

Evaluation of data-parallel H.264 decoding approaches for strongly resource-restricted architectures

Florian H. Seitner · Michael Bleyer ·
Margrit Gelautz · Ralf M. Beuschel

Received: date / Accepted: date

Abstract Decoding of an H.264 video stream is a computationally demanding multimedia application which poses serious challenges on current processor architectures. For processors with strongly limited computational resources, a natural way to tackle this problem is the use of multi-core systems. The contribution of this paper lies in a systematic overview and performance evaluation of parallel video decoding approaches. We focus on decoder splittings for strongly resource-restricted environments inherent to mobile devices. For the evaluation, we introduce a high-level methodology which can estimate the runtime behaviour of multi-core decoding architectures. We use this methodology to investigate six methods for accomplishing data-parallel splitting of an H.264 decoder. These methods are compared against each other in terms of runtime complexity, core usage, inter-communication and bus transfers. We present benchmark results using different numbers of processor cores. Our results shall aid in finding the splitting strategy that is best-suited for the targeted hardware-architecture.

Keywords Video decoding · H.264/AVC · Multimedia · Multi-core · Embedded architectures

1 Introduction

The H.264 video standard [9] is currently used in a wide range of video-related areas such as video content distribution and television broadcasting. Compared to preceding standards such as MPEG-2 and MPEG-4 SP/ASP, improved coding efficiency could be reached by introducing more advanced pixel processing algorithms (e.g. quarter-pixel motion estimation) as well as by the use of more sophisticated algorithms for predicting syntax elements from neighbouring macroblocks (e.g. context-adaptive VLC). These new coding tools result in significantly increased CPU and memory loads required for decoding a video stream. In environments of limited processing power such as embedded systems, the high computational demands pose a challenge for practical H.264 implementations [6]. Multi-core systems provide an elegant and power-efficient solution to overcome these performance limitations. They enable running complex multimedia applications such as, the one described in [8] on embedded architectures.

However, the efficient distribution of the H.264 video algorithm among multiple processing units is a non-trivial task. For using the available processing resources efficiently, an equally balanced distribution of the decoder onto the hardware units must be found. The system designer has to consider data dependency issues as well as inter-communication and synchronization between the processing units. Furthermore, the resource limitations in an embedded environment such as low computational power, small-sized on-chip memories and low bus bandwidth require an efficient software design. A parallel decoder approach for mobile systems must be able to work under these resource restrictions.

Florian H. Seitner · Michael Bleyer · Margrit Gelautz · Ralf M. Beuschel
Institute for Software Technology and Interactive Systems,
Vienna University of Technology,
Favoritenstrasse 9-11/188/2, A-1040 Vienna, Austria
E-mail: {seitner, bleyer, gelautz, schreier}@ims.tuwien.ac.at

The contributions of this paper are twofold. Our first and major contribution is an evaluation of different parallel decoding approaches. We evaluate these approaches in terms of runtime complexity, efficient core usage, data transfers and buffer sizes. This information is of prime importance when designing a resource-efficient multi-core decoder. Our results can be exploited for choosing that parallel approach which performs best under certain hardware restrictions. We provide a detailed overview of six different data-parallel splitting methods for H.264 and compare these approaches against each other. We investigate the impact of raising the number of cores on the resource requirements by processing different test sequences.

To accomplish our benchmark, we introduce a high-level simulation methodology for parallel decoder systems. This methodology represents the second contribution of this paper. The methodology, referred to as *Partition Assessment Methodology*, allows estimating the complex runtime behaviour of a parallel decoding architecture. The Partition Assessment Methodology models critical multi-core issues such as data-dependencies between cores, buffer limitations and concurrency. We overcome the problems of traditional estimation techniques that are typically not suitable for multi-core environments and not flexible enough for exploring a large range of different decoder configurations.

In the following, we go into more detail on both contributions and discuss them in the context of prior work. First, we describe methods for splitting the H.264 decoder onto multiple processors. These methods form the foundation for the splitting approaches investigated in this paper. We then focus on runtime estimation techniques that we consider as being related to the proposed Partition Assessment Methodology.

1.1 Methods for parallel decoding

Various approaches for parallelizing the H.264 decoding process have been introduced in literature. Van der Tol et al. [25] investigate methods for mapping the H.264 decoding process onto multiple cores. Functional splitting of a video decoder and the use of macroblock pipelining at thread-level are demonstrated in [3, 4, 10, 21]. Zhao and Liang [28] exploit frame parallelism in the Wavefront technique. A hierarchical approach working at group-of-picture and frame levels is demonstrated in [20]. Meenderinck et al. [13] investigate the scalability of H.264 for a data-parallel decoding approach operating on the macroblock-level and on multiple frames in parallel. Moreover, these authors introduce an efficient technique for H.264 frame-level parallelization, namely the 3D-Wave strategy. All of those papers share a focus on parallelization in terms of algorithmic scalability. Upper limits on the number of processors and frames processed in parallel are given. However, the memory restrictions of embedded environments make these approaches hardly usable for mobile and embedded architectures. More resource-efficient H.264 splitting approaches are introduced in [23, 24, 26]. The focus of these papers is on efficient decoder implementations for embedded architectures.

This paper focuses on evaluating the behaviour of different parallel decoding approaches in terms of runtime complexity, efficient core usage, data transfers and buffer sizes. This information is of prime importance when designing a resource-efficient multi-core decoder. The information can be exploited for choosing the parallel approach which performs best under certain hardware restrictions.

However, benchmarking a large variety of different decoder partitionings running on multiple cores poses two problems. Firstly, architectures with 4 and more cores and appropriate benchmarking tools are not yet commonly available. Secondly, implementing the different partitioning methods and adapting them to run on a specific architecture is a highly labour-intensive task and limits the number of approaches that can typically be evaluated. The estimation techniques described in the next section provide a way to gain insights into a video decoder's runtime behaviour without necessarily requiring a completely functional system implementation.

1.2 Runtime estimation techniques

Analysing an algorithm's definition (e.g., its source code) can provide coarse estimation on the computational complexity of an algorithm [12, 17]. However, these techniques do, in general, not consider the impact of the input data on the program's execution behaviour (e.g., input-dependent recursions and branches). For analysing video decoding systems where the runtime behaviour is typically strongly dependent on the input stream, these techniques are not suitable.

Profiling methods [1, 5, 7, 15, 18, 19, 27] overcome this problem. These methods either simulate or extend the program in a way that the program's execution behaviour can be observed during runtime. This allows measuring the complexity of decoding functions in the context of a specific input stream. The profiling procedure

typically outputs measurements of consumed time or cycle counts of the investigated program running on a specific hardware.

However, profilings are typically available only for single-core architectures. For video decoding systems with strong variations in the runtime complexity, making predictions about the parallel decoding system's behaviour based on a single-core profiling is non-trivial. Algorithmic dependencies, differences in workload and resource limitations result in varying runtime constraints between the processing units. These constraints must be considered.

The simulator introduced in this work overcomes these shortcomings by extending the traditional profiling methods. Our simulator is capable of predicting the runtime behaviour of a virtual multi-core decoder based on information gained from traditional single-core profilers. In contrast to hardware-based development techniques such as HW/SW-Codesign methods [2, 11], the labour- and time-intensive task of implementing a fully functional hardware as well as the decoder software is avoided. This results in a high flexibility when exploring different hardware configurations and decoder partitionings. In this work, we use this method to compare different competing decoder partitionings and architecture configurations against each other to find the best-suited one.

1.3 Paper structure

The remainder of the paper is organized as follows. Section 2 focuses on the fundamental differences between functional and data-parallel splitting approaches. Additionally, it describes the H.264 decoding process and the algorithmic dependencies that counteract the parallelization. In Section 3, we first analyse a single-core profiling of an H.264 decoder. We then derive our high-level simulation methodology for parallel decoders from this analysis. Section 4 gives a detailed overview of the six evaluated parallelization approaches. We provide a structured analysis and describe our motivation for including them in our survey. Results are provided in Section 5. Final conclusions and an outlook on future work are given in Section 6.

2 Parallel H.264 decoding

This section first explains the two options for partitioning a video decoder. It then analyses the influence of the algorithmic structure of an H.264 video decoder on the partitioning process.

2.1 Functional and data-parallel splittings

The memory restrictions of most embedded video coding architectures require the use of small data packages and typically result in a macroblock-based decoding system. In functional partitioned decoding systems, the decoding tasks such as parsing, motion compensation or deblocking are assigned to individual processing cores, typically one task per each processing unit. Macroblocks are processed by one processor after the other. The multiple processors allow starting the next macroblock's decoding tasks before computation of the current macroblock has finished.

This splitting method has the advantage that each processing unit can be optimised for a certain task (e.g. by adding task-specific hardware extensions) and minimal-sized instruction caches. In contrast to data-dependent parallelization, also strongly sequential tasks can be accelerated by this strategy. The disadvantages are an unequal workload balancing and high transfer rates for inter-communication.

As opposed to functional splitting methods, data-parallel systems do not distribute the functions, but the macroblocks among multiple processing units. For efficient parallelization, the macroblocks' core assignment algorithm has to address the following issues:

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

Scalability in parallel systems requires minimal data-dependency between the cores. A compromise between small memory size and data-dependencies can be reached by grouping the macroblocks as described in [25].

