A high-level simulator for the H.264/AVC decoding process in multi-core systems

Florian H. Seitner, Ralf M. Schreier, Michael Bleyer, Margrit Gelautz

Institute for Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstrasse 9-11/188/2, A-1040 Vienna, Austria

ABSTRACT

ABSTRACT H.264 as a new-generation video coding algorithm is becoming increasingly important for international broadcasting standards such as DVB-H and DMB. In comparison to its predecessors MPEG-2 and MEPG-4 SP/ASP, H.264 achieves improved compression efficiency at the cost of increased computational complexity. Real-time execution of the H.264 decoding process poses a large challenge on mobile devices due to low processing capabilities. Multi-core systems provide an elegant and power-efficient solution to overcome this performance limitation. However, efficiently distributing the video algorithm among multiple processing units is a non-trivial task. It requires detailed knowledge about the algorithmic complexity, dynamic variations and inter-dependencies between functional blocks. The objective of this paper is an investigation on the dynamic behavior of the H.264 decoding process and on the interaction between the main decoding tasks in the context of multi-core environments. We use an H.264 decoder model to investigate the efficiency of a decoding system under various conditions (e.g. different FIFO buffer sizes, bitstreams, coding features and bitrates). The gained insights are finally used to optimize the runtime behavior of a multi-core decoding system and to find a good trade-off between core usage and buffer sizes.

Keywords: Multi-core, video, decoding, H.264/AVC, modeling

1. INTRODUCTION

Increasing the coding efficiency of video codecs with the common combination of temporal prediction and lossy transform coding is basically a matter of reducing the remaining redundancy in the data streams. In the H.264 standard,[1] this goal has been achieved by means of more advanced pixel processing algorithms (e.g. quarter-pixel motion estimation) as well as using more sophisticated algorithms for predicting syntax elements from neighboring macroblocks (e.g. context adaptive VLC). However, the advanced coding tools result in significantly increased CPU and memory loads on the encoder as well as the decoder. The high computational demands pose a challenge for practical H.264 implementations in environments of limited processing power such as mobile devices. Understanding the runtime behavior of the H.264 decoder is therefore essential for meeting the desired performance requirements on the underlying platform.

If the computational requirements of a video algorithm cannot be met with a single processing unit, multi-core systems often provide an elegant and power-efficient alternative. Unfortunately, efficiently distributing the video algorithm among multiple processing units is a non-trivial task. It requires detailed knowledge about the algorithmic complexity and inter-dependencies between functional blocks. The most severe problem, however, is that there are strong dynamic variations in runtime on the basic level of the H.264 decoding process, namely the level of macroblocks. We go into more detail on this dynamic behavior in the following.

Roughly spoken, an H.264 stream can be regarded as a sequence of compressed macroblocks. In the decoding process, each macroblock is sent through the decoder pipeline one after the other in order to derive the uncompressed video information. When looking at the computation time consumed by each macroblock, it is observed that there are large variations. To illustrate this phenomenon, we have measured the runtime of individual macroblocks in six standard sequences and plot corresponding histograms in Figure 1. (A more in-depth

Further author information: E-mail: {seitner, schreier, bleyer, gelautz}@ims.tuwien.ac.at

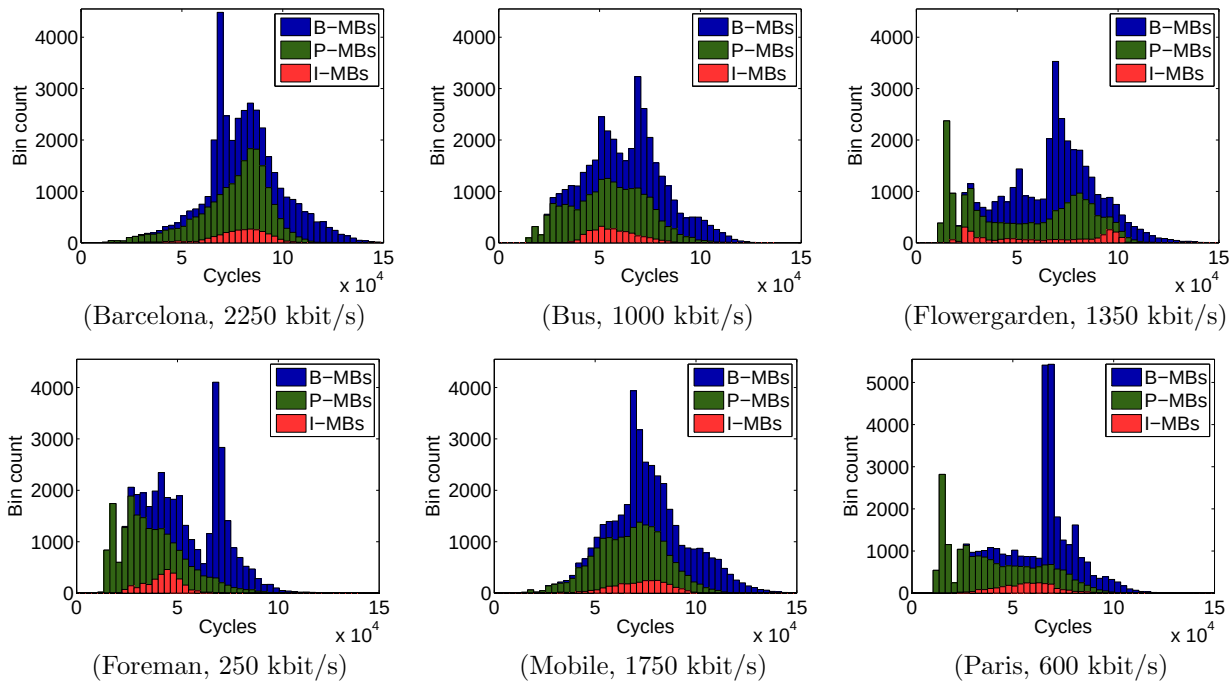

Figure 1. Dynamic variations in the execution times of individual macroblocks in the H.264 decoding process. Histogram bins plot the number of macroblocks having similar runtimes. The colors indicate the contributions of I-, P- and B-macroblocks to the overall bin counts. Histograms are shown for six IPB coded sequences of 100 frames with Group of Pictures (GOP) sizes being 13. It is observed that the runtimes of macroblocks significantly vary within a sequence due to different image content. This observation is also made when considering I-, P- and B-macroblocks separately.

interpretation of the histograms is given at a later point in this paper.) Not very surprisingly, the figure shows that variations in execution times stem to a large extent from the coding mode that is used for the current video frame. Macroblocks of an I-frame show different runtime characteristics from those of predicted P- or B-frames. However, significant differences are also observed within these three classes due to the encoded video content. For example, the execution times for macroblocks of predicted frames strongly depend on their partitioning modes and sub-pixel motion vector positions. We can conclude that the overall runtime of the decoder strongly depends on the content of the encoded video material.

