A QHD-Capable Parallel H.264 Decoder

Chi Ching Chi Ben Juurlink
Embedded Systems Architectures
Technische Universität Berlin
10587 Berlin, Germany
{cchi, juurlink}@cs.tu-berlin.de

ABSTRACT

Video coding follows the trend of demanding higher performance every new generation, and therefore could utilize many-cores. A complete parallelization of H.264, which is the most advanced video coding standard, was found to be difficult due to the complexity of the standard. In this paper a parallel implementation of a complete H.264 decoder is presented. Our parallelization strategy exploits function-level as well as data-level parallelism. Function-level parallelism is used to pipeline the H.264 decoding stages. Data-level parallelism is exploited within the two most time consuming stages, the entropy decoding stage and the macroblock decoding stage. The parallelization strategy has been implemented and optimized on three platforms with very different memory architectures, namely an 8-core SMP, a 64-core cc-NUMA, and an 18-core Cell platform. Evaluations have been performed using $4k \times 2k$ QHD sequences. On the SMP platform a maximum speedup of 4.5× is achieved. The SMP-implementation is reasonably performance portable as it achieves a speedup of 26.6× on the cc-NUMA system. However, to obtain the highest performance (speedup of 33.4× and throughput of 200 QHD frames per second), several cc-NUMA specific optimizations are necessary such as optimizing the page placement and statically assigning threads to cores. Finally, on the Cell platform a near ideal speedup of 16.5× is achieved by completely hiding the communication latency.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; I.4 [**Image Processing and Computer Vision**]: Compression (Coding)

General Terms

Algorithms, Design, Performance

Keywords

H.264, $4k \times 2k$, decoding, Cell, NUMA, SMP, parallel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.
Copyright 2011 ACM 978-1-4503-0102-2/11/05 ...$10.00.

1. INTRODUCTION

A major concern for moving to many-core architectures is the usefulness from an application point-of-view. As a recent study shows [6], contemporary desktop applications rarely require more compute power to justify the parallelization effort. Video decoding, however, is one of the application domains that follow the trend of demanding more performance every new generation [15].

With the introduction of the H.264 video coding standard, compression rate, quality, but also the computational complexity have significantly increased over previous standards [12, 24]. For H.264 video decoding, contemporary multicores can be used to deliver a better experience. Next-generation features like $4k \times 2k$ Quad High Definition (QHD), stereoscopic 3D, and even higher compression rates, on the other hand, will demand full multicore support.

A full parallelization of the H.264 decoder, however, is not obvious. Higher compression is achieved by removing more redundancy, which in turn complicates the data dependencies in the decoding process. Most previous works, therefore, focused mainly on the Macroblock Decoding (MBD) stage, which exhibits fine-grained data-level parallelism. Attempts at parallelizing the Entropy Decoding (ED) stage are rare and have not resulted in a scalable approach. The ED stage is about as time consuming as the MBD stage and has, therefore, been found to be the main bottleneck [5, 8, 10, 13, 19]. Furthermore, previous works have not evaluated their parallelization strategies on several parallel platforms, and therefore have not evaluated the performance portability of their approaches.

In this paper a fully parallel, highly scalable, QHD-capable H.264 decoding strategy is presented. The parallel decoding strategy considers the entire application, including the ED stage. The parallelization strategy has been implemented and optimized on three multicore platforms with significantly different memory architectures. The main contributions of this work can be summarized as follows.

- We propose a fully parallel and highly scalable H.264 decoding strategy, which is compliant with all the H.264 coding features for higher compression rate and quality. Function-level parallelism is exploited at the highest level to pipeline the decoder stages. In addition, data-level parallelism is exploited in the ED stage and the MBD stage.

- We target QHD resolution, while all previous works targeted FHD or lower resolutions. QHD is more meaningful, because contemporary high performance pro-

cessors, e.g., Intel Sandybridge or AMD Phenom II, can achieve the computational requirements of FHD using a single thread, while for QHD this is not the case.

- We implement and evaluate the parallel decoding strategy on three platforms with significantly different memory hierarchies, namely an 8-core SMP, an 64-core cc-NUMA, and an 18-core Cell platform. Optimization for the memory hierarchy are performed and compared for each platform.