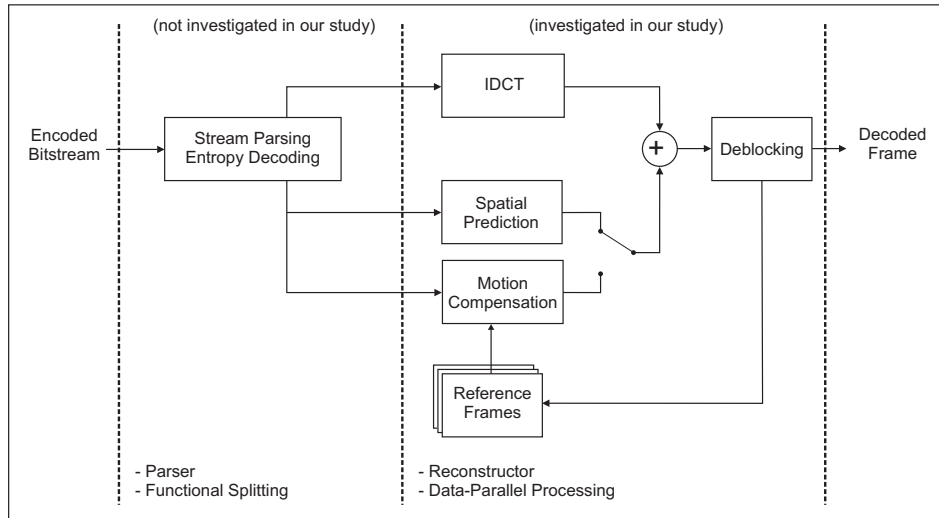


Fig. 1 The H.264 decoding process. Macroblocks run through the decoder pipeline, one after the other

For supporting caching strategies at a more global scale, the groups of macroblocks assigned to one core must be aligned closely together in each frame. By introducing a centralized and constant macroblock assignment for each frame, this global caching issue can be addressed efficiently. Additionally, a constant macroblock assignment allows the parsed macroblocks to be written directly to the input FIFOs of the corresponding reconstructing cores.

In this work, we focus solely on approaches based on data-parallelization. For a high core count, this parallelization paradigm offers better scalability and a more balanced workload than functional splitting approaches [25].

2.2 The H.264 decoder

Figure 1 illustrates the principle of the H.264 decoding process. The decoder can be divided into two fundamentally different parts, the parser and the reconstructor.

The parser includes context calculations for the context-adaptive part of the entropy decoding process, the low-level entropy decoding functions (binary-arithmetic or variable length coding) and the reconstruction of the macroblock's syntax elements such as motion vectors or residual information. This decoder part is highly sequential with short feedback loops at bit- and syntax element-levels. Furthermore, the conditional calculations of the entropy encoder's context information require short branch executions of the executing processor and efficient data access. This makes data-parallel processing hardly usable in these decoding stages and functional splitting is preferable. Our work therefore focuses on the reconstructor module of the H.264 decoder, which provides a wide range of possibilities for exploiting data-parallelization. In our simulations, we assume that stalls cannot be caused by the parser. In other words, an idealized parser always has those macroblocks available that are currently needed in the reconstructor module.

In the reconstruction part, the H.264 decoder uses the parsed syntax elements to generate spatial (=intra) or temporal (=inter) predictions for each macroblock. The residual coefficients are inverse transformed (IDCT) and added to the macroblock's prediction. Finally, an adaptive deblocking filter is applied to the outer and inner edges of the 16×4 pixel sub-blocks of a macroblock in order to remove blocking artefacts.

2.3 Macroblock dependencies

Data-parallel splitting of the decoder's reconstruction module is challenging due to dependencies between spatially as well as temporally neighbouring macroblocks. These dependencies originate from three sources illustrated in Figure 2 and described as follows.

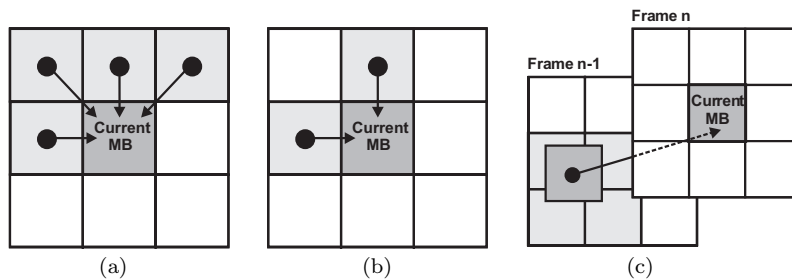


Fig. 2 Macroblock dependencies in H.264 decoding. Arrows mean that the macroblock at the origin of the arrow needs to be processed before decoding the current macroblock. (a) Intra-prediction dependency. (b) Deblocking filter dependency. (c) Inter-prediction dependency

Firstly, in progressive frame coding, the intra-prediction uses unfiltered pixel information from up to four spatially neighbouring macroblocks to predict the current macroblock. These dependencies are depicted in Figure 2a. In general, it is a good option to gather the current macroblock and its reference macroblocks on the same CPU to avoid expensive inter-processor communication for resolving this dependency. For an efficient parallelization, this dependency must be addressed carefully.

Secondly, the deblocking filter imposes additional spatial dependencies. For filtering the outer edges of the current macroblock, up to four pixel rows/columns from the upper and left neighbouring macroblocks are used as filter input. These macroblock dependencies are visualized in Figure 2b. An efficient parallelization method will focus on avoiding that these dependencies have to be resolved across individual processors.

The third and final macroblock dependency arises from the inter-prediction. The inter-prediction reads reference data from macroblocks of previously decoded frames. Obviously, it is required that processing of these reference macroblocks has already been completed before they can be used for inter-prediction of the current macroblock. This results in the temporal dependency depicted in Figure 2c. In fact, the current macroblock can depend on a rather large amount of reference macroblocks. H.264 allows splitting of the current macroblock into small sub-blocks. For each of which, a separate motion vector is computed. In P-slices, each inter-coded macroblock can contain up to 16 motion vectors and point to one reference frame. For bi-directionally predicted macroblocks in B-slices, a maximum of 32 motion vectors and two reference frames is possible.

3 A method for simulating video decoding architectures

In this section, we describe the simulation methodology used for this evaluation. First, we analyse a single-core decoder profile and describe the profiling information that we use as input for the high-level simulation of a parallel decoder. Second, we provide a detailed description of the methodology and the internals of the high-level simulation approach.

3.1 Analysis of a single-core decoder

Macroblock profiling

Traditional dynamic profiling provides runtime information for each decoding function. In our implementation, such profiles are derived from a single-core instruction-set (IS) simulator for the multimedia processor CHILI [22]. The profiler of the CHILI simulator provides the time of each function call and each function return that has occurred during the execution of the decoder (Figure 3). Note that in this work we have used the CHILI profiler for obtaining this information. Nevertheless, this is no restriction of our approach, since profilers for other platforms are, in general, also able to provide this information.

The idea behind recording the time of function calls and returns is to determine the computation time that an individual macroblock spends in the decodings steps depicted in Figure 1. Figure 3 visualizes this idea of mapping each function call to the decoding task of an individual macroblock. For example, we want to determine the computation time that is required for the intra-prediction of the third macroblock in the stream. We know that a function *IntraPrediction* implements the intra-prediction for this macroblock. Let us

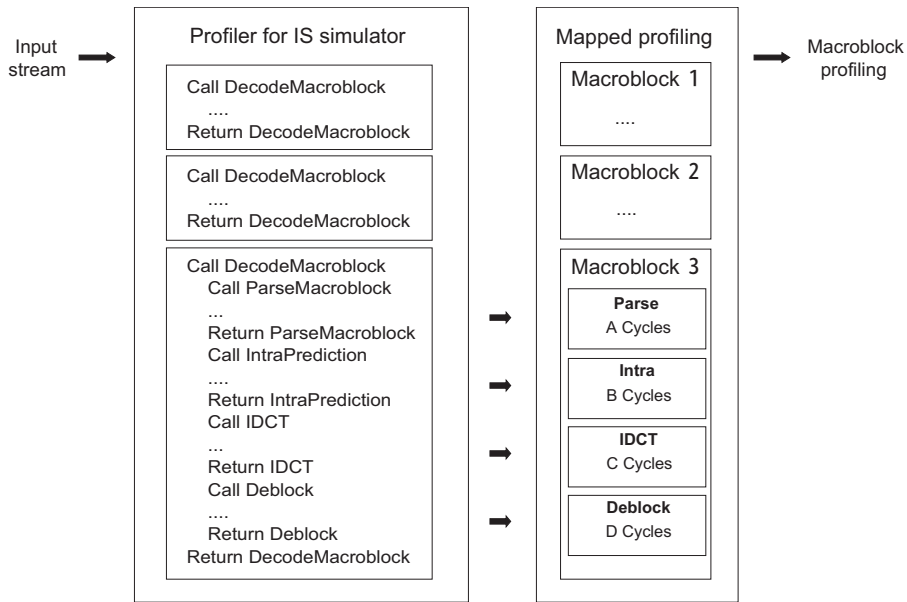






Fig. 3 A dynamic profiler delivers the exact execution times for function calls and returns. We map this information to the major decoding functions of each macroblock and calculate the number of cycles spent in each decoding block. We refer to this macroblock-assigned profiling information as *macroblock profiling*

Table 1 Four test sequences at 720p resolution. Rows 3 and 4 show the resulting bitrates (at 25 frames per second) and the frame resolution in pixels

Sequences			
Bus	Shields	Stockholm	Parkrun
12.6 MBit/s	18.8 MBit/s	25.6 MBit/s	50.8 MBit/s
1280x720 pixels	1280x720 pixels	1280x720 pixels	1280x720 pixels
moderately textured moderate motion	strongly textured moderate motion	moderately textured continuous global motion	strongly textured strong motion
			

assume that in the decoding of a macroblock this function is called exactly once. The processing times for the third invocation of *IntraPrediction* will consequently represent the time that Macroblock 3 spends for the intra-prediction.