When implementing a multi-core system, a straightforward way for obtaining a functional splitting of the H.264 decoder is to use the results of H.264 profilings such as presented in various papers.[2–4] These benchmark approaches decode whole video sequences and then report the execution time spent in each functional block of the decoder as a percentage of the overall runtime. To determine a splitting point, for example, in a dual-core system, one can exploit the benchmark results to identify a decoder function so that approximately 50% of the execution time is consumed before and another 50% after this function. The problem of this strategy, however, is that it does not account for the above described dynamics in runtimes of individual macroblocks. More precisely, there will be macroblocks that consume significantly more time on the first processor than on the second one and vice versa. Consequently, this strategy leads to unbalanced load distributions on the two processors. Investigating profiling results on the macroblock level is therefore more promising.

A first step towards macroblock-based profiling of an H.264 decoder has recently been taken by Kalva and Furht.[5] The goal of the authors is to provide an accurate prediction of the overall execution time of a specific video stream. The authors build a categorization of macroblocks and measure the frequency of occurrence for each macroblock type in the profiled video stream. In a preceding paper,[6] we have also suggested to perform H.264 profiling on the macroblock level. We have presented a set of models that can accurately predict the runtime of a macroblock in the decoder given a small set of the macroblock's features as input.

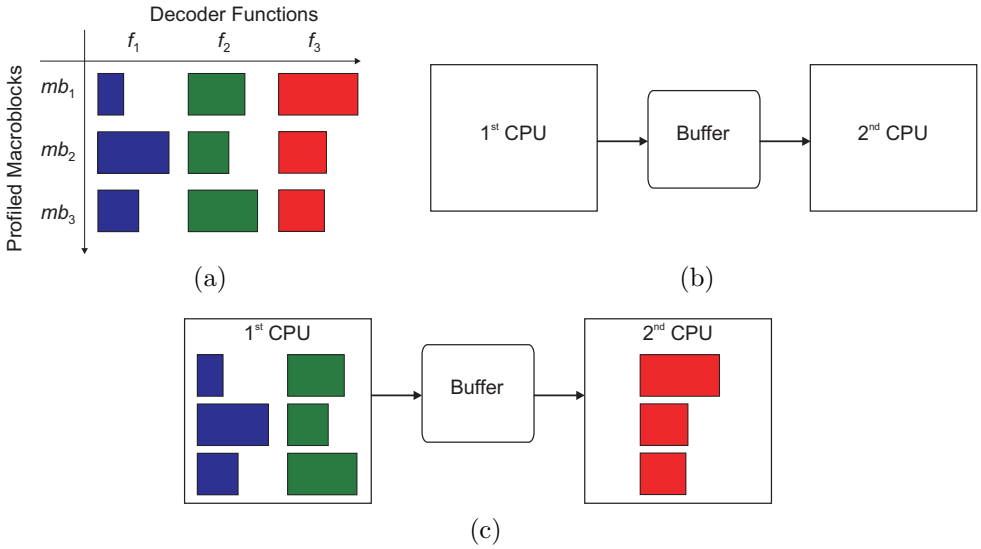

Figure 2. Concept of the simulator. (a) Profiling data. The amount of execution time is proportional to the sizes of rectangles. (b) Underlying hardware. (c) Simulation of a splitting that maps f_1 and f_2 to the first core and f_3 to the second one. Explanation is given in the text.

In this paper, we present a high-level simulator for multi-core implementations of the H.264 decoder. The basic idea behind this simulator is illustrated in Figure 2 and explained as follows. We generate profiling information for each major functional block of the H.264 decoder on a macroblock level. This is illustrated in Figure 2a where execution times for macroblocks are computed for various decoder functions. This profiling step extends our work on macroblock profiles. In a previous paper,[6] we have divided the decoder into two functions, namely a parser and a reconstructor function. In this paper, we break down these two functions into their sub-components in order to derive the profiling precision required for our simulator.

Our simulator operates on multi-core hardware configurations with an arbitrary number of processors. For simplicity, we have chosen a dual-core system in Figure 2b as an example for the underlying hardware. In the figure, the two CPUs are connected via a buffer in order to exchange data. In this configuration, the second CPU has to wait for data written into the buffer by the first CPU. Once the buffer is full, the first CPU has to wait for the second CPU to consume data. It is one goal of our simulator to allow for identifying decoder splittings that minimize the times that the CPUs spend in the waiting mode.

The profiling information is then used to run simulations on the given hardware configuration. In Figure 2c, we test the performance of a decoder partitioning that maps decoder functions f_1 and f_2 to the first core and f_3 to the second CPU. To measure the goodness of this splitting, we simulate the behavior of the resulting multi-core system so that the first CPU stops if the buffer is full and the second one if the buffer is empty. Our simulator then computes the total runtime of the multi-core system for a given video sequence. In this paper, we use our simulator to compare the efficiency of two competing splitting methods of the decoder. In the example of the figure, we can compare the efficiency of the illustrated splitting against a splitting that assigns function f_1 to the first CPU and f_2 and f_3 to the second processor. Moreover, as we will also show in the paper, our simulator can be used for identifying hardware components required for meeting a given performance condition. In this context, we use the simulator for finding an optimal size of the buffer that interconnects the CPUs.

Summarizing the above, our contributions are:

- We investigate the dynamic behavior of the H.264 decoding tasks in a systematic way.
- We develop a simulator for mapping the H.264 decoding process onto various hardware architectures in a flexible way.

