information of each MB from an input video sequence. The input video sequence is considered a closed GOP with one IDR start frame and several P, B frames following. It is also assumed that a H.264 stream parser and an entropy decoder operate in a previous step. The parser extracts MB encoding information from the stream, while the entropy decoder decodes the coefficients of each macroblock coded either in CAVLC or CABAC.

In the output, the model provides simulation time for input sequence decoding, worker utilization time and number of workers used during decoding. Additionally, charts showing data transactions during decoding process are available. Exploration of those charts can provide additional information about behavior of a specific component operation, connection element etc. The model supports an arbitrary number of resolution and frame count for the input sequences. Moreover, it supports an arbitrary number of slice partitions as well as workers per slice. This allows a wide range of configurations during simulation of various video sequences.
Sequential video sequence decoding simulation is also possible, by configuring only a single worker for decoding.

The concept of the model’s operation is described as follows: an input video sequence (or a GOP partition of a closed GOP sequence) is available in the input. The encoding information and the decoding time of each MB are sent in a slice by slice manner from the frame manager to the slice worker. Slices of the same frame are concurrently processed by invoking equal number of slice workers reading different slice information from a queue. The frame manager waits until all slices of the same frame are processed before it sends the slice(s) information of the next frame, until all frames are processed. The slice worker distributes MB encoding information to several MB workers. Several macroblocks can be processed concurrently by invoking an equal number of MB workers. The number of concurrently executed MBs is defined in run time by the scheduler’s algorithm and the availability of MB workers.

The basic blocks of the decoding model are:

- The frame manager
- The slice worker, which consists of
  - The scheduler
  - The MB worker

The frame manager is the front-end of the model and the overall controller. It receives input video MB data and initializes all required encoding information that is going to be fed to the decoding process. The frame manager outputs packets of MB encoding information data for a slice in a message queue which is shared by several slice workers. The slice workers are reading MB information data for one slice from the queue and start processing that slice. The Frame manager sends one packet per slice and then blocks until the slice schedulers signal that all slices of each frame are processed. Then the next frame’s slices are sent. When all frames of the sequence are processed, the frame manager terminates. The frame manager features 2 functions:

- *initframe*: initializes slice encoding data read from input text file
- *sendinfo*: sends slice encoding information to slice workers

The **slice worker** is the basic processing block of the model. It receives slice encoding information and decodes a complete slice of encoded MBs. When the slice is decoded, the slice worker outputs a signal to the frame manager that the processing is over and that the worker is available to receive a new slice for decoding. The slice worker has been designed as a hierarchical block consisting of:

a) **the scheduler**

The scheduler is the critical component of the model since it defines the parallel operation. It is implemented according to the proposed algorithm as presented in section 3.2. It receives slice encoding information from the frame manager as well as an acknowledgement from the MB worker when a MB has been processed. The acknowledgement is a message from a separate queue indicating which MB has just been processed. The message is essentially the MB’s coordinate. In the output, the scheduler puts ready tasks to an output task queue. The task information been sent is the decoding time of the MB in question and is read by the MB worker.

In more detail, the scheduler receives the slice encoding data and creates the dependency table of the tasks (MB decoding operations) that have to be executed within that slice. Whenever a task with dependency value of zero is identified, it is forwarded to an output queue. Multiple MBs can be ready to be executed on a time instance, so the tasks are forwarded to a queue. When a MB has been processed, a message arrives to the scheduler indicating which MB has been decoded. Then the scheduler updates the dependency table appropriately and the new MBs that can be submitted for execution in the output queue are identified and forwarded. The scheduler starts upon receiving slice information and submits directly the first macroblock. Then it blocks until ready MB messages arrive from the MB workers. When all MBs are processed, the scheduler signals the frame worker that the allocated slice has been decoded and terminates until a new slice is allocated by the frame manager. The scheduler features 3 functions:
- *inittables*: initializes dependency table with respect to MB encoding information and submits the first task for execution

- *updatedep*: after each task execution, checks neighbour MBs to update dependencies. If all dependencies for a MB are resolved, it is signalled as ready for execution. Returns when all MBs of a slice have been processed.

- *submitnewmb*: Reads how many MBs are ready for execution and forwards them to the output queue.

**b) the MB worker**

the macroblock worker is a simple component that emulates MB decoding by issuing a delay equal to the current MB’s decoding time. The component reads the next MB’s coordinate from the task queue written by the scheduler. It then reads the current MB’s decoding time from a shared data structure provided by the frame manager. Finally, the component writes the ready MBs queue with the MB’s coordinate after the delay value has elapsed and blocks until the next MB ready for execution is available. The macroblock worker features 1 function:

- *delay*: issues a delay equal to the MB’s decoding time and outputs a ready signal to the output queue read by the scheduler.