However, it should be noticed that this mapping is considerably more complex in practice, since functions are typically called arbitrary times for the decoding of a single macroblock, which depends on the macroblock's coding mode. We have therefore implemented an interpreter that performs the mapping of raw profiling data to meaningful function units.

Decoder software

The profiles are based on a commercial H.264/AVC main profile decoder for embedded architectures. This decoder has been optimised in terms of memory usage and support of DMA transfers. In addition, the regular pixel-based processing functions of the decoder (e.g., interpolation, prediction) have been assembly optimised to make use of the SIMD processor commands. For mobile multimedia applications, using SIMD instructions typically improves the run-time performance in a power-efficient way [16].

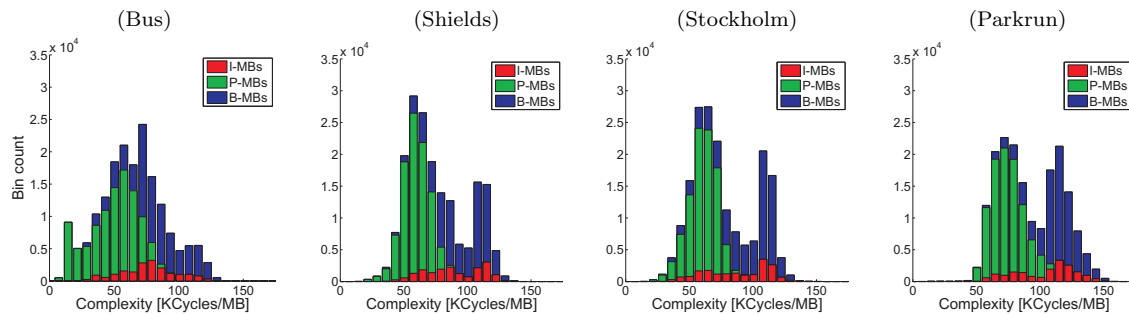


Fig. 4 Dynamic variations in the execution times of individual macroblocks in the H.264 decoding process. Histogram bins plot the number of macroblocks having similar runtimes. The colours indicate the contributions of macroblocks from I-, P- and B-slices to the overall bin counts. Histograms are shown for 50 frames of the four sequences from Table 1. These sequences are IPB coded with the Group of Pictures (GOP) size being 12. It is observed that the runtimes of macroblocks considerably vary within a sequence. This observation is also made when considering I-, P- and B-macroblocks separately

Test sequences

We use four HD test sequences that are shown in Table 1. At a Peak Signal-to-Noise Ratio of approximately 40 db, the four test sequences have bitrates between 12.6 and 50.8 MBit/s. The bitrate is an indicator for the amount of texture and motion occurring in a sequence. The Bus sequence with large, low-textured regions and moderate motion has the smallest bitrate of all test sequences (12.6 MBit/s). Shields and Stockholm are both moderately textured. The horizontal movement in the Shields sequence as well as the zooming operation of the camera in the Stockholm sequence result in a slightly higher motion activity than in the Bus sequence. The higher texturedness and motion activity for the Shields and Stockholm sequences lead to bitrates of 18.8 and 25.6 MBit/s, respectively. The background in the Parkrun sequence contains strong texture patterns and a strong horizontal motion with various temporal occlusions. This leads to the highest bitrate in our test set (50.8 MBit/s).

Dynamic runtime behaviour

We have computed macroblock-based runtime measurements for the test sequences of Table 1. Figure 4 visualizes the macroblocks' decoding complexity for all four sequences. Here, the important point lies in the dynamic behaviour of macroblocks. It can be clearly seen that cycle counts are very different among individual macroblocks. As is also shown in the figure, this observation can still be made when considering the classes of I-, B- and P-macroblocks alone. The pixel information that each macroblock encodes seems to have large influence on its runtime behaviour.

Variations in the execution times of the decoding functions will result in a highly dynamic system when developing parallel decoders. Runtime prediction of a parallel decoding system is not straight-forward in the light of this dynamic behaviour. Moreover, multi-core decoder simulation is complicated by dependencies that exist between individual macroblocks. The methodology described in the following section considers these variations and dependencies when predicting the runtime of a multi-core decoding system.

3.2 Methodology

For a flexible description of the hardware as well as the decoding algorithm, we propose a high-level modelling approach. This approach is implemented by a simulator that we refer to as the *Partition Assessment Simulator (PAS)*. Figure 5 illustrates the basic idea behind the PAS. To use the PAS, one has to give the following inputs. First, a description of a specific video decoding algorithm is required. Second, the user provides an abstract description of the hardware as well as a partitioning of decoder functions onto this hardware. Third, the user presents macroblock profiling results and coding information (e.g. coding type and motion vectors) of a specific stream to the PAS. The task of the PAS is then to simulate the runtime behaviour of the described distributed decoder running on this virtual architecture.

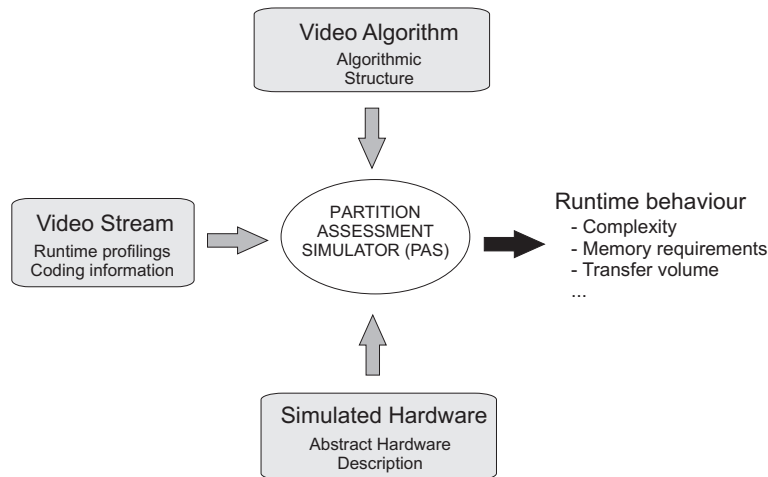


Fig. 5 The Partition Assessment Simulator (PAS). Our simulator combines information about a video codec (e.g., H.264), a virtual multi-core architecture and a video input stream in order to estimate the runtime behaviour of this specific video decoding architecture. Approximations of the decoder’s performance (e.g., complexity, memory accesses, etc.) are accomplished without the need to fully implement the hardware or the decoder software

Modelling the hardware and the decoder software results in high flexibility when doing design space explorations. This includes the evaluation of unknown hardware configurations as well as decoder software partitionings. For example, let us assume that we want to test a new multi-core hardware system. In this case, the system designer just needs to provide an abstract description of the new hardware as well as an updated mapping of decoder tasks to the given processing cores. These small modifications are already sufficient to estimate the runtime behaviour of the decoder software on the new hardware.

Figure 6 illustrates how the PAS works in more detail. The decoding process for three macroblocks is shown in six simulation steps. Core 1 performs the first two decoding tasks (i.e. parsing and entropy decoding) and writes the results to FIFO 1. In this example, the buffer size of FIFO 1 is limited to one macroblock. Core 2 reads the results from FIFO 1 and applies the remaining decoding tasks. Internally, the PAS maintains a list for each processing unit and FIFO buffer. It stores the states and the filling levels of each processing unit and FIFO, respectively. The simulator sequentially processes one macroblock decoding task after the other. It uses the single-core macroblock profiles to determine the time required to finish a specific decoding task. The data-dependencies of a macroblock depend on the macroblock’s coding mode. This coding mode is derived from the macroblocks’ coding information.

For each decoding task, the duration and the dependencies on other tasks and macroblocks are determined by the macroblock profiling and coding information. Firstly, the PAS uses the algorithm mapping to determine the processing unit executing a decoding task. Secondly, it evaluates when this processing unit can start to execute a task. In Figure 6a, this is indicated by the white marker (S). The core’s execution counter is increased by the task’s duration. The black marker (E) indicates the end of a task. At this point, the core has finished the task execution and written its results to the output buffer FIFO 1. At this stage, the PAS does not know when the data is removed by another task and marks the state of FIFO 1 as occupied.

In Figure 6b, Core 2 reads the macroblock data from the FIFO and frees the occupied memory in FIFO 1. During the decoding of a macroblock, functional dependencies between the decoding tasks and data dependencies between the individual macroblocks exist. We have presented macroblock dependencies for the H.264 codec in Section 2.3. These dependencies are determined by the algorithm definition and change according to a macroblock’s specific coding (e.g. intra-prediction poses dependencies on the spatially neighbouring macroblocks). These dependencies obviously change the output of our simulator. We use the macroblocks’ coding information for adapting the dependencies of a macroblock according to its coding. In Figure 6b, a read stall due to data-dependencies is illustrated. Core 2 cannot start its decoding operations simultaneously with Core 1, but has to wait until the required data becomes available in the input buffer. The PAS uses the algorithm description for detecting this read stall. The start of the task execution is delayed automatically.