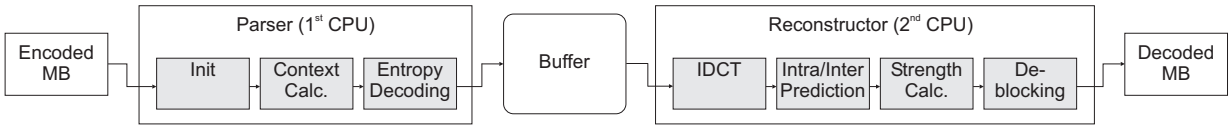

Figure 3. Partitioning the H.264 decoder on a dual-core system. The decoder is split into parser and reconstructor modules. Each module is assigned to a CPU. The CPUs are connected via a buffer in order to exchange macroblock data.

- We estimate the required system resources in an accurate way and identify possible bottlenecks as well as inefficient parts of the system.
- We demonstrate the possibilities of design space exploration with our simulator in two example case studies.
- We provide a systematic guideline on how our simulator can be used for describing additional video algorithms.

The remainder of the paper is organized as follows. In Section 2, we provide a brief overview of the H.264 decoding process and explain how it can be mapped onto parallel hardware architectures. Section 3 focuses on macroblock-based profiling of the H.264 decoder. We investigate the dynamic behavior of the decoding process and the effects of these variations on the system's performance. A high-level simulator is introduced in Section 4. Section 5 describes two test scenarios for demonstrating the capabilities of this simulator. Finally, we present our conclusions in Section 6.

2. MULTI-PROCESSOR DECODING

Due to its high complexity, especially for higher bitrates and resolutions, parallel decoder implementations of the H.264 standard have evolved.[7,8] In our implementation, a functional splitting of the decoding task for a dual-core processor architecture is chosen. The simulation results presented in this paper are based on the functional mapping shown in Figure 3. While the parser processor performs all functions related to bitstream parsing, the reconstructor processor handles all pixel-based operations. Roughly spoken, the parsing processor expands the compressed stream information into an intermediate representation which is subsequently interpreted by the reconstructor.

Parsing and interpreting the compressed bitstream is basically a sequential task, since bits can only be removed from the input stream in sequential order of their appearance in the stream. Although the parsing process is sequential, a large amount of data loads and stores can be performed in parallel. Besides the basic entropy decoding of picture data such as motion vectors and DCT residuals (*Entropy Decoding*), the parser also has to consider the context of neighboring macroblocks (*Context Calc.*). This context calculation includes the prediction step of context adaptive VLC coding for residuals as well as motion vector prediction and prediction of other syntax elements. In an optimized embedded implementation of the H.264 decoder, also the non-algorithmic tasks such as the memory initialization of macroblock data structures as well as DMA descriptor set-ups become a relevant runtime-factor (*Init*).

For pixel-based operations, the reconstructor can make efficient use of data parallelism by parallel instruction execution and SIMD instructions. Especially the intra and inter prediction (*Intra/Inter Prediction*) routines and the inverse residual transformation (*IDCT*), which are based on multiples of the 4×4 pixel block size, allow for efficient usage of 32-bit RISC architectures with SIMD extensions. In H.264, filter strength coefficients for the deblocking process are calculated (*Strength Calc.*) before applying the deblocking filter (*Deblocking*) as the last step in the macroblock decoding process.

3. MACROBLOCK-BASED H.264 DECODER PROFILING

In this section, we describe the test sequences used in this work and provide details on the applied profiling methodology as well as on the hardware and software architectures of our system. Furthermore, we show the dynamics of the video decoding process for the described test sequences.

Table 1. Six test sequences with normalization to 35 dB. For three sequences (Foreman, Flowergarden, Barcelona) additional profiles at 30/32/38 and 40 dB are generated. Columns 2 and 3 show the PSNR values used for normalizing the sequences and the resulting bitrates. The quantization values used for the coding of I-, P- and B-frames are given in columns 4 to 6. A short description of the sequences' characteristics is provided in the last column.

Sequence	PSNR (in dB)	Bitrate@25Hz (in kbit/s)	QP_I	QP_P	QP_B	Remarks
Foreman	30	105	41	39	37	Little texture, strong motion activity
	32	137	38	36	36	
	35	250	33	31	31	
	38	480	28	26	26	
	40	750	25	23	23	
Flowergarden	30	610	36	34	33	Strong differences between top half of slice (sky, poorly textured) and bottom half (meadow, strongly textured)
	32	848	33	32	31	
	35	1350	30	28	28	
	38	2080	27	25	24	
	40	2650	25	22	22	
Barcelona	30	600	34	33	33	Complex texture, complex motion
	32	1200	32	30	28	
	35	2250	28	26	25	
	38	3525	25	23	22	
	40	4500	22	21	20	
Paris	35	600	31	28	28	Little texture, simple motion activity
Bus	35	1000	29	28	28	Highly textured, high motion activity
Mobile	35	1750	29	27	26	Highly textured

3.1 Test sequences

The runtime behavior of the decoder is profiled using six standard image sequences listed in Table 1. In compiling the test set, we focused on reaching a high diversity in the test sequences' contents and on covering the whole complexity range of typical H.264 sequences at CIF resolution. The test sequences are encoded in H.264 main profile with a GOP size of 13 frames using the JM12.2 encoder[9] (CIF, IPB, VLC, deblocking active, all prediction modes allowed, SR +/–16 pixels, 3 reference frames, 1 slice per frame).

To avoid impacts of the encoder rate control algorithm as well as coding quality effects on the decoder runtime, all sequences are coded with constant quantization parameters at an average SNR of 35 dB. We therefore adjust the values of parameters QP_I, QP_P and QP_B that are responsible for quantization of I-, P- and B-frames so that each individual frame of the sequence exhibits an SNR of approximately 35dB. The corresponding settings for the quantization parameters as well as resulting bitrates are found in Table 1. In addition to these streams, we include three sequences normalized to a PSNR of 30, 32, 38 and 40 dB in order to provide the same sequence at different quality levels.

3.2 Profiling and complexity mapping

A single-core simulator for the very long instruction word (VLIW) multimedia processor CHILI[10] provides the profiling results used in this work. The simulator's accuracy has been verified with an ASIC implementation of this processor. The CHILI is a RISC processor. It can process four instructions in parallel which can be any combination of 32-bit arithmetic instructions and load-store operations. For parallel pixel operations, 16-bit SIMD instructions are provided. Each processor has a dedicated 64 Kbyte local memory for fast data access and a 64 Kbyte instruction cache. Data is transferred via DMA or direct access between the processor's local memory and 64 Kbyte shared on-chip SRAM as well as up to 64 Mbyte external memory. The processor can be programmed in C/C++ with intrinsic SIMD functions as well as in low-level assembly language for time-critical routines.