Once the decoding model is available, several experiments can be performed using real data from video sequences. The goal of the experiments would be to measure and evaluate the performance of the proposed scheduling approach. Based on those measurements, several conclusions can be extracted regarding to how to design and integrate an efficient scalable video decoding/transcoding system: optimal system architecture, optimal resource usage, video encoding features support etc. The next section describes the parallel video decoding experiments performed and presents the collected results.
Chapter 5

Experiments and results

A parallel H.264 decoding model is an attractive method to evaluate scheduling approach performance and eventually propose a scalable improvement in any distributed video processing application in need of fast decoding, such as video transcoding systems. The experiments that are described in this section are performed via simulation of H.264 video sequences decoding in Cofluent Studio environment. The set-up of the input video sequences and the configuration of the model are described below.

5.1 Input video streams set-up

Several H.264 video streams are freely available for testing and benchmarking purposes. However, it is much more efficient to be able to encode streams according to specific encoding features. In this experiment, the streams need to be partitioned in slices in order to be used as input to the decoding model. For that purpose the reference H.264 encoder software is used to encode raw YUV video streams. Streams of several resolutions are used so that the effect of resolution and slice size will be evaluated during decoding operation. The resolutions used are qcif, cif, vga, 720p and 1080p and 4K [21]. The slice size is configured as number of MBs and is kept stable within the same sequence in order to evaluate the performance in equal terms for all slice workers. The quantization parameter is set at a value of 28 which correspond to a median between 12 and 51 available value range. The
median value is chosen in order to avoid evaluating decoding of video under the extreme conditions; very low bitrate—very low quality or very high quality—very high bitrate. The video streams’ size was selected to be up to 20 frames long. Although no restriction in the size exists within the model, this value was selected in order not to have too long simulation times which would make the experiments difficult to manipulate. ‘Frame per second’ value was selected according to typical video stream display values of 25 or 30 fps.

In all video sequences the deblocking filter operation was switched off, so that the encoded data are unfiltered and thus no filtering is needed during decoding. The reason behind this lies in the attributes of the filtering operation in H.264 [4], where the filtering does not only update the pixels of the current MB on which it operates, but it can also update the neighbour pixels of either the left or the top neighbour MB. This is an extra dependency on a task graph of MB decoding tasks which is resolved automatically in the normal ‘worst case’ ‘2D’ scheduler implementation as explained in paragraph 3.2. However, in the proposed scheduling approach there are cases where the filtering operation might update the top MBs bottom pixels lines earlier than allowed. This would create fluctuations in video display and more importantly inconsistencies between the frame buffers of a parallel decoder and an accompanied encoder working, for example, in one transcoding system. In many systems the deblocking operation is postponed after the full decoding of a slice or a frame has occurred [22]. This is how the reference encoder also implements the deblocking filter operation. Thus in our measurements we only check performance of decoding streams that have not been filtered during encoding. It is assumed that filtering occurs at a later stage in the system. It has to be noted that parallelization of the deblocking operation is possible and can be implemented for example in a pipeline manner as an extension to the decoding model proposed. In anycase the usage of unfiltered encoded data does not postpone the generic usage of the proposed model.

The frame types of the sequences used are I or P type. B types is not necessary,
as the evaluation is on intra frame parallelization and not on ‘3D’ inter frame parallelization, where B frames can have significantly different decoding behavior. Intra encoding types are all included and motion estimation search range is left unrestricted (as opposed to a ‘3D wave’ parallelization). Rate control is disabled in order not to force an encoding type over another during encoding mode decision. Finally, when a MB is encoded in intra4x4 mode, the encoder must be configured so that the top right 4x4 sub-MB does not use the pixel samples of the top right MB in modes vertical-left, and diagonal-down-left (check Figure 2.3).

Table 5.1 summarizes the characteristics of the encoded input video sequences.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolution</td>
<td>qcif, cif, vga, 720p, 1080p, 4K</td>
</tr>
<tr>
<td>Frames per second</td>
<td>25, 30</td>
</tr>
<tr>
<td>Q parameter</td>
<td>28</td>
</tr>
<tr>
<td>GOP size</td>
<td>15 to 20</td>
</tr>
<tr>
<td>Frame type</td>
<td>I, P</td>
</tr>
<tr>
<td>Intra modes</td>
<td>16x16 all, 4x4 all</td>
</tr>
<tr>
<td>Motion estimation search</td>
<td>full</td>
</tr>
<tr>
<td>Deblocking filter</td>
<td>off</td>
</tr>
<tr>
<td>Rate control</td>
<td>off</td>
</tr>
</tbody>
</table>

The encoded video streams are fed to the reference H.264 decoder [23]. The reference decoder is modified in order to track the macroblock decoding time of each macroblock in the stream, in nanosecond (ns) accuracy. This is achieved by using high performance counters windows API [24]. Additionally, for each macroblock the encoding type and in case of intra16x6 prediction, also the 16x16 prediction mode is tracked. After the sequences are decoded by the reference decoder, all needed macroblock information is available.
5.2 Decoding model set-up

Before executing the model for an input sequence, the model graph has to be configured in Cofluent Studio with respect to how many slices exist in each frame of the sequence, so that an equal number of slice worker instances will be created. Additionally the number of MB worker instances inside each slice worker is configured depending on how many workers should be included for sequence decoding execution and monitoring. The queues’ capacities also have to be set to a value high enough so that no deadlock occurs between MB workers and the scheduler. For example, in sequential decoding simulation (1 worker), the queues capacities have to be increased to tolerate the increased token rate.