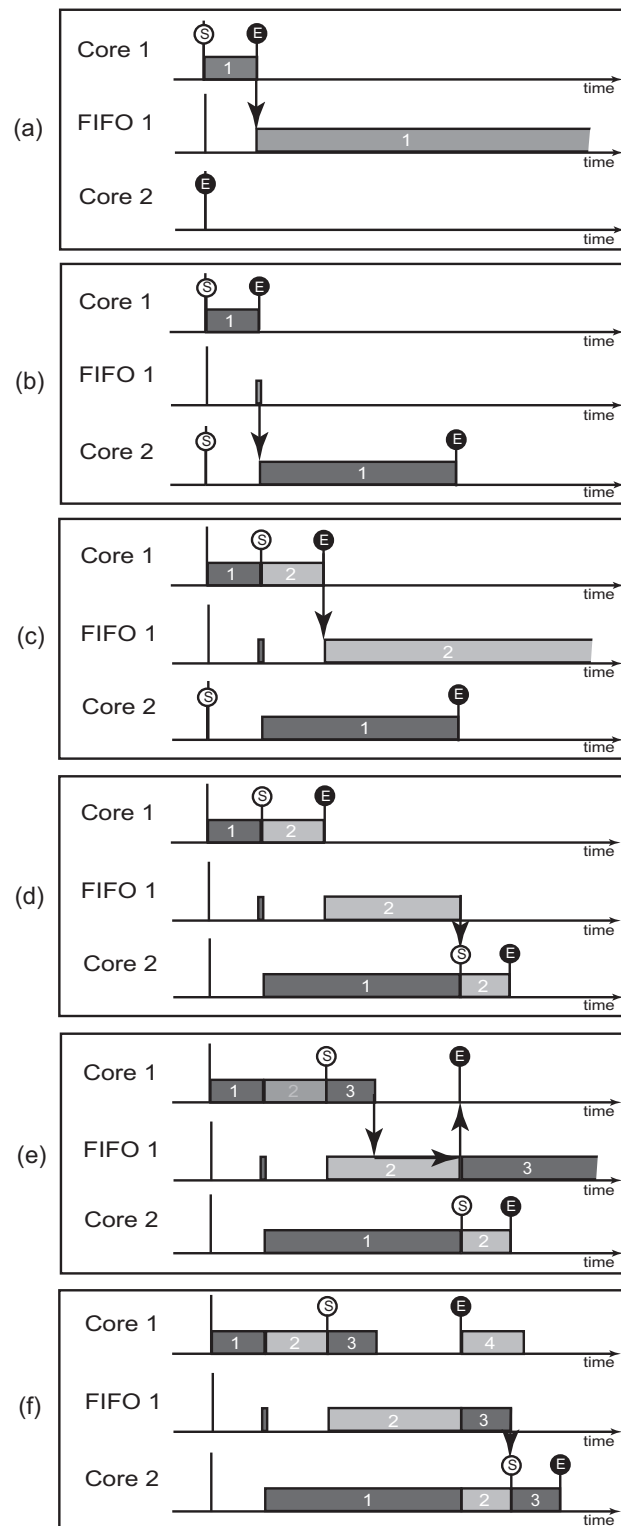


Fig. 6 Visualization of the internal simulation process in the PAS. The figure shows the execution states and buffer levels of two cores and one FIFO, respectively. Three macroblocks are processed on two individual cores. After partially decoding each macroblock, Core 1 writes the results to FIFO 1. For simplicity, the FIFO's maximum size is set to 1 macroblock. Core 2 reads the macroblocks from this buffer and computes the remaining decoding tasks. A detailed explanation is given in the text

In Figure 6c, the second macroblock is executed by Core 1 and written to FIFO 1. After Core 2 has finished its decoding operations for Macroblock 1, it reads Macroblock 2 and executes it (Figure 6d).

Until now we have only considered the impact of computational complexity on our multi-core decoding system. Additionally, the PAS provides flexible interfaces to describe buffer size constraints. For each macroblock task, limitations in the buffer communication (e.g., write stalls due to insufficient buffer sizes) can be considered. The PAS maintains a set of user-defined rules on how the runtime behaviour of the decoder is influenced if buffer scarcity occurs. For example, the parsing task requires a certain amount of free memory for writing its results into an output buffer. We extract this implementation-dependent information from the macroblocks' coding information. We specify how much memory is required and how the decoder's behaviour is influenced by accessing these resources (e.g., "What happens in the case of insufficient buffer?" or "How does the memory access latency influence the decoder's runtime?"). The PAS provides a language for fast specification of buffer and memory resources.

Figure 6e visualizes a case where insufficient buffer is available and a write stall occurs. Since FIFO 1 has a maximum size of 1, Core 1 cannot write the results of decoding Macroblock 3 immediately. It has to wait until Core 2 has read Macroblock 2 and freed the occupied memory in FIFO 1. A write stall occurs in this case. In the following, Macroblock 2 is finished and Core 2 reads Macroblock 3 from FIFO 1 and decodes it. After Macroblock 3 could be written to FIFO 1, Core 1 continues with the next macroblock.

4 Evaluated methods

The approaches considered in this evaluation represent the main groups of data-parallel approaches addressed in the literature. All of them work under the restrictions of typical embedded architectures. These restrictions include low system memory and buffer sizes on the hardware side as well as thread-less and application-dedicated runtime environments on the OS side.

Our evaluation study compares the performance of six different approaches for accomplishing data-parallel splitting of the decoder's reconstructor module. These are (1) a Single-row approach, (2) a Multi-column approach, (3, 4) a blocking and a non-blocking version of a Slice-parallel method, (5) a dynamic version of the non-blocking Slice-parallel approach as well as (6) a Diagonal technique. Details on the benchmarked approaches are provided in the following.

4.1 Single-row approach

The first splitting strategy investigated in our study distributes horizontal lines of macroblocks among different processors as shown in Figure 7. More formally, let N be the number of processors. Processor $i \in \{0, \dots, N-1\}$ is then responsible for decoding the y th row of macroblocks if $y \bmod N = i$.

To illustrate the Single-row (SR) approach, we give an example with two processors on an image divided into 8×8 macroblocks in Figure 8. In this example as well as in the following ones of Figures 11, 13, 14 and 16, we assume that it takes a constant value of 1 unit of time to process each macroblock. It is, however, important to notice that this is a coarse oversimplification. In real video streams, there are large variations in the processing times of individual macroblocks, which makes it difficult to evaluate the goodness of a parallelization approach. In Figure 8, only processor 1 is able to decode macroblocks at time $t = 2$, since all other macroblocks are blocked as a consequence of the dependencies illustrated in Figure 2. After the first two macroblocks of row 1 have been computed, the second core can start processing the first macroblock of the second row, since the dependencies for this macroblock are now resolved ($t = 3$). The next interesting event occurs at $t = 8$, when Core 1 has finished the computation of the first row. Macroblocks of the second row have already been computed and therefore Core 1 can start decoding macroblocks of row 3 that are dependent on their upper neighbours. At time $t = 10$ we obtain the same situation as at $t = 2$ where the first core unlocks the second one. Finally, the whole frame has been decoded at $t = 34$.

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

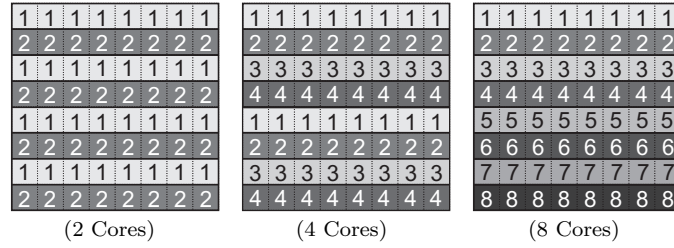


Fig. 7 The Single-row splitting approach. The assignment of processors to macroblocks is shown

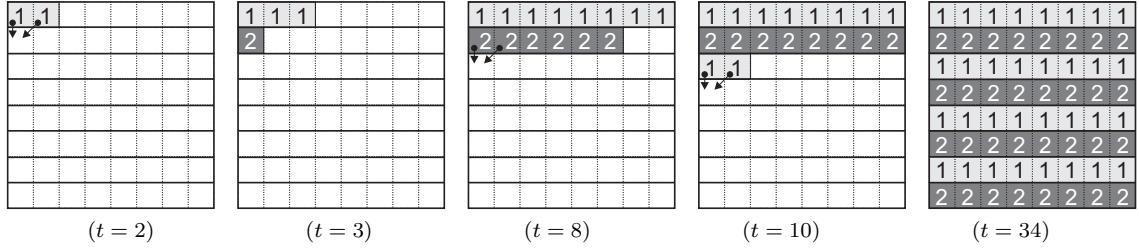


Fig. 8 Example of the Single-row splitting approach used with two cores. Processed macroblocks are shown at different instances of time t . It takes a constant value of 1 unit of time to process a macroblock

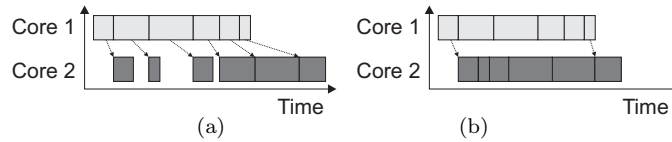


Fig. 9 The number of inter-processor dependencies is crucial for the overall performance of the multi-core system. Rectangles represent macroblocks. A macroblock's width indicates the required processing time. Arrows between two macroblocks mean that processing of the macroblock which the arrow points to can only start after the other macroblock has been decoded. (a) A large number of inter-processor dependencies slows down the system. (b) Due to the low amount of inter-processor dependencies, different running times of individual macroblocks become averaged out. This should improve the overall performance

upper neighbours that are processed by a different Core $i-1$. If processor $i-1$ fails to deliver these macroblocks at the right time, this will immediately produce stalls at Core i . This behaviour is shown in Figure 9a. On the other hand, this strong coupling of CPUs can potentially lead to low buffer requirements.

4.2 Multi-column approach

The second splitting method of our study divides the frame into equally-sized columns as shown in Figure 10. Let w denote the width of a multi-column that is typically derived by dividing the number of macroblocks in a row by the number of processors. Processor i is then responsible for decoding a macroblock of the x th column if $iw \leq x < (i+1)w$. A similar method to partition the image has recently been proposed for the H.264 encoder in [24].