In order to obtain detailed profiling results for the analysis, the simulator of the VLIW media processor and its data memory system (DMS) is extended to enable function profiling on a macroblock level. Hence, the

simulator provides very detailed records of program execution from which the call graph as well as individual execution times for each function call can be extracted. This allows measuring the effects of coding modes on the computational complexity with very fine granularity. The exact execution time for each macroblock is measured by accumulating the function cycle-counts of all sub-functions for each functional block. The functions called during the decoding process of a macroblock are mapped to the functional blocks of Figure 3.

The profilings are based on a commercial H.264/AVC main profile decoder for embedded architectures that has been optimized in terms of memory usage and support of DMA transfers. Additionally, the regular pixel-based processing functions of the decoder (e.g. interpolation, prediction, IDCT) have been assembly optimized to make use of the SIMD processor commands. Macroblock-based runtime measurements for all test sequences of Table 1 are computed for each decoder function.

3.3 Complexity variations in H.264 decoding

Based on the discussed profiling environment, we have measured the macroblock execution times for the six presented test sequences. The results of Figure 1 show the dynamic variations by histogram classification of the execution times. It can be well observed that execution times vary significantly between different video sequences and frame types within the same video sequence.

Several classes of sequences can be distinguished by their histograms in Figure 1. Sequences with homogeneous texture and small motion activity such as Barcelona and Mobile show almost Gaussian distributions for all three macroblock types. In contrast to this, the Flowergarden sequence shows a large spread of execution times caused by the fact that the untextured sky in the top half of the images exhibits significantly different characteristics in comparison to the highly textured flowers in the bottom half of the frames. For this test sequence, the execution times vary by significant factor, even if only macroblocks of the same type are considered. Furthermore, the Flowergarden and Paris sequences show relatively different execution time distributions for P- and B-macroblocks, while all other sequences have more similar distributions throughout different coding modes.

An interesting effect occurs for skipped macroblocks with bi-directional prediction. These macroblocks cause the significant peak at 7×10^4 clock cycles in all six histograms. The sharp characteristics of this peak can be explained by the fact that the largest portion of runtime for these macroblocks is consumed by the two sub-pixel interpolation functions for forward and backward temporal prediction and the rather complex filter strength calculation. Apart from that, the skipped macroblocks have no residual information resulting in negligible runtime variations in the entropy decoder.

Regarding the effects of an increased data rate, it is observed that the average runtime increases from less than 5×10^4 clock cycles for the Foreman sequence coded at 250 kbit/s to approximately 8×10^4 clock cycles for the Barcelona sequence coded at 2.5 Mbit. Hence, an increase of a factor of 10 in data rate roughly doubles the execution time per macroblock.

4. HIGH-LEVEL SIMULATION

Using the macroblock-based profiling results described in Section 3, we are able to describe the complexity of each macroblock in the decoder pipeline with a high accuracy. However, in a multi-core environment various approaches for splitting an H.264 decoder are possible. The efficiency and speed-up of such a parallel decoding system strongly depends on an accurate distribution of the decoding tasks. Additionally, buffering between the hardware blocks is required to compensate the workload variations.

For estimating the runtime behavior and buffer requirements of a program in a multi-core environment, we create a high-level simulator. Figure 4 shows a schematic description of the structure of this simulator. The user has to define a hardware architecture, the algorithm (e.g. H.264 decoding) and how the algorithm is mapped to this platform.

Defining a hardware architecture works via a simple description language. The language serves to describe the hardware blocks of the architecture (e.g. arithmetic units, memory entities) and the hardware characteristics such as core frequency, memory size or memory latency.

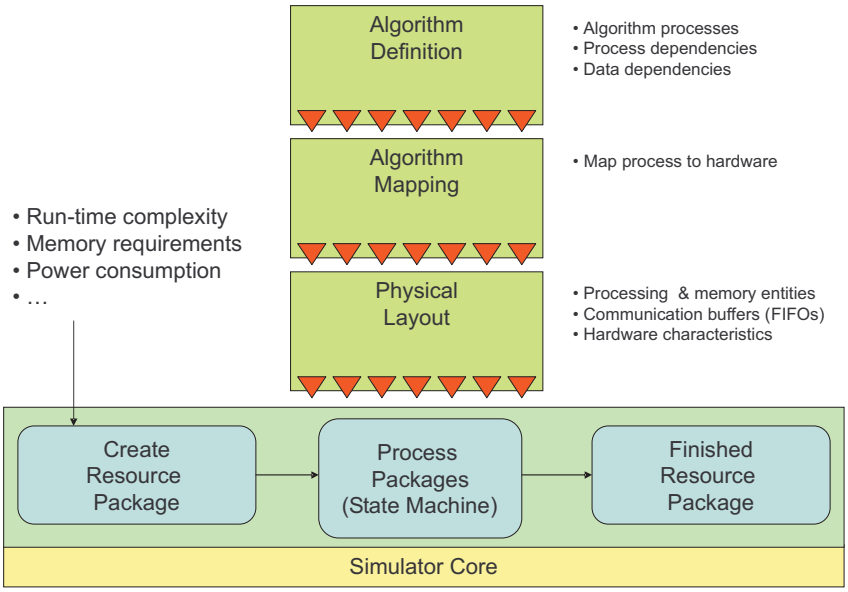

Figure 4. Schematic structure of the high-level simulator. Firstly, we define the algorithm and the physical environment and map the algorithm to this environment. The profiling results provide detailed information about the required resources and computational complexity of the algorithm. The simulator core uses this information - together with the architecture description - for simulating the algorithm behavior on the modeled architecture.

Clearly separated from this hardware description, the simulation environment provides a compact and flexible language to define algorithms at an arbitrary level of detail. This language describes an algorithm by a set of processes and process dependencies. Each process describes a sub-task of the algorithm (e.g. motion compensation in the H.264 decoding algorithm). The dependencies define the temporal processing order between the processes. This corresponds to the program flow.