5.3 Measurements

After the macroblock information is available and the Cofluent model graph is configured, the simulations can take place and the measurements can be gathered. Table 5.2 lists the video sequences used for measurements. Column MB* lists the number of MBs whose types are useful to the EIA scheduler (intra16x16, intra4x4, P16x8) as explained in paragraph 3.2.

<table>
<thead>
<tr>
<th>Video Name</th>
<th>Resolution</th>
<th>Frame count</th>
<th>MBs total</th>
<th>MB*</th>
<th>Slice/frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>news qcif.yuv</td>
<td>176x144</td>
<td>5</td>
<td>495</td>
<td>129</td>
<td>3</td>
</tr>
<tr>
<td>news cif.yuv</td>
<td>352x288</td>
<td>15</td>
<td>5940</td>
<td>612</td>
<td>1,2,3,6</td>
</tr>
<tr>
<td>asahi vga.yuv</td>
<td>640x480</td>
<td>15</td>
<td>18000</td>
<td>2046</td>
<td>1,2,3,5</td>
</tr>
<tr>
<td>madagascar.yuv</td>
<td>1280x720</td>
<td>20</td>
<td>72000</td>
<td>8231</td>
<td>1,3,5,9</td>
</tr>
<tr>
<td>pedestrian.yuv</td>
<td>1920x1088</td>
<td>20</td>
<td>163200</td>
<td>54153</td>
<td>1,2,4</td>
</tr>
<tr>
<td>foreman4k.yuv</td>
<td>3840x2160</td>
<td>5</td>
<td>162000</td>
<td>41520</td>
<td>1,3</td>
</tr>
</tbody>
</table>

Table 5.2: Input H.264 video sequences

Via simulation of each sequence several measurements are gathered: slice/frame decoding time, total decoding time, number of workers used for MB processing and
average utilization of workers per slice. Each sequence is simulated in 3 different decoding modes: sequential operation (1 macroblock worker, 1 slice worker), parallel operation using normal ‘worst-case’ 2D scheduling and parallel operation using the proposed, encoding information-aware (EIA) scheduling.

The results per sequence are summarised in the tables that follow. The speed-up value is calculated by the following ratio:

\[ \text{speedup} = \frac{T_{seq}}{T_{par}} \]

where \( T_{seq} \) is the sequential decoding time and \( T_{par} \) is the parallel decoding time.

The value denoted as speedup* is calculated by the following ratio:

\[ \text{speedup}^* = \frac{tsch_{old}}{tsch_{new}} \]

where \( tsch_{old} \) is the parallel decoding time using normal 2D scheduling and \( tsch_{new} \) the parallel decoding time using EIA scheduling approach. It expresses the improvement of EIA scheduling over the normal 2D scheduling.

The average throughput is calculated as the ratio of the total number of bits produced in the decoder output divided by the total decoding time i.e.:

\[ \text{avg.
throughout} = \frac{x \times y \times \text{resolution} \times \text{frame number} \times 8 \times 1.5}{\text{total decoding time}} \]

Table 5.3 shows results for the cif input sequence while Table 5.4 shows results for vga input sequence.
Table 5.3: Results news cif sequence

<table>
<thead>
<tr>
<th>mode</th>
<th>slice per frame</th>
<th>speed-up</th>
<th>speed-up*</th>
<th>max. workers</th>
<th>average throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1</td>
<td>-</td>
<td></td>
<td>1</td>
<td>42,01</td>
</tr>
<tr>
<td>normal 2D</td>
<td>1</td>
<td>9,22</td>
<td></td>
<td>11</td>
<td>334,96</td>
</tr>
<tr>
<td>EIA</td>
<td>1</td>
<td>9,41</td>
<td>1,03</td>
<td>17</td>
<td>344,10</td>
</tr>
<tr>
<td>seq</td>
<td>2</td>
<td>-</td>
<td></td>
<td>1</td>
<td>43,19</td>
</tr>
<tr>
<td>normal 2D</td>
<td>2</td>
<td>10,85</td>
<td></td>
<td>9</td>
<td>531,24</td>
</tr>
<tr>
<td>EIA</td>
<td>2</td>
<td>11,06</td>
<td>1,02</td>
<td>12</td>
<td>541,83</td>
</tr>
<tr>
<td>seq</td>
<td>3</td>
<td>-</td>
<td></td>
<td>1</td>
<td>42,37</td>
</tr>
<tr>
<td>normal 2D</td>
<td>3</td>
<td>10,13</td>
<td></td>
<td>6</td>
<td>612,88</td>
</tr>
<tr>
<td>EIA</td>
<td>3</td>
<td>10,43</td>
<td>1,02</td>
<td>9</td>
<td>624,51</td>
</tr>
</tbody>
</table>