Figure 11 simulates the Multi-column (MC) approach using two processors. Core 1 thereby starts processing the first row of macroblocks until it hits the border to the macroblocks assigned to processor 2 ($t = 4$). Since the dependency for the leftmost macroblock of Core 2 is now resolved, processor 2 can finish decoding its first macroblock at $t = 5$. We obtain a similar situation at $t = 8$. The dependencies of the leftmost macroblock of the second row have been resolved, and Core 2 can therefore continue its work. Decoding of the frame is finally completed at $t = 36$.

The basic idea behind using the Multi-column approach is to obtain a looser coupling of processor dependencies. In fact, the processor assignment borders are significantly reduced in comparison to the Single-row approach. One processor has to wait for the results of another one only at the boundary of its multi-column. Within the multi-column, macroblock dependencies can be resolved on the same processor. This should lead

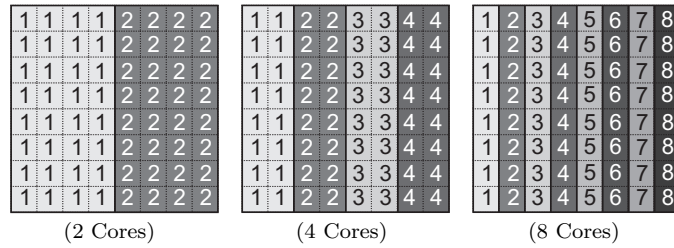


Fig. 10 The Multi-column splitting approach

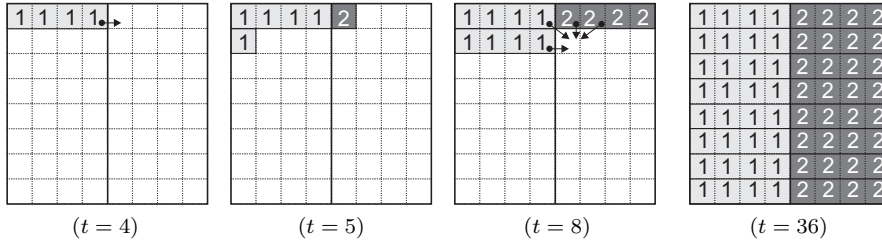


Fig. 11 Example of the Multi-column splitting approach

to reduced inter-processor dependencies and could therefore improve the overall runtime behaviour of the multi-core system as is depicted in Figure 9(b).

4.3 Slice-parallel approach

As a third splitting method, we investigate a 90-degree rotated version of the Multi-column approach that divides the frame into even-sized rows. This method is depicted in Figure 12. Formally spoken, let h denote the height of a multi-row. A macroblock of the y th row is then assigned to Core i if $ih \leq y < (i+1)h$.

The runtime behaviour of the Slice-parallel (SP) approach is illustrated in Figure 13. Here, Core 2 has to wait for a relatively long time ($t = 26$) until the dependencies for its first assigned macroblock are resolved. While the first processor can complete its work on the current frame at $t = 32$, it still takes 26 units of time until the second core finishes processing the remainder of the frame at $t = 58$. In the following, we refer to this approach as the blocking Slice-parallel technique.

The incorporation of the Slice-parallel approach into our benchmark is also motivated by recent work [14] that has presented a non-blocking encoder version of this method. The authors encode their video streams so that slice borders coincide with horizontal lines in the frames. Since neither dependencies introduced by intra-prediction nor dependencies introduced by the deblocking filter occur across slice borders, the multi-rows can be processed completely independent from each other.

Obviously, this non-blocking SP approach (NBSP) requires having full control over the encoder, which will not be the case for many applications. This downside should be considered when interpreting the results for this method in our benchmark. For completeness, we also give an example of this approach in Figure 14. Here, both cores can start processing their assigned macroblocks immediately ($t = 1$) and finish the decoding of the complete frame at $t = 32$.

4.4 Rotating slice-parallel approach

Content-dependent differences in the decoding complexity of the slices will result in an unequal workload of each core [23]. For example, static and low textured frame regions will use coarse macroblock partitions and few residual information. This requires less processing power than for moving and strongly textured regions. If we assign a slice with low complexity constantly to the same core, a drift between this core and the cores working on more complex slices will occur on the long term in the non-blocking NBSP approach. This will result in low system usage and high buffer size requirements for compensating the differences in the workload.

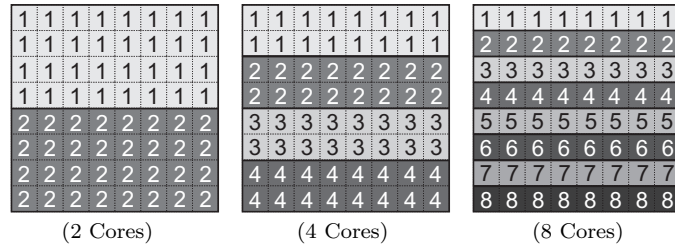


Fig. 12 The Slice-parallel splitting approach

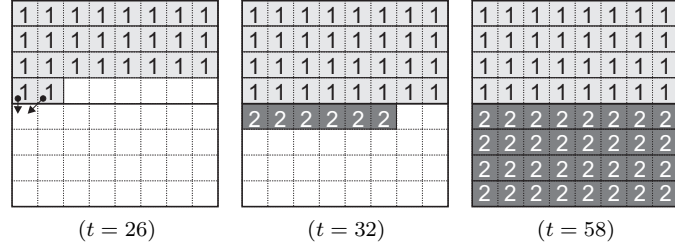


Fig. 13 Example of the Slice-parallel approach in the blocking version

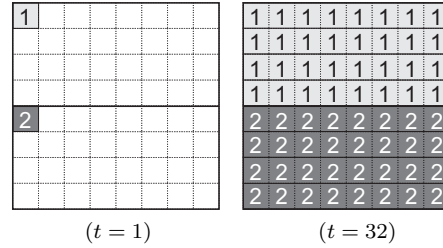


Fig. 14 Example of the Slice-parallel approach in the non-blocking version

For investigating the impact of the sequence content on the NBSP approach’s runtime behaviour, we introduce a modified version of this approach (NBSPr approach). It supports a rotating core assignment depending on the frame number. For example, in a two-core splitting, the first processor would process the upper image part in the first frame, the bottom part in the second frame, then again the upper part and so on. In contrast to the NBSP approach with static assignment, on average, this rotating approach should reach a more balanced system workload.

4.5 Diagonal approach

As a final approach, we include the splitting method illustrated in Figure 15 into our evaluation. This processor assignment is obtained by dividing the first line of macroblocks into equally-sized columns. The assignments for the subsequent lines are then derived by left-shifting the macroblock assignments of the line above. This procedure leads to diagonal patterns.

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

Inclusion of the Diagonal approach into our evaluation is motivated by the macroblock partitioning approaches proposed in [25]. The Diagonal splitting method is regarded as an approach that “respects” the dependency patterns spanned by the intra-prediction and the deblocking filter. (Dependencies are shown in

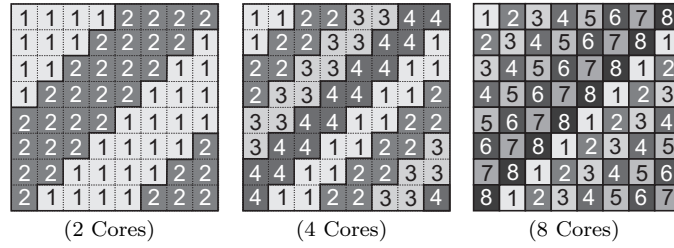


Fig. 15 The Diagonal splitting approach

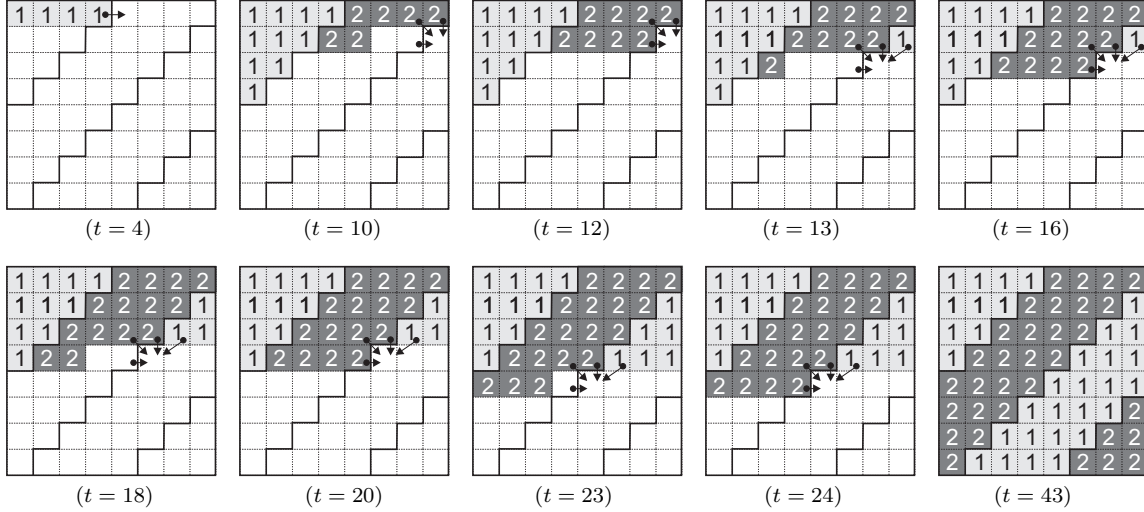


Fig. 16 Example of the Diagonal approach

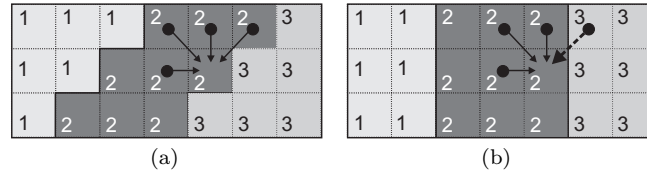


Fig. 17 Processor dependencies in the Diagonal and Multi-column approaches. (a) In the Diagonal method, dependencies for Core 2 originate solely from macroblocks assigned to Core 1. Core 2 therefore never has to wait for Core 3. (b) In the Multi-column approach, macroblocks assigned to Core 2 are also dependent on processor 3, as indicated by the dotted arrow

Figure 2). We illustrate the idea behind the Diagonal splitting method in Figure 17. The figure compares the inter-processor dependencies introduced by Diagonal and Multi-column splitting techniques. The Diagonal method thereby only shows dependencies on macroblocks from its left neighbouring processor, which is in contrast to the Multi-column method that contains dependencies on macroblocks of both neighbouring cores. These reduced inter-processor dependencies could lead to an improved runtime behaviour of the multi-core system.