The relation between hardware model and algorithm description is defined via process mappings. A process is always bound to a single arithmetic processing unit which provides the computational resources for executing the process. First-In First-Out (FIFO) buffers must be defined which serve as communication channels between dependent processes. A FIFO buffer has a predetermined and constant size and is located at a single memory entity. This memory entity characterizes the FIFO's properties such as read/write access behavior, size limitations or latency. Each process receives its input via an input FIFO and writes its output to an output FIFO.

Apart from the hardware layout and the algorithm's structure and flow, additional information on the runtime behavior and the algorithm's resource requirements (e.g. memory requirements, buffers) must be provided. This information represents the characteristics of a program on a specific hardware architecture and is implemented via resource packages. Each resource package represents a data entity (e.g. a macroblock) which enters the system and is processed by a set of algorithmic tasks and according to the algorithm flow. The package describes the runtime and resource requirements for each processing step. The description of the algorithm and the physical structure of the system define the processing order of each resource package. Based on this processing flow and the package's resource requirements, the simulator updates time and size information for the processing units, memories and buffers.

The runtime information for the resource packages is provided by the profiling results described in Section 3. In a previous publication,[6] we have also demonstrated the feasibility of runtime estimation based on macroblock characteristics which can be extracted directly from the compressed data stream. By combining this method with the simulator environment presented in this publication, the dynamic runtime behavior of the H.264 decoding process can be explored without time-extensive hardware simulations. This is a topic that could be addressed in further work.

In case of the H.264 decoder as described in Figure 3, hardware blocks for the parser and the reconstructor,

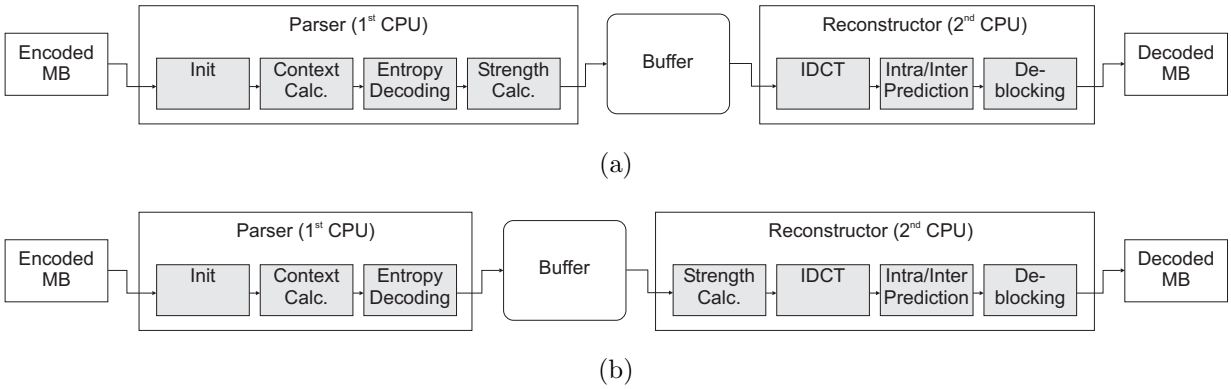

Figure 5. Two methods for partitioning the H.264 decoder on a dual-core system. (a) Scenario 1: The function *Strength Calc.* is part of the parsing module. (b) Scenario 2: The function *Strength Calc.* is part of the reconstructor.

memory units and FIFOs are defined. In the next step, processes for all decoding tasks are defined and mapped to the two processors. The program flow is uniquely determined by defining the input and output FIFO for each process and the process dependencies. For each macroblock, a resource package is created and initialized according to the macroblock's runtime measured in the profiling stage.

Accurate values for the memory and buffer requirements of each task are extracted analytically from the source code of the decoder implementation. Although the simulator supports dynamic memory allocations during program execution, we can neglect this issue for this decoding application. In our H.264 decoding system, dynamic memory allocation is only done during the decoder start-up phase. This improves the runtime efficiency, since no memory is freed or re-allocated and guarantees the availability of sufficient memory during program execution. Except for small structures which are stored on the stack, no additional memory is allocated during runtime. However, buffer transfers for the communication between the cores must be considered. For the buffer communication between the two cores, a macroblock sending structure with constant size is used for each macroblock transfer. This size is used as buffer requirement for each package.

5. RESULTS

This section introduces two case studies for the high-level simulator. The first case study demonstrates the simulator's ability to flexibly describe various software mappings in a multi-core environment. We investigate the system's efficiency for two splitting approaches. In the second case study, we focus on estimating the buffer size requirements in a multi-core decoding system.

5.1 Case study 1: Variation of partitioning

In the following, we examine the effects on the overall system performance when changing the multi-processor splitting approach. As illustrated in Figure 5a, for our experiment, the deblocking filter operation is split into two sub-functions. While the filter strength calculation (*Strength Calc.*) is a mostly context related and sequential task, the actual filtering (*Deblocking*) of pixel data can exploit data parallelism on neighboring pixels. The initial algorithm mapping illustrated in Figure 5a assigns all stream parsing related tasks as well as the filter strength calculation for the deblocking to the parser processor. This splitting has the advantage that all context related functions are implemented exclusively on one processor. The aim of this case study is to analyze this splitting approach with our high-level simulator.

In Figure 6a, it is seen that the described splitting (Figure 5a) results in very unbalanced processor usages for P- and B-macroblocks. The runtime of intra-coded macroblocks is centered around a balanced 50% processor usage with a slight shift towards parser processor usage for higher bitrates. In the case of intra macroblocks, the parser runtime for a macroblock is defined by the significant amount of residual data which is compensated by a low-complexity filter strength calculation. In contrast to this, the P- and B-macroblocks have significantly less residual data, but the filter strength calculation becomes significantly more complex. Therefore, the P-macroblocks result in the highest parser processor usage whereas the B-macroblocks are slightly better balanced