Table 5.4: Results asahi vga sequence

<table>
<thead>
<tr>
<th>mode</th>
<th>slice per frame</th>
<th>speed-up</th>
<th>speed-up*</th>
<th>max. workers</th>
<th>average throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1</td>
<td>-</td>
<td></td>
<td>1</td>
<td>46,70</td>
</tr>
<tr>
<td>normal 2D</td>
<td>1</td>
<td>10,13</td>
<td></td>
<td>19</td>
<td>630,88</td>
</tr>
<tr>
<td>EIA</td>
<td>1</td>
<td>10,43</td>
<td>1,03</td>
<td>28</td>
<td>649,54</td>
</tr>
<tr>
<td>seq</td>
<td>2</td>
<td>-</td>
<td></td>
<td>1</td>
<td>45,21</td>
</tr>
<tr>
<td>normal 2D</td>
<td>2</td>
<td>15,12</td>
<td></td>
<td>15</td>
<td>911,19</td>
</tr>
<tr>
<td>EIA</td>
<td>2</td>
<td>15,38</td>
<td>1,017</td>
<td>16</td>
<td>927,10</td>
</tr>
<tr>
<td>seq</td>
<td>3</td>
<td>-</td>
<td></td>
<td>1</td>
<td>46,16</td>
</tr>
<tr>
<td>normal 2D</td>
<td>3</td>
<td>17,57</td>
<td></td>
<td>10</td>
<td>1081,65</td>
</tr>
<tr>
<td>EIA</td>
<td>3</td>
<td>17,77</td>
<td>1,011</td>
<td>11</td>
<td>1093,75</td>
</tr>
</tbody>
</table>

Table 5.5 shows results for 720p input sequence.
Table 5.5: Results madagascar 720p sequence

<table>
<thead>
<tr>
<th>mode</th>
<th>slice per frame</th>
<th>speed-up</th>
<th>speed-up*</th>
<th>max. workers</th>
<th>average throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1</td>
<td>-</td>
<td></td>
<td>1</td>
<td>41,32</td>
</tr>
<tr>
<td>normal 2D</td>
<td>1</td>
<td>15,35</td>
<td></td>
<td>28</td>
<td>634,2</td>
</tr>
<tr>
<td>EIA</td>
<td>1</td>
<td>16,19</td>
<td>1,055</td>
<td>45</td>
<td>669,17</td>
</tr>
<tr>
<td>seq</td>
<td>3</td>
<td>-</td>
<td></td>
<td>1</td>
<td>46,08</td>
</tr>
<tr>
<td>normal 2D</td>
<td>3</td>
<td>27,25</td>
<td></td>
<td>15</td>
<td>1255,82</td>
</tr>
<tr>
<td>EIA</td>
<td>3</td>
<td>28,26</td>
<td>1,04</td>
<td>21</td>
<td>1302,33</td>
</tr>
<tr>
<td>seq</td>
<td>9</td>
<td>-</td>
<td></td>
<td>1</td>
<td>48,21</td>
</tr>
<tr>
<td>normal 2D</td>
<td>9</td>
<td>37,87</td>
<td></td>
<td>5</td>
<td>1826,26</td>
</tr>
<tr>
<td>EIA</td>
<td>9</td>
<td>38,01</td>
<td>1,004</td>
<td>8</td>
<td>1832,83</td>
</tr>
</tbody>
</table>

Table 5.6 shows results for fullHD input sequence.

<table>
<thead>
<tr>
<th>mode</th>
<th>slice per frame</th>
<th>speed-up</th>
<th>speed-up*</th>
<th>max. workers</th>
<th>average throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1</td>
<td>-</td>
<td></td>
<td>1</td>
<td>43,82</td>
</tr>
<tr>
<td>normal 2D</td>
<td>1</td>
<td>23,91</td>
<td></td>
<td>46</td>
<td>1047,77</td>
</tr>
<tr>
<td>EIA</td>
<td>1</td>
<td>26,03</td>
<td>1,089</td>
<td>67</td>
<td>1140,78</td>
</tr>
<tr>
<td>seq</td>
<td>2</td>
<td>-</td>
<td></td>
<td>1</td>
<td>39,90</td>
</tr>
<tr>
<td>normal 2D</td>
<td>2</td>
<td>34,53</td>
<td></td>
<td>34</td>
<td>1377,91</td>
</tr>
<tr>
<td>EIA</td>
<td>2</td>
<td>37,98</td>
<td>1,10</td>
<td>40</td>
<td>1515,51</td>
</tr>
<tr>
<td>seq</td>
<td>4</td>
<td>-</td>
<td></td>
<td>1</td>
<td>42,62</td>
</tr>
<tr>
<td>normal 2D</td>
<td>4</td>
<td>45,65</td>
<td></td>
<td>17</td>
<td>1945,57</td>
</tr>
<tr>
<td>EIA</td>
<td>4</td>
<td>47,99</td>
<td>1,05</td>
<td>30</td>
<td>2045,58</td>
</tr>
</tbody>
</table>
Table 5.7 shows results for 4K input sequence