5 Results

5.1 Run-time complexity

Two major indicators for the efficiency of a multi-core decoding system are the decoder's runtime and the number of data-dependency stalls occurring during the decoding process. A low runtime indicates a high

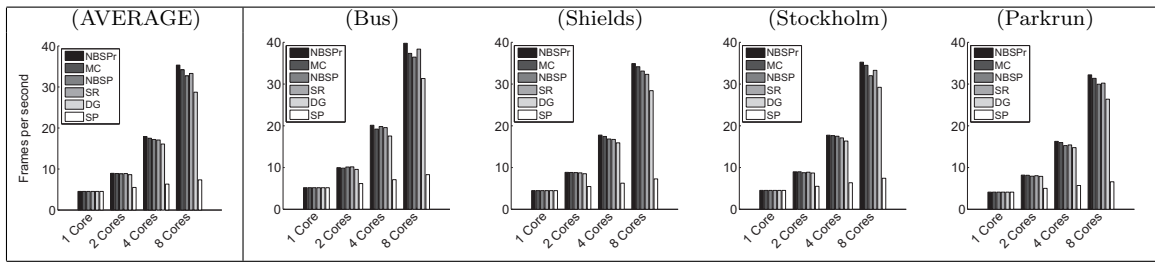


Fig. 18 Frames per second achieved for a system running at 800 MHz. The achieved frame rate for each parallelization approach on average is shown. We considered 1, 2, 4 and 8 cores. The parallelization methods are: the Single-row method (SR), the Multi-column method (MC), the non-blocking versions of the Slice-parallel method (NBSPr/NBSPr), the Diagonal method (DG) and the blocking version of the Slice-parallel method (SP). Additionally, the figure provides detailed runtime results for each of the four 720p sequences

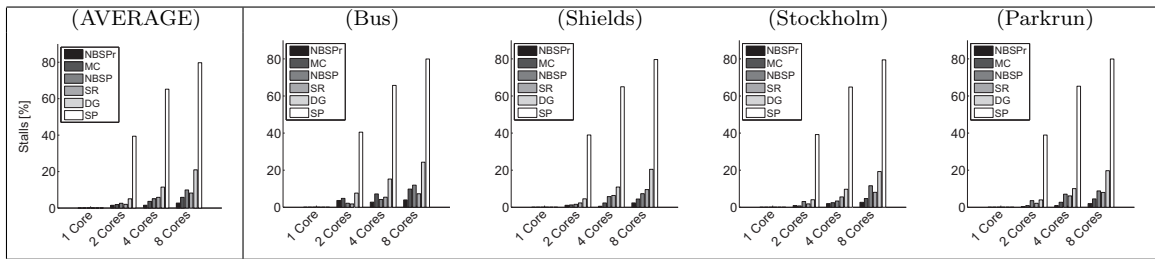


Fig. 19 Stall cycles caused by the data-dependencies between the cores. We plot the percentage of stall cycles in relation to the overall cycles spent for decoding a sequence. The figure shows the average number of stalls over all HD sequences and for the individual HD sequences considering 1, 2, 4 and 8 cores

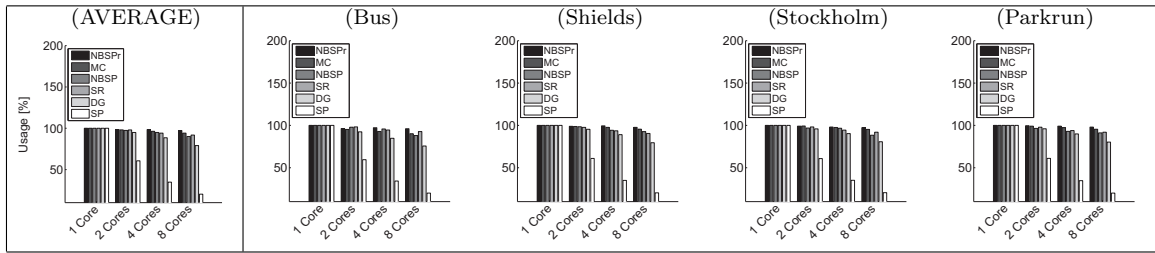


Fig. 20 Average core usage for four test sequences. For the parallelization approaches, the system's average usage is shown for 1, 2, 4 and 8 cores

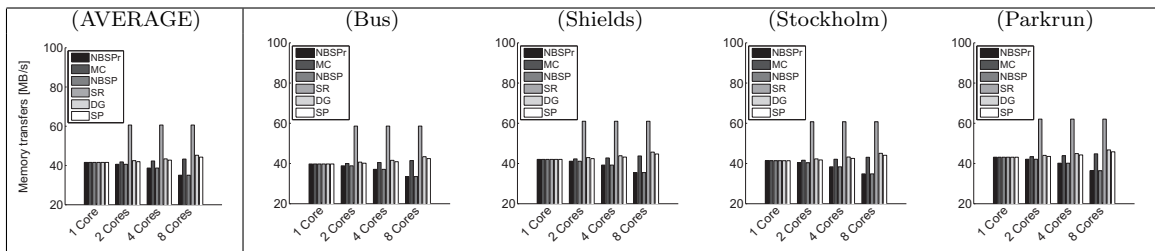


Fig. 21 Data transfer volume for intra and inter reference data and deblocking information, if one macroblock line can be cached in local core memory

Table 2 Results of our evaluation. We evaluate four performance metrics for each test sequence and decoder splitting approach. These are the number of frames decoded in 1 second (FPS), the percentage of stall cycles in relation to the overall cycle count (S), the average core usage (U) given in percent and the data transfer (T) measured in MB/sec. The table’s values are visualized in Figures 18-21. The subscripts denote ranks computed by comparison against the competing splitting approaches. Rank 1 means that the approach is the best-performing one according to the performance metric. The column Avg. Rank is computed as the average of all subscripts within a row. To derive an overall performance ranking of splitting approaches, we sort the table according to Avg. Rank