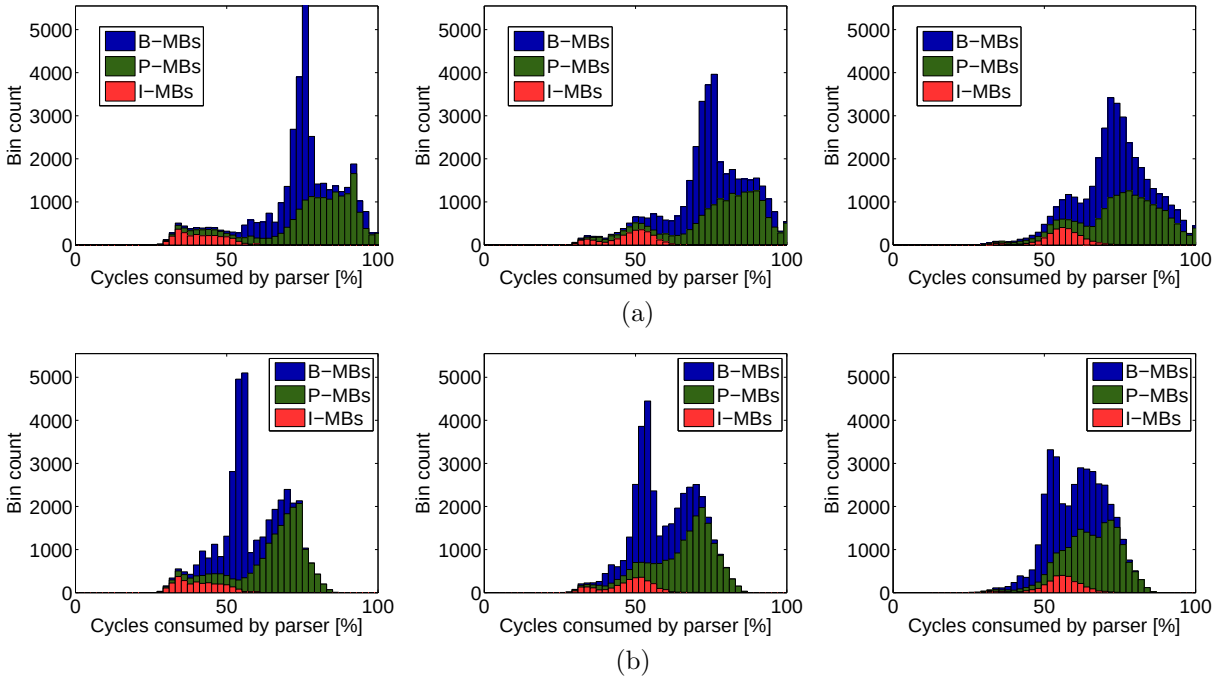

Figure 6. The Foreman sequence at different bitrates. Macroblocks are classified based on the percentage of the overall time that is spent in the parsing module of the decoder. A percentage of 80 means that 80% of the runtime is spent in the parser, while 20% are consumed in the reconstructor. A value of 50% indicates a perfect balance. (a) The filter strength calculation is assigned to the parser module. (b) filter strength calculation is assigned to the reconstructor module.

due to the additional (second) sub-pixel interpolation in the reconstructor. The effects of unbalanced work load can also be observed in the idle time analysis shown in Figure 7a. With the original algorithm splitting, the reconstructor processor's idle time is approximately 40% in all three test sequences and for all data rates.

Based on the above observations a modified algorithm mapping is examined using the high-level simulator. In this splitting, the filter strength calculation is moved from the parser processor to the reconstructor processor as illustrated in Figure 5b. The simulation results for the Foreman sequence in Figure 6b now indicate that the work load balancing between the two processors is significantly improved. For the B-macroblocks, an almost symmetrical CPU usage can be achieved, while the P-macroblocks still show a significant overhead in the parser. One consequence of this insight could be to separate the code for strength calculation for B- and P-macroblocks and optimize the less complex calculations for P-macroblocks separately.

The improvement of CPU usage can also be observed in Figure 7b. In the case of the Foreman sequence, which is coded with 105 kbit/s (30dB) to 750 kbit/s (40dB), the reconstructor idle time can be reduced below 15%. For the test sequences with higher data rates, namely Flowergarden (610 kbit/s to 2.650 kbit/s) and Barcelona (600 kbit/s to 4500 kbit/s), the reconstructor idle time increases with the higher quality. Hence, the higher amount of residual information still causes a significant unbalance in processor usages when using higher data rates.

5.2 Case study 2: Variation of buffers

It is a well-known fact that the sizes of transfer buffers in parallel systems have a high impact on the system's overall performance. The buffers compensate workload differences and improve the cores' usages. However, choosing accurate buffer sizes is challenging and typically represents a trade-off between computational performance and monetary costs that go along with larger memory sizes. In order to find a good trade-off, it requires knowledge about the system's performance at various buffer sizes, which is the issue addressed in this case study.