<table>
<thead>
<tr>
<th>mode</th>
<th>slice per frame</th>
<th>speed-up</th>
<th>speed-up*</th>
<th>max. workers</th>
<th>average throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>1</td>
<td>-</td>
<td></td>
<td>1</td>
<td>51.75</td>
</tr>
<tr>
<td>normal 2D</td>
<td>1</td>
<td>46.07</td>
<td></td>
<td>87</td>
<td>2663.04</td>
</tr>
<tr>
<td>EIA</td>
<td>1</td>
<td>51.45</td>
<td>1.12</td>
<td>139</td>
<td>2384.86</td>
</tr>
<tr>
<td>seq</td>
<td>3</td>
<td>-</td>
<td></td>
<td>1</td>
<td>42.44</td>
</tr>
<tr>
<td>normal 2D</td>
<td>3</td>
<td>79.26</td>
<td></td>
<td>43</td>
<td>3364.55</td>
</tr>
<tr>
<td>EIA</td>
<td>3</td>
<td>83.96</td>
<td>1.06</td>
<td>65</td>
<td>3563.85</td>
</tr>
</tbody>
</table>

Measurements listed in the tables can be also presented graphically where the effect of EIA scheduling is clearly identified. Three charts per resolution can be extracted: worker count, speed-up and throughput (Y axis) in relation to slice per frame count (X axis). Figure 5.1 shows the charts for vga resolution, figure 5.2 shows the charts for 720p resolution and figure 5.3 shows the charts for full HD resolution.

Additionally, another set of charts can show the average utilization of each worker in a parallel region (slice/frame) for each sequence in a X-Y axis. In those charts, the Y axis lists the utilization percentage of each worker and the X-axis lists worker id by their increasing number. The utilization of workers is presented based either on normal 2D scheduler or on the EIA scheduler. Figure 5.4 shows the charts for vga resolution, while figure 5.5 shows the charts for 720p resolution and figure 5.6 shows the charts for full HD resolution. The measurement results show a positive impact of the proposed EIA scheduling approach in parallel H.264 video decoding. In the next chapter, the results are discussed in detail.
Figure 5.1: Vga results
Figure 5.2: 720p results
Figure 5.3: Full hd results
Figure 5.4: Worker utilization results vga 1 slice/frame and 3 slices/frame
Figure 5.5: Worker utilization 720p 1 slice/frame and 3 slices/frame
Figure 5.6: Worker utilization fullHD 1 slice/frame and 3 slices/frame
Chapter 6

Conclusions

The parallel video decoding experiment described in the previous chapters provides a rich set of measurements based on which significant conclusions can be discussed. In overall, the proposed EIA scheduling approach shows improved performance compared to normal 2D scheduling used in parallel H.264 decoding. Since the performance results are promising, implementation details can be proposed. In this case, the critical part for any implementation would be the optimal allocation of the parallel decoding processes in high density multi-layer multicore architectures.

6.1 Performance

The experimental results can be used to assess trade-offs specifically in speed, resource utilization and performance goals in order to optimally design next generation distributed video decoding solutions. The results are discussed below in more detail in terms of a) speed-up, b) throughput c) worker count d) worker utilization.

• Speed up
Parallel decoding using EIA scheduler achieves high speed-up value compared to sequential decoding. The speed-up value scales with the slice number, as shown in Figures 5.1, 5.2, 5.3; the more slices available the more independent parallel operation per frame is possible. It also scales with the video resolution ranging
from a value of 6 (cif) up to 83(4K). This is shown in Figure 6.1, where same slice per frame value is assumed for all different resolutions.

Compared to normal 2D scheduler, EIA scheduler achieves a speed improvement of up to 12 % (for 4K resolution). The improvement depends also on slice count and on resolution. This is due to the fact that the scheduling algorithm performs according to the spatial size of the parallel region i.e. the size of the slice/frame. Thus in large slices and/or high frame resolutions the improvements are greater. When encoding a sequence with many slice partitions, the speed-up improvement is almost completely due to the independent slice decoding rather than the scheduling approach. However, since too many slices degrade the visual quality [8], [13], it is not recommended to encode sequences with heavy slice partitioning when specific quality goals are targeted. In that case, EIA scheduler provides considerable speed-up improvement.