Algorithm	Avg. Rank ↓	Bus				Shields				Stockholm				Parkrun			
		FPS	S	U	T	FPS	S	U	T	FPS	S	U	T	FPS	S	U	T
1 Core																	
NoSplit	-	5.2	0.0	100.0	39.8	4.5	0.0	100.0	42.0	4.5	0.0	100.0	41.4	4.1	0.0	100.0	43.1
2 Cores																	
NBSPr	1.6	10.0 ₃	3.6 ₃	96.3 ₃	38.8 ₁	8.8 ₁	1.0 ₁	98.9 ₁	41.1 ₁	9.0 ₂	0.9 ₂	99.1 ₂	40.4 ₁	8.2 ₁	0.3 ₁	99.6 ₁	42.1 ₁
MC	2.4	9.9 ₄	4.7 ₄	95.2 ₄	40.0 ₃	8.8 ₂	1.2 ₂	98.7 ₂	42.3 ₃	9.0 ₁	0.7 ₁	99.3 ₁	41.6 ₃	8.1 ₂	0.9 ₂	99.1 ₂	43.4 ₃
NBSP	2.7	10.1 ₂	2.1 ₂	97.8 ₂	38.8 ₁	8.8 ₃	1.6 ₃	98.4 ₃	41.1 ₁	8.8 ₄	3.1 ₄	96.9 ₄	40.4 ₁	7.9 ₄	3.6 ₄	96.4 ₄	42.1 ₁
SR	3.6	10.2 ₁	1.7 ₁	98.2 ₁	58.6 ₆	8.7 ₄	2.3 ₄	97.7 ₄	61.0 ₆	8.9 ₃	1.8 ₃	98.2 ₃	60.8 ₆	8.0 ₃	2.1 ₃	97.9 ₃	62.1 ₆
DG	5.0	9.6 ₅	7.6 ₅	92.4 ₅	40.7 ₅	8.5 ₅	4.5 ₅	95.4 ₅	42.9 ₅	8.7 ₅	4.0 ₅	95.9 ₅	42.3 ₅	7.9 ₅	3.9 ₅	96.0 ₅	44.0 ₅
SP	5.5	6.2 ₆	40.4 ₆	59.5 ₆	40.1 ₄	5.5 ₆	38.9 ₆	61.0 ₆	42.4 ₄	5.5 ₆	39.2 ₆	60.8 ₆	41.8 ₄	5.0 ₆	38.9 ₆	61.1 ₆	43.5 ₄
4 Cores																	
NBSPr	1.0	20.1 ₁	2.7 ₁	97.2 ₁	37.1 ₁	17.8 ₁	0.5 ₁	99.5 ₁	39.2 ₁	17.7 ₁	1.9 ₁	98.0 ₁	38.4 ₁	16.3 ₁	0.9 ₁	99.1 ₁	40.2 ₁
NBSP	2.5	19.8 ₂	4.2 ₂	95.8 ₂	37.1 ₁	16.8 ₃	7.3 ₃	94.2 ₃	39.2 ₁	17.5 ₃	3.3 ₃	96.6 ₃	38.4 ₁	15.3 ₄	7.0 ₄	92.9 ₄	40.2 ₁
MC	2.6	19.2 ₄	7.1 ₄	92.8 ₄	40.5 ₃	17.5 ₂	4.4 ₂	97.7 ₂	42.8 ₃	17.7 ₁	2.5 ₂	97.5 ₂	42.1 ₃	16.0 ₂	2.7 ₂	97.3 ₂	43.8 ₃
SR	4.1	19.6 ₃	5.4 ₃	94.6 ₃	58.6 ₆	16.7 ₄	9.5 ₄	93.6 ₄	61.0 ₆	17.1 ₄	5.6 ₄	94.4 ₄	60.8 ₆	15.4 ₃	6.1 ₃	93.9 ₃	62.1 ₆
DG	5.0	17.6 ₅	15.2 ₅	84.8 ₅	41.6 ₅	15.9 ₅	20.5 ₅	89.1 ₅	43.9 ₅	16.3 ₅	9.7 ₅	90.3 ₅	43.2 ₅	14.8 ₅	10.0 ₅	89.9 ₅	44.9 ₅
SP	5.5	7.1 ₆	65.0 ₆	34.3 ₆	40.9 ₄	6.3 ₆	79.6 ₆	35.0 ₆	43.2 ₄	6.4 ₆	64.8 ₆	35.2 ₆	42.5 ₄	5.7 ₆	65.3 ₆	34.7 ₆	44.3 ₄
8 Cores																	
NBSPr	1.0	39.8 ₁	3.9 ₁	96.1 ₁	33.5 ₁	34.9 ₁	2.3 ₁	97.7 ₁	35.5 ₁	35.2 ₁	2.6 ₁	97.3 ₁	34.8 ₁	32.2 ₁	1.9 ₁	98.0 ₁	36.4 ₁
MC	2.4	37.3 ₃	9.7 ₃	90.2 ₃	41.4 ₃	34.2 ₂	4.4 ₂	95.6 ₂	43.7 ₃	34.5 ₂	4.7 ₂	95.3 ₂	43.1 ₃	31.4 ₂	4.5 ₂	95.2 ₂	44.8 ₃
NBSP	3.1	36.5 ₄	11.9 ₄	88.1 ₄	33.5 ₁	33.1 ₃	7.3 ₃	92.7 ₃	35.5 ₁	32.0 ₄	11.6 ₄	88.4 ₄	34.8 ₁	30.0 ₄	8.8 ₄	91.2 ₄	36.4 ₁
SR	3.8	38.4 ₂	7.3 ₂	92.7 ₂	58.6 ₆	32.3 ₄	9.5 ₄	90.5 ₄	61.0 ₆	33.3 ₃	8.0 ₃	91.9 ₃	60.8 ₆	30.2 ₃	8.0 ₃	92.0 ₃	62.1 ₆
DG	5.0	31.3 ₅	24.3 ₅	75.7 ₅	43.4 ₅	28.4 ₅	20.5 ₅	79.5 ₅	45.7 ₅	29.2 ₅	19.3 ₅	80.7 ₅	45.1 ₅	26.4 ₅	19.7 ₅	80.3 ₅	46.7 ₅
SP	5.5	8.3 ₆	79.9 ₆	20.1 ₆	42.4 ₄	7.3 ₆	79.6 ₆	20.4 ₆	44.7 ₄	7.4 ₆	79.5 ₆	10.5 ₆	44.1 ₄	6.6 ₆	79.9 ₆	20.1 ₆	45.8 ₄

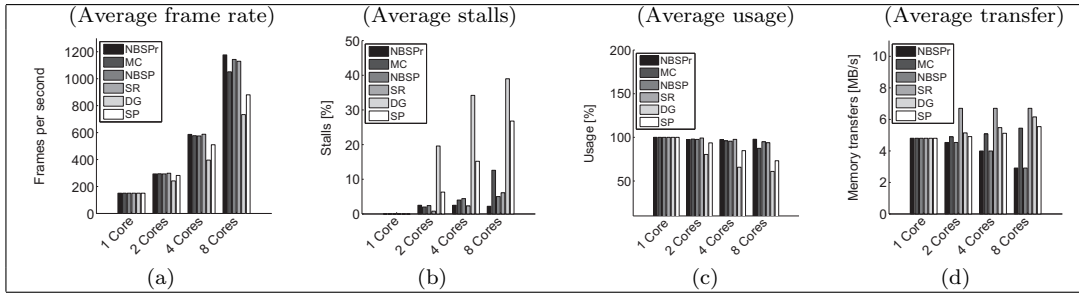


Fig. 22 Average frame rate, stall cycles and usage for a system running at 800 MHz and decoding sequences with CIF resolution. The sequences included in these calculations are *Barcelona*, *Bus*, *Foreman*, *Flowergarden* and *Mobile*. We calculated (a) the average framerate, (b) the percentage of stall cycles, (c) the average usage and (d) the memory transfer rate for 1, 2, 4 and 8 cores

system decoding performance, while the number of stalls provides an estimate on how efficiently the system’s computational resources are used.

Figures 18 provides runtime estimates for the decoder performance in frames per second for a system running at 800 MHz. Additionally, Figure 19 shows the percentage of stall cycles occurring during the overall decoding process for the six investigated approaches. We provide the average core usage of the decoders using different approaches and core counts in Figure 20. Additionally, Table 2 provides detailed performance numbers and shows the average performance results over all HD sequences considering 1, 2, 4 and 8 cores. It ranks the approaches according to their runtime performance and memory transfer rates.

In Figure 18, it is observed that the SR method and MC method as well as the NBSP approach perform considerably better than the DG and the blocking SP approaches. Despite the usage of depending partitions, the MC approaches even outperform the NBSP approach without data-dependencies between the cores.

We found that the constant core assignment to frame regions and the runtime differences between these regions result in an unbalanced workload distribution for the NBSP approach. According to our results, approaches working on a finer granularity level and alternating the cores over the frame more frequently (e.g. the SR approach) adapt better to local complexity peaks.

For comparing the scalability of the parallel approaches for different resolutions, we have included five common CIF sequences in our evaluation. These are the *Barcelona*, the *Bus*, the *Flowergarden*, the *Foreman*

and the Mobile sequences which have been encoded using the same encoding parameters as for the four HD sequences. Figure 22 shows the average performance results in terms of runtime, stalls, core usage and memory transfer rates for these sequences. For CIF resolution, the NBSP and the SR show similar runtime performance. With increasing number of cores the NBSP approach has the advantage of less dependencies between the splitting partitions which results in a better runtime performance.

The impact of the rotating order in the NBSPr approach can be observed for all core counts and sequences. However, architectures with high core counts tend to benefit stronger from this rotating order than systems with few cores. A good ranking of the NBSPr approach for the CIF and the HD sequences indicates that rotating the slices increases the runtime performance independent of the resolution.

Evaluating only the encoder-independent approaches shows a similar scalability for the SR and the MC approaches for HD decoding. For most of these sequences and core counts, the MC approach outperforms the SR approach. For the SR approach, the different complexity between adjacent lines accumulates and results in a faster increase of the stall cycles for increasing core counts. For decoding sequences at CIF resolution, the SR approach seems to be more suitable. The simulations show that the performance for the MC approach with 8 cores decreases more strongly compared to the systems with fewer cores. At CIF resolution with a frame width of 22 macroblocks, no partitioning into equally sized vertical partitions is possible for more than two cores. Additionally, the column width decreases with each additional core and results in a more frequent inter-communication between the cores.

The diagonal (DG) approach shows relatively poor performance, especially for high core counts and small resolutions. For each additional core, the DG approach creates two additional partitions with dependencies on other cores' partitions. Adding a new core to the system raises the number of dependencies faster than for the other approaches and results in a higher number of stalls. For CIF resolution and high core counts a stronger decrease in performance than for HD resolution is observed.

For the SP approach, the coarse horizontal splitting and the dependencies between each partition and its upper partitions results in high stall counts and the worst runtime of all approaches. With increasing resolutions, each core has to wait longer until the previous partition is finished. Hence, the efficiency of the SP approach decreases with increasing resolution.

5.2 Inter-communication

Memory transfers to and from the external DRAM and between the cores' local memories are expensive in terms of power consumption and transfer times. In a video decoding system, these transfers are strongly influenced by the core inter-communication and the loading of reference data and deblocking pixels. Depending on the size of the cores' local memories and how efficiently the parallelization approach exploits locality, macroblock information can be kept on a core and reused for the decoding of future macroblocks. In our work, we assume that exactly one macroblock line of intra-prediction and deblocking data can be kept in the core memory of each core. The resulting transfer rates for HD and CIF resolution videos are shown in Figure 21 and Figure 22d, respectively.

A strong difference between the approaches can be observed for the HD as well as for the CIF sequences. The SR approach only exploits data locality between horizontally neighbouring macroblocks. The fine horizontal splitting into macroblock lines results in the highest transfer rate of all approaches and represents the worst case for horizontal splitting approaches in terms of memory transfers. Note that for two or more cores the transfer rate of the SR approach is constant. The alternating processing order of the macroblock lines requires the transfer of vertical reference data and deblocking information between the cores.

For the MC approach, a smaller amount of data has to be exchanged between the cores and efficient data reuse is possible. This results in the lowest transfer rate of all encoder-independent approaches and even outperforms the blocking SP approach at HD as well as CIF resolution. Increasing the core counts only results in a slight transfer increase for the MC and the SP approaches. For the NBSP and the NBSPr approaches, the data communication even decreases for high core counts, since the number of independent frame partitions increases. However, this results in blocking artefacts at slice borders and a less efficient prediction.