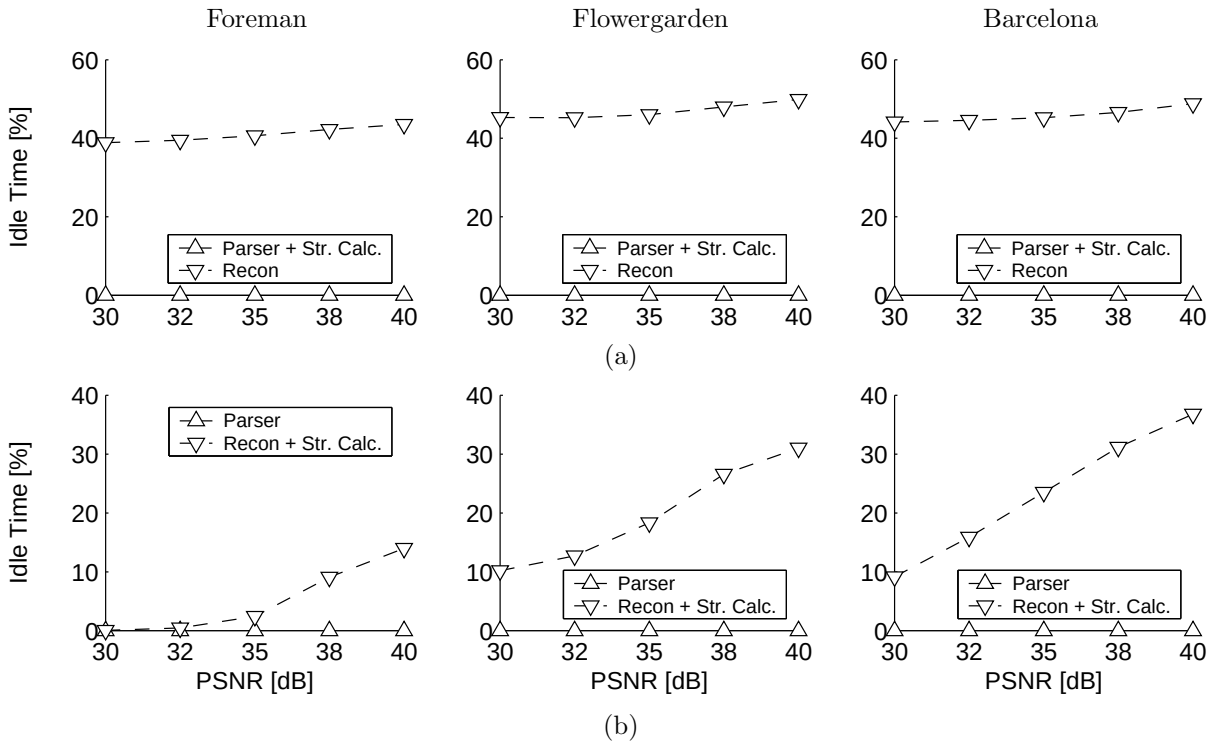

Figure 7. Idle time for parser and reconstructor core for three test sequences. (a) Filter strength calculation is done at the parser side. (b) Filter strength calculation is performed at the reconstructor side.

One major advantage of the functional splitting approach investigated in this work is that the parser as well as the reconstructor can work autonomically and no feedback information from the reconstructor to the parser is required. However, the system's performance in the context of this two-core environment is limited by the transfer rate between parsing and reconstruction core. If the parser cannot deliver macroblocks at an adequate speed, the reconstructor runs out of data and has to wait until new macroblocks arrive. This results in read stalls at the reconstructor side. On the other hand, if the reconstructor processes macroblocks more slowly than the parser, the system's transfer buffer will fill up. If the buffer size is too small to compensate this unequal workload, write stalls at the parser will occur.

Figure 8 visualizes the average idle time of the two decoder cores at different PSNR values. The calculation of the filter strength is mapped to the reconstructor core as shown in Figure 5b. We run simulation for three test sequences. The size of the buffer is thereby varied. We test buffer sizes of one and five macroblocks. Since the core usage is different for each frame type, separate plots for intra- and inter-coded frames are provided.

The first row (Figure 8a) shows the average idle time when decoding intra-coded frames. Increasing the PSNR value (and the bitrate) mainly raises the macroblock processing complexity at the parsing core. At a buffer size of one macroblock the Foreman sequence performs best at 35 dB. For lower PSNR values the parser is faster than the reconstructor and write stalls occur when the transfer buffer is full. For higher PSNR values in the Foreman sequence, the parser is the performance bottleneck and the higher idle time is mainly due to read stalls in the reconstructor. For sequences with typically higher bitrates than the Foreman sequence, we can observe a continuous performance decrease with increasing PSNR values. For these sequences, the parser is the main bottleneck and only few write stalls at the parser side occur.

Inter-coded P- and B-frames (Figure 8b and c) show a different behavior than intra-coded frames. Macroblocks in inter-coded frames are coded more efficiently and typically have a lower number of syntax bits. This results in lower parsing complexity. Furthermore, the complexity in the reconstructor raises due to the computationally more complex motion compensation and deblocking processes. Intuitively, one would assume that this should lead to a higher amount of write stalls at the parser side. For the Foreman sequence, this assumption