It is also interesting to compare experimental results with the maximum theoretical parallelization speed-up limits as calculated in [17]. The values in [17] refer to maximum speed-up for one frame. Using the model we simulate the first intra frame of a 2160p (4K) sequence with both the normal 2D and the EIA scheduler. The results are shown in table 6.1:
Table 6.1: Experimental vs theoretical 2D speed-up

<table>
<thead>
<tr>
<th>video resolution</th>
<th>max. theoretical speed-up</th>
<th>speed-up normal 2D scheduler</th>
<th>speed-up EIA scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>3840x2160</td>
<td>63.78</td>
<td>47.21</td>
<td>80.2</td>
</tr>
</tbody>
</table>

Since the theoretical speed-up is calculated based on the assumption that MB decoding time is constant and the parallelization operates with worst-case dependencies –identical to normal 2D scheduler- the lower experimental speed-up for the legacy scheduler is justified. However, using EIA scheduling approach for real (non-constant) MB decoding timing, the maximum speed-up exceeds the value forecasted by the current theoretical model. The experimental values express the maximum speed-up for a frame, as the frame used is the first, intra frame of the sequence, which practically has the maximum speed-up in the proposed EIA scheduling algorithm. The reason why speed-up value from EIA scheduling exceeds the theoretical calculation can be seeing by comparing Figures 3.4 and 3.7: in the theoretical model of 2D scheduling fewer tasks can be spawned at any given time unit. In EIA scheduling, more tasks can be spawned after 'MB decode’ tasks are finished, provided that there are MB types that can be exploited by EIA scheduler for faster task placement in the task execution queue (intra MBs, inter P16x8 MBs etc).

• Throughput

As shown in Figures 5.1, 5.2, 5.3, the proposed EIA scheduler achieves throughput improvement linear to the speed-up improvement i.e. has a maximum of up to 12 % for 4K sequences. Identically to speed-up, throughput increases with the number of slices, i.e. the number of independent decoding processes. For 4K sequences, decoding with 3 slices per frame in a 5-frame GOP leads to output throughput of more than 3.5 Gbps. Of course, throughput scales also with video resolution as
shown in Figure 6.2, where as in speed-up case we assume same slice per frame value per resolution.

- Worker count

One of the basic motives for exploring an improved scheduling algorithm for parallel H.264 video decoding is to exploit the presence of a large pool of processing elements (PEs) -generally referred as workers- in future integrated computing platforms. Thus, the experimental results of worker count during decoding are of significant importance. Results from Figures 5.1, 5.2, 5.3 show that EIA scheduler exploits better the presence of additional workers which would not be used at all for decoding tasks with normal 2D scheduling. The worker count per parallel region is increased up to 68% when decoding full HD sequences. In general, the bigger the parallel region is, the more workers are invoked for decoding of that region. That’s why the worker count per region decreases with slice count but scales with resolution (for the same slice partition) as shown in Figure 6.3. As calculated in the case of speed-up, Table 6.2 summarizes the comparison of the experimental worker count per parallel region results with the maximum theoretical worker count according to [17]. The parallel region in this example is a complete frame, i.e. 1 slice/frame.
Table 6.2: Experimental vs theoretical 2D worker count

<table>
<thead>
<tr>
<th>resolution</th>
<th>max. theoretical workers</th>
<th>worker count normal 2D scheduler</th>
<th>worker count EIA scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1280x720</td>
<td>40</td>
<td>28</td>
<td>45</td>
</tr>
<tr>
<td>1920x1088</td>
<td>60</td>
<td>46</td>
<td>67</td>
</tr>
<tr>
<td>3840x2160</td>
<td>120</td>
<td>85</td>
<td>139</td>
</tr>
</tbody>
</table>

From table 6.2 it can be observed that EIA scheduling exceeds also the number of workers used for decoding a frame as forecasted by the theoretical model. This is explained by the same justification given for the speed-up excess case as described above. It is important to note that the worker count values discussed so far are the number of workers used for decoding in one parallel region, typically a slice. Depending on how many slices are used when the video sequence is encoded, the total amount of workers needed to decode the sequences used in the experiment is listed in table 6.3. It is evident that in contrast with normal 2D scheduler, the total worker count scales with slice count when the EIA scheduler is used for parallel H.264 video decoding. Table 6.3 also lists the increase in worker count when EIA scheduler is used. Comparing the worker count increase with the speed-up increase from tables 5.3, 5.4, 5.5, 5.6, 5.7 it can be noticed that speed-up increase requires
a lot higher worker count increase. For example, for a 4K sequence a speed-up improvement of 12% is achieved by increasing the workers by 59%. Therefore, it is important to consider worker utilization details.

Table 6.3: Total workers used for parallel decoding

<table>
<thead>
<tr>
<th>Video Sequence Name</th>
<th>Slice per frame</th>
<th>total workers normal 2D scheduler</th>
<th>total workers EIA scheduler</th>
<th>worker increase (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>news cif</td>
<td>1</td>
<td>11</td>
<td>17</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>18</td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>18</td>
<td>27</td>
<td>33</td>
</tr>
<tr>
<td>asahi vga</td>
<td>1</td>
<td>19</td>
<td>28</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>30</td>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>30</td>
<td>33</td>
<td>10</td>
</tr>
<tr>
<td>madagascar 720p</td>
<td>1</td>
<td>28</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>45</td>
<td>63</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>45</td>
<td>72</td>
<td>60</td>
</tr>
<tr>
<td>pedestrian 1080p</td>
<td>1</td>
<td>46</td>
<td>67</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>68</td>
<td>80</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>68</td>
<td>120</td>
<td>76</td>
</tr>
<tr>
<td>foreman 2160p</td>
<td>1</td>
<td>87</td>
<td>139</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>129</td>
<td>195</td>
<td>51</td>
</tr>
</tbody>
</table>

- Worker utilization
  The proposed EIA scheduler can spawn an increased number of parallel video decoding processes, allocated in a high number of workers. For high-end video
sequences (4K or more), more than 130 workers can operate in parallel and perform MB decoding tasks. Since the number of workers scales with increasing resolution, it is critical to assess worker utilization during decoding.