The transfer rates for the DG approach are slightly worse than for the MC and the SP approaches for all core counts. These results can be observed for HD as well as CIF resolutions. The two additional partitions for each additional core result in new data dependencies to neighbouring partitions and a stronger increase of the data transfers when raising the number of cores.

6 Conclusions

In this study, we have evaluated six data-parallel approaches for decoding H.264 video streams in environments with strong resource restrictions. We compare the approaches against each other and investigate the impact of raising the core count on the decoder's runtime, core usage and transfer rates. For accomplishing this evaluation, we have developed a high-level simulator. It is capable of estimating the runtime behaviour of a virtual multi-core decoder based on single-core profilings.

The runtime of each parallelization approach is strongly influenced by the frame partitions' sizes and shapes. Large and dependency-minimizing partitions cause less inter-communication between the cores and a lower runtime. Additionally, a static assignment of each core to constant frame regions makes an approach highly sensitive to local complexity peaks and can result in an unequal workload balancing. Assigning each core to multiple and spatially separated frame regions (e.g. the SR approach) or rotating the assignment order between frames can compensate these complexity variations more effectively. A rotating assignment has the advantage that it can reduce dependencies by maintaining large partitions and process diverse frame regions on each core.

For approaches working on larger partitions, it is generally easier to exploit locality. This reduces transfer rates, since more spatially close neighbouring macroblocks are processed on the same core and efficient data reuse is possible. Reusing the data from horizontal neighbouring macroblocks is easily possible by keeping a single macroblock's prediction and deblocking information in the local cache. Vertical caching requires that the partitions' width is smaller than the number of locally storable macroblocks. For approaches based on horizontal splitting, this results in the highest requirements on local memories.

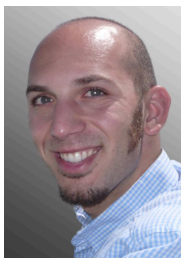
In future work, we are planning to investigate the efficiency of parallel coding approaches for different H.264 coding tools and GOP structures. Other video standards such as VC-1 and AVS shall be included in further evaluations.

Acknowledgements This work has been supported by the Austrian Federal Ministry of Transport, Innovation, and Technology under the FIT-IT project VENDOR (Project nr. 812429). Michael Bleyer would like to acknowledge the Austrian Science Fund (FWF) for financial support under project P19797.

References

1. Ball T, Larus JR (1994) Optimally profiling and tracing programs. *ACM Trans on Programming Languages and Systems* 16(4):1319–1360
2. Cesario WO, Lyonnard D, Nicolescu G, Paviot Y, Yoo S, Jerraya AA, Gauthier L, Diaz-Nava M (2002) Multiprocessor SoC platforms: A component-based design approach. *IEEE Journal of Design and Test of Computers* 19(6):52–63
3. Chen TW, Huang YW, Chen TC, Chen YH, Tsai CY, Chen LG (2005) Architecture design of H.264/AVC decoder with hybrid task pipelining for high definition videos. In: *Proc. of the IEEE Int. Symposium on Circuits and Systems*, pp 2931–2934
4. Chen YK, Tian X, Ge S, Girkar M (2004) Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures. In: *Proc. of the 18th Int. Parallel and Distributed Processing Symposium*, vol 1, pp 63–72
5. Cmelik B, Keppel D (1994) Shade: a fast instruction-set simulator for execution profiling. In: *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp 128–137
6. Faichney J, Gonzalez R (2001) Video coding for mobile handheld conferencing. *Journal on Multimedia Tools and Applications* 13(2):165–176
7. Graham SL, Kessler PB, McKusick MK (1982) gprof: a call graph execution profiler. In: *Proc. of the SIGPLAN Symposium on Compiler Construction*, pp 120–126
8. Gulliver SR, Ghinea G, Patel M, Serif T (2007) A context-aware tour guide: User implications. *Mobile Information Systems* 3(2):71–88
9. ITU-T, ISO/IEC (2005) Advanced video coding for generic audiovisual services (ITU Rec. H.264 — ISO/IEC 14496-10). ITU-T and ISO/IEC
10. Jeon J, Kim H, Boo G, Song J, Lee E, Park H (2000) Real-time MPEG-2 video codec system using multiple digital signal processors. *Journal on Multimedia Tools and Applications* 11(2):197–214

11. Knudsen PV, Madsen J (1996) Pace: A dynamic programming algorithm for hardware/software partitioning. In: Proc. of the Int. Workshop on Hardware-Software Co-Design, pp 85–92
12. Malik S, Martonosi M, Li YTS (1997) Static timing analysis of embedded software. In: Proc. of the 34th ACM/IEEE Design Automation Conference, pp 147–152
13. Meenderinck C, Azevedo A, Juurlink B, Alvarez M, Ramirez A (2008) Parallel scalability of video decoders. *Journal of Signal Processing Systems*
14. Moriyoshi T, Miura S (2008) Real-time H.264 encoder with deblocking filter parallelization. In: IEEE Int. Conference on Consumer Electronics, pp 63–64
15. Nachtergaele L, Cattoor F, Kapoor B, Janssens S, Moolenaar D (1996) Low power storage exploration for H.263 video decoder. In: Proc. of the IX. Workshop on VLSI Signal Processing, pp 115–124, DOI 10.1109/VLSISP.1996.558310
16. Paver N, Khan M, Aldrich B (2006) Optimizing mobile multimedia using SIMD techniques. *Journal on Multimedia Tools and Applications* 28(2):221–238
17. Puschner PP, Koza C (1989) Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems* 1(2):159–176
18. Ravasi M, Mattavelli M (2003) High-level algorithmic complexity evaluation for system design. *Journal of Systems Architecture* 48(13-15):403–427
19. Ravasi M, Mattavelli M (2005) High abstraction level complexity analysis and memory architecture simulations for multimedia algorithms. *IEEE Trans on Circuits and Systems for Video Technology* 15(5):673–684
20. Rodriguez A, Gonzalez A, Malumbres M (2006) Hierarchical parallelization of an H.264/AVC video encoder. In: Proc. of the Int. Symposium on Parallel Computing in Electrical Engineering, pp 363–368
21. Schöffmann K, Fauster M, Lampl O, Böszörményi L (2007) An evaluation of parallelization concepts for baseline-profile compliant H.264/AVC decoders. In: Proc. of the Euro-Par 2007, pp 782–791
22. Seitner F, Meser J, Schedelberger G, Wasserbauer A, Bleyer M, Gelautz M, Schutti M, Schreier R, Vaclavik P, Krottendorfer G, Truhlar G, Bauernfeind T, Beham P (2008) Design methodology for the SVENm multimedia engine. In: Proc. of the Austrochip 2008, poster presentation
23. Seitner FH, Schreier RM, Bleyer M, Gelautz M (2008) A high-level simulator for the H.264/AVC decoding process in multi-core systems. In: Proc. of the SPIE, Multimedia on Mobile Devices, vol 6821, pp 5–16
24. Sun S, Wang D, Chen S (2007) A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition. In: Proc. of the 3rd Int. Conference on High Performance Computing and Communications, pp 577–585
25. van der Tol EB, Jaspers EG, Gelderblom RH (2003) Mapping of H.264 decoding on a multiprocessor architecture. In: Proc. of the SPIE, vol 5022, pp 707–718
26. Wang SH, Peng WH, He Y, Lin GY, Lin CY, Chang SC, Wang CN, Chiang P (2003) A platform-based MPEG-4 advanced video coding (AVC) decoder with block-level pipelining. In: Proc. of the 2003 Joint Conference of the 4th Int. Conference on Information, Communications and Signal Processing and the 4th Pacific Rim Conference on Multimedia, vol 1, pp 51–55
27. Witchel E, Rosenblum M (1996) Embra: fast and flexible machine simulation. In: Proc. of the ACM SIGMETRICS Int. Conference on Measurement and Modeling of Computer Systems, pp 68–79
28. Zhao Z, Liang P (2006) A highly efficient parallel algorithm for H.264 video encoder. In: Proc. of the 31st IEEE Int. Conference on Acoustics, Speech, and Signal Processing, vol 5, pp 489–492



Florian H. Seitner received his diploma degree (M.S.) in Computer Science from the Vienna University of Technology in 2006. He is currently employed as a project assistant at the Vienna University of Technology where he works on his Ph.D. degree. His scientific interests include video coding, parallel software design and human gesture analysis from images and videos.



Michael Bleyer received his M.S. and Ph.D. degrees in Computer Science from the Vienna University of Technology in 2002 and 2006, respectively. He is currently employed as a post-doc researcher at the Vienna University of Technology. His research interests include stereo matching, optical flow computation, combinatorial optimization and video coding.



Margrit Gelautz received her PhD degree in Computer Science from Graz University of Technology, Austria, in 1997. She worked on optical and radar image processing for remote sensing applications during a postdoctoral stay at Stanford University. In her current position as an associate professor at Vienna University of Technology, her research focuses on image and video analysis for multimedia applications. She is Women in Engineering (WIE) officer of IEEE Austria Section.



Ralf M. Beuschel was born in Friedrichhafen, Germany in 1974. He received the diploma degree (M.Eng.) in electrical engineering (systems engineering) from the University of Ulm in 2000 with a best thesis award donated by the VDE/VDI. From 2000 to 2006 he was a research fellow at the Microelectronics Department at the University of Ulm doing research in the field of low-delay video coding algorithms and real-time DSP implementations. Afterwards he was with the Institute of Software Technology and Interactive Systems of the TU Vienna for two years. In early 2009 he hired at DSP-Weuffen in Amtzell, Germany in the position of a Senior System Designer for DSP video applications. He is member of IEEE and the German TV and Cinema Technology Society (FKTG).