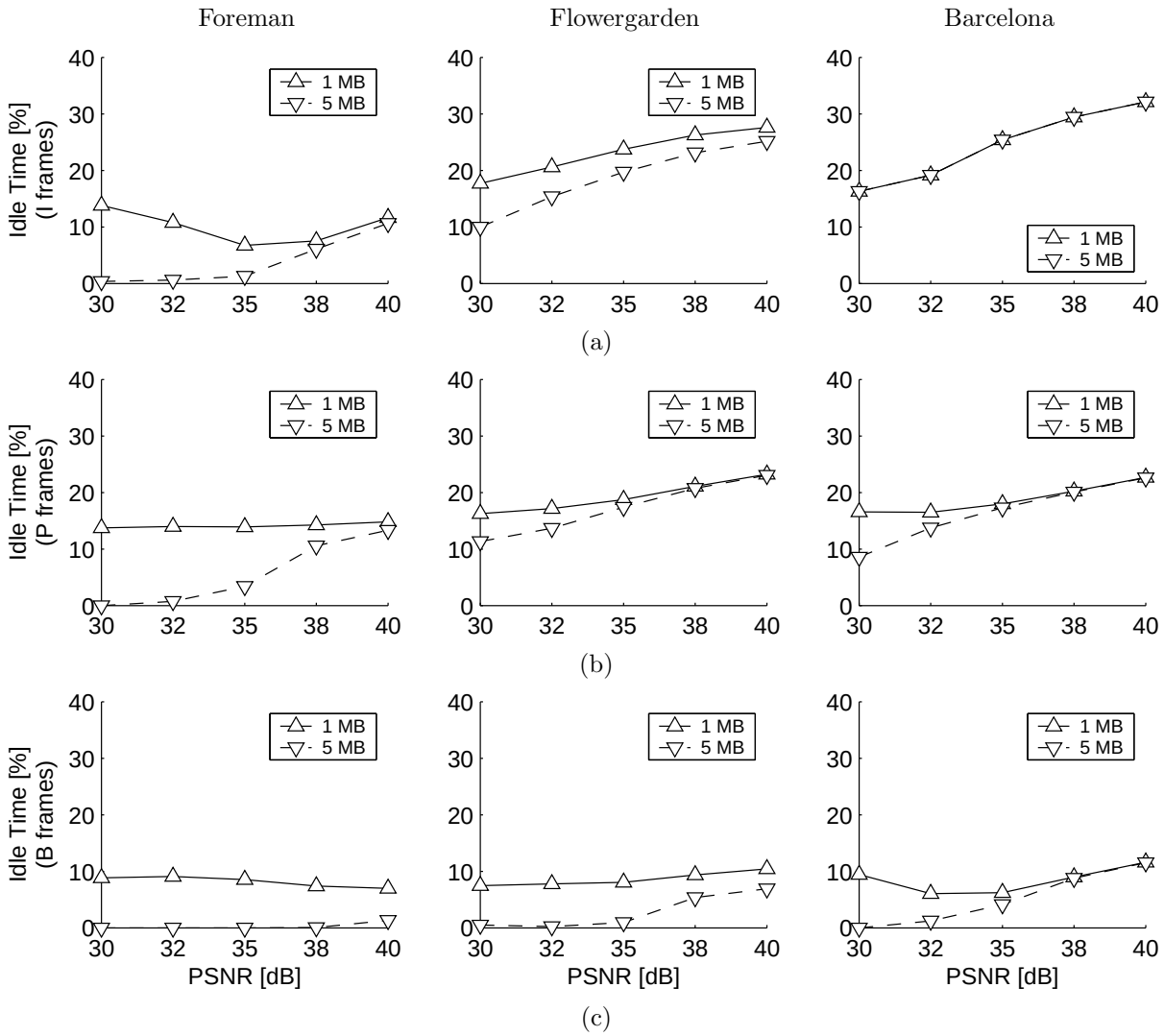

Figure 8. Average idle times of all system cores while decoding (a) intra-coded, (b) inter-coded P- and (c) inter-coded B-frames of three test sequences. For the simulations the calculation of the filter strength was assigned to the reconstructor core. The idle time changes for different buffer sizes and PSNR values.

can be confirmed, since for P- and for B-frames the idle time stays nearly constant for all PSNR values. This indicates that the parsing core is faster than the reconstructor, since with raising bitrate (and parser complexity) nearly no changes in the overall system usage can be seen. For P-frames in the two more complex sequences, a raising idle time for higher PSNR values indicates a significant higher amount of read stalls.

When raising the transfer buffer to five macroblocks buffer size, a continuous performance decrease with increasing PSNR values can be observed for the Foreman sequence. Compared to single-macroblock buffering, a strong idle time reduction for PSNR values lower than 35 dB of nearly 13% is achieved. This is mainly due to a reduction of the write stalls at the parser side. For higher PSNR ranges, less reduction is achieved and at 40 dB the parser starts to exclusively reduce the systems performance.

The intersection point where the average idle time of a system is equal for low and high buffering sizes indicates at which PSNR range the parser becomes the exclusive bottleneck. Buffering between the cores does not improve the system's performance at this quality level. This can be seen for I- and P-frames in the highly textured Flowergarden and the Barcelona sequences (Figures 8a and b). For these sequences, the higher parsing complexity results in typically higher idle times and less performance improvements at higher buffer sizes.

6. CONCLUSION AND FUTURE WORK

In this work, we have measured the runtime complexity of an H.264 decoder on the level of macroblocks. A high-level simulator for mapping the H.264 decoding process onto various hardware architectures has been introduced. The proposed simulation environment can describe parallel algorithm implementations in a flexible way and simulate their behavior on various hardware architectures. Based on the profiling information and the simulator results, we have gained insights into the dynamic behavior of the H.264 decoding process. In two case studies, we have demonstrated the simulators abilities to analyze the efficiency of a multi-core architecture under various conditions. The resulting idle time curves provide a powerful tool for extracting optimal buffer sizes and for estimating PSNR values where the decoder performs most efficient. Our simulation results allow for optimizing a decoding system for specific application aspects such as quality, memory requirements or core usage.

In the first case study, we have tested two competing splitting methods for the H.264 decoder. We have found strong indications that moving the filter strength calculation to the reconstruction core of our system improves the system's efficiency significantly. In the second case study, we have tested various buffer sizes in a dual-core decoder system. Our results reveal that buffering has a high impact on the decoding process for inter-coded slices and for sequences with low coding complexity. In these cases, a significant amount of write stalls is removed. For sequences of high quality (high PSNR values), increasing the buffer sizes does not lead to significant improvement in the system's performance.

Further work will focus on the following aspects. Apart from algorithm mappings that apply a functional splitting, also data-parallel processing can easily be modeled in our simulator. For data-parallel processing, we can define multiple instances of the same process and map these instances to different hardware units. These parallel approaches and further implementation aspects such as power consumption or latency will also be addressed in future work.

ACKNOWLEDGMENTS

The authors would like to thank ON DEMAND Microelectronics AG[10] for providing the processor simulator and infrastructure for this work and in-depth knowledge in the field of video coding. This work has been supported by 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. T. Wiegand, G. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," in *IEEE Trans. on Circuits and Systems for Video Technol.*, **13**, pp. 560– 576, July 2003.
2. M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, "H.264/AVC baseline profile decoder complexity analysis," in *IEEE Trans. on Circuits and Systems for Video Technol.*, **13**, pp. 704–716, (2003), July 2003.
3. V. Lappalainen, A. Hallapuro, and T. Hämäläinen, "Complexity of optimized H.26L video decoder implementation," in *IEEE Trans. on Circuits and Systems for Video Technol.*, **13**, pp. 717–725, July 2003.
4. T.-T. Shih, C.-L. Yang, and Y.-S. Tung, "Workload characterization of the H.264/AVC decoder," in *Proc. of the 5th IEEE Pacific-Rim Conference on Multimedia*, pp. 957–966, 2004.
5. H. Kalva and B. Furht, "Complexity estimation of the H.264 coded video bitstreams," in *Computer Journal*, **48**(5), pp. 504–513, 2005.
6. F. Seitner, R. Schreier, M. Bleyer, and M. Gelautz, "A macroblock-level analysis on the dynamic behaviour of an H.264 decoder," in *Proc. of ISCE 2007*, (Dallas), June 2007.
7. E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture," in *Proc. of the SPIE*, **5022**, pp. 707–718, May 2003.
8. S. H. Wang, W. H. Peng, Y. He, G. Y. Lin, C. Y. Lin, S. C. Chang, C. N. Wang, and P. Chiang, "A platform-based MPEG-4 advanced video coding (AVC) decoder with block level pipelining," in *Proc. of the 2003 Joint Conf. of the 4th Int. Conf. on Information, Communications and Signal Processing and the 4th Pacific Rim Conf. on Multimedia*, **1**, pp. 51– 55, Dec 2003.
9. Joint Model Version 12.2, *http://iphome.hhi.de/suehring/tml/*, 07.01.2008.
10. ON DEMAND Microelectronics AG, *http://www.odmsemi.com*, 07.01.2008.