EIA scheduling algorithm exploits inherent parallelization in H.264 encoded streams and spawns higher number of parallel processes when compared to normal 2D scheduling. From the experimental results shown in charts 5.4, 5.5, 5.6 it can be observed that EIA scheduler increases the worker utilization as the parallel region grows and more workers are invoked in the decoding process. As observed from the charts, a worker with low id number is invoked earlier in the decoding process compared to a worker with higher id. For example, worker 0 starts executing a MB decoding task immediately (MB 0) while worker 87 executes a similar task when 86 MBs are being decoded and the 87th is eligible for processing. Since the normal 2D scheduler does not exploit all parallelization capabilities, it does not invoke as many workers and for that many tasks as EIA scheduler.

In low worker id though, the utilization is similar for the two scheduling approaches. In fact, for full HD sequence with 1 slice per frame shown in 5.6, the normal 2D scheduler shows better utilization in low worker id which gradually decreases as the worker id increases. This can be explained by the fact that as the parallel region grows, more parallelization can be exploited i.e. more workers can be invoked in the process. This means that the MB decoding tasks will be distributed in more workers whose number grows with the resolution and size of the parallel region. Since the normal 2D scheduler cannot invoke more workers than EIA scheduler, it loads more tasks to fewer workers, so utilization of these workers will be higher with normal 2D scheduling that with EIA. However, the experimental results show that significant utilization difference in low worker id happens only in full HD sequences with 1 slice per frame; in other cases EIA scheduler has similar or slightly better utilization performance, while in high worker id it gives always better results. Thus, unless the parallel region is very large (complete frames above fullHD), the EIA scheduler improves worker utilization compared with the normal 2D scheduler.
Practically though, in very high resolution sequences slice partitioning is normally applied to accelerate the decoding time.

6.2 Optimal allocation in multicore architecture

Parallel video decoding has been already realized in mainstream multi-core architectures [7], [18], [22]. In this work however, the focus is to consider a proposal for an optimal allocation of parallel video decoding in hardware architectures featuring a very high-density of processing and storage elements, as described in the AMEBA project concept [11]. In that context, high-density, multilayer, “3D” system-in-package based ‘node’ topologies are available for design exploration. With respect to AMEBA terminology shown in Figure 6.4, the proposed parallel decoding model can be implemented in those architectures as follows:

- MB worker modules are the Cell Agents i.e. lowest level kernels that run in multiple instances in the platform.

- Slice worker modules are the Cluster Agents i.e. modules that implement structures (scheduling) that manipulate data-segments and monitor and control the cell agents

- Slice parser and entropy decoder modules are the Platform Agents i.e. higher level modules decomposing and providing application input data into segments for the lower levels and trigger the overall application (decoding).

This set of cluster agents comprising the platform can have arbitrary independent instances in multi-layer 3D hardware architecture. Furthermore a parallel video decoding application in such architecture would be mapped as follows:
• H.264 GOP data stored separately. H.264 streams are composed by ‘closed’ GOPS, which makes their processing independent from each other as shown in Figure 6.5. Thus no expensive data exchange between GOPS is needed. GOP data can be stored arbitrarily distantly (spatially) from each other. Program data can be ‘moved’ close to the data location easily. Thus in GOP decoding level, implementation follows a distributed memory approach.

• Within a GOP area, the used parallel regions (complete frames or several slices per frame) form individual clusters as shown in Figure 6.6. Clusters share the GOP
memory via local interconnection; possibly a bus with a controller acting as the ‘master’ for communication. This local interconnection would connect from one to several clusters; however since extended slice partition should not be applied the number of clusters will not be too large. That eliminates contention problem in GOP memory access. A memory controller is needed also since two agents in each cluster –the entropy decoder and the parser- perform read access in the memory. A separate agent act as the scheduler in each cluster controlling the multiple MB cell agents. Each cluster can have a dedicated interface to an external GOP decode buffer where the final decoded video is stored. This is shown in figure 6.7.
• MB cell agents would have private input cache memory each. The cache would include the needed macroblock’s coefficients and the neighbor MB’s pixels or the needed reference motion compensation area for decoding. The decoded data would be stored in an output buffer. Since the number of MB agents can scale in very large number, distributed shared memory design is needed for the output buffer to which all MB agents need access. Additionally, those agents need to have access to output buffers in other clusters which are processing frames which are needed for motion estimation reference. Figure 6.8 shows the high-level architecture of the cluster agent implementing a slice worker.

Considering the processing elements’ architecture of the cluster agent (cell MB agent), they must form an efficiently connected processing group. Their interconnection should be optimized with respect to the dependencies among the elements. Due to the high number of workers (up to 139 per cluster in 4K sequences), a shared path dynamic interconnection scheme is not convenient. Furthermore, static interconnection with all possible dedicated links between processing elements would provide concurrent data transactions but would be too costly for practical implementations. In addition, some links may not be used at all.
as there can be processing elements that do not have to communicate.

Actually, in real implementations communication of PEs is defined by the mapping of MBs to MB workers and their subsequent allocation to processing elements. The scheduler’s dependency table defines the dependencies among MBs during run time from which the decoding task graph is extracted. As shown before, this task graph depends on the encoding information of MBs in each sequence thus it can vary at run time in the general case. Since the task is defined as 'MB decode’ (actually 'MB reconstrucr’), the task graph reflects also spatial dependencies between MBs in a frame. Spatial dependency defines decoded MB data that have to be used from MB neighbours. So, mapping decoding tasks of macroblock neighbours to the same or to physically neighbouring processing elements can also vary in different frames/slices.

There can be several ways of mapping MB decoding tasks to processing elements. The task mapping algorithms often located together with the scheduling algorithm in a controller component as in Figure 6.8 map decoding tasks with various criteria including energy efficiency, load balancing, memory accesses etc [25], [26]. Depending on the task mapping approach, different interconnections will be required from simple linear connection to topologies including star, ring, mesh etc.

This leads to the conclusion that a switched network with dynamic re-configurable connections is more appropriate for connecting processing elements. Being dynamic, the interconnection network could implement all possible processing elements’ connections. Being reconfigurable, instantiation of only the needed number of elements would be possible when the implementation platform would be for example re-configurable FPGA.
Chapter 7

Future work

In this work, a parallel video decoding model for H.264 streams has been developed based on an encoding-info-aware (EIA) scheduler for improved parallelization and decoding speed-up. Decoding simulation of several H.264 encoded sequences was performed using the model. Experimental results were studied and important conclusions about the scheduler’s contribution in performance were drawn. This study has also revealed several topics where future work should focus in order to enhance the parallel decoding model functionality and upgrade its operation.

• Refine model for sub-tasks in MB level.

The parallel decoding model is based on the task definition of ‘MB level’ decoding. The MB worker is handling the task of decoding a complete MB. As shown in paragraph 2.2, MB decoding operation can be parallelized in function-based level when considering the dependencies in H.264 decoding. Specifically, MB prediction and IQ/IDCT do not have dependency on each other; they can be executed in parallel once the parsing and CABAC decoding operations have completed. If sub-tasks of ‘perform IQ/IDCT’ and ‘produce MB prediction’ are defined, they can be allocated in different workers and the overall MB decoding time will be reduced. The model can support such simulations directly by giving the MB worker time a new value. This can be, for example, the overall profiled decoding time reduced by the time of MB prediction process. Otherwise, an additional worker component can be included in the model which will express together with the existing worker
the set of workers allocated for MB task execution. Since the proposed model does not allocate ‘worker’ to a single physical processor, several workers can operate concurrently and assumed to be included in the same physical resource, for example threads in a multicore CPU.

- ‘3D Wave’ inter-frame parallelization support.

A basic feature of the proposed H.264 decoding model is that it supports intra-frame parallelized ‘2D Wave’ decoding. However, the same model can be extended to support 3D parallelization. In that context, the frame manager –or a new higher hierarchy component- can include control structures to manage the parallel decoding of several frames in a sequence. The parallel decoding of several frames essentially requires management of the motion estimation dependencies of frames and a mechanism to monitor and resolve those dependencies. Additionally, the frame manager will not have to wait until all slices in a frame are decoded; once a slice is decoded, all slices in other frames that have dependencies in that slice can start decoding.

- Energy efficient decoding exploration.

As discussed in the previous chapter, the invocation of a high number of new workers with the EIA scheduling, provides enhancements in performance, however considerations have to be raised on effective worker utilization. From the energy efficiency or general design economy point of view, it might be problematic to use more workers for decoding when each one is utilized for only a small fraction of its total time budget. Thus a monitor/control structure and a switching policy can be defined in order to exploit free workers between parallel regions and allocate them with decoding tasks among different slices/frames. This will improve dramatically the utilization rate thus will contribute to a more energy efficient decoding on many-PE environments.

- Platform specific decoding measurements.

Finally, the decoding model can be updated with profiling measurements of data
transfers, memory read/write access times etc. That data can be used to replace the
default values of the timing attributes of the queues in the model, in order to simulate
video decoding with respect to real data access times per platform. There can be
a wide selection of diverse platforms with different memory organization and data
exchange architectures. Updating the model with real numbers on data transfer can
output more accurate results and propose optimal platform and optimal mapping
when integrating parallel distributed video decoding with specific performance goals.
Bibliography