This paper is organized as follows. Section 2 provides an overview of related work. Section 3 describes the parallel H.264 decoding strategy. Section 4 details the experimental setup. Sections 5 to 7 present the implementations, optimizations, and experimental results for each platform. Finally, in Section 8 conclusions are drawn.

2. RELATED WORK

Roitzsch [19] proposed a slice-balancing approach to improve the load balance of exploiting slice-level parallelism. Slice-level parallelism, however, is impaired by a reduced compression rate due to adding more slices in a frame. Finchelstein et al. [10] addressed this by line interleaving the slices. The coding inefficiency of regular slicing is in this approach reduced by allowing context selection over slice boundaries. This approach, however, would require a change of the H.264 standard.

Baik et al. [4] combined function-level parallelism (FLP) with data-level parallelism (DLP) to parallelize an H.264 decoder for the Cell Broadband Engine. The entropy decoding, motion compensation, and deblocking filter kernels are pipelined at the granularity of macroblocks (MBs), and the motion compensation of the MB partitions are performed in a data-parallel fashion using three SPEs. Nishihara et al. [18] and Sihn et al. [21] used similar approaches for embedded multicores. Nishihara et al. investigated prediction based preloading for the deblocking filter to reduce memory access contention. Sihn et al. observed memory contention in the parallel motion compensation phase and introduced a software memory throttling technique to reduce this. The parallelism in these approaches is limited, however.

Van der Tol et al. [23] considered FLP as well as DLP and argued that the most scalable approach is the use of DLP in the form of MB-level parallelism within a frame. Alvarez et al. [1] analyzed this using trace driven simulation with several dynamic scheduling approaches. Meenderinck et al. [16] showed that a 3D-wavefront strategy, which combines intra- and inter-frame MB-level parallelism, results in huge amounts of parallelism. Azevedo et al. [3] explored this further using a multicore simulator and showed a speedup of 45× on 64 cores. The employed simulator, however, does not model memory and network contention in detail but assumes that the average shared L2 access time is 40 cycles.

Seitner et al. [20] performed a simulation based comparison of several static MB-level parallelization approaches for resource-restricted environments. Baker et al. [5] used Seitner's "single row" approach in their Cell implementation. This approach is promising due to the abundant parallelism and low synchronization overhead. In our previous work [7] a variant of the "single row" approach with distributed control was implemented on the Cell processor. By exploiting the Cell memory hierarchy a scalability was achieved that approached the theoretical limit. In most of these works (e.g., [1, 3, 5, 7, 16, 20, 23]), the entropy decoding was not considered or mapped on a single core, which causes a scalability bottleneck.

Cho et al. [8] recently presented a parallel H.264 decoder for the Cell architecture in which the entropy decoding is also parallelized. They found that the dependencies in the entropy decoding between MBs in different frames are only to the co-located MBs. They exploited this using a parallelization strategy similar to the Entropy Ring (ER) approach presented in this paper. Their approach can cause load imbalance, however, due to high differences in entropy decoding times of different types of frames, and we introduce the B-Ring (BR) approach to address this. Furthermore, their Cell implementation only uses the PPEs for the entropy decoding and the SPEs for the MB decoding, which causes a bottleneck. In our Cell implementation the entropy decoding can be performed on both the PPEs and any number of SPEs simultaneously, resolving the entropy decoding bottleneck.

3. PARALLEL H.264 DECODER

In this section the highly scalable parallel H.264 decoding strategy is introduced. In this strategy parallelism is exploited in two directions. Function-level parallelism (FLP) is exploited to pipeline the decoder stages and data-level parallelism (DLP) is exploited within the time-consuming ED and MBD pipeline stages. A MB is a 16×16 pixel block of the frame, e.g., a QHD frame has 240 MBs in the horizontal direction, forming a MB line, and 135 of such MB lines in the vertical direction. Previous work mostly exploited either the limited FLP or the DLP in the MBD stage. Without combining FLP and DLP, however, significant speedup over the entire application cannot be achieved. First, the pipelining approach is discussed, followed by the strategies for exploiting the DLP within the ED and MBD stages.

3.1 Pipelining H.264

Figure 1 depicts a simplified overview of the pipeline stages of our H.264 decoder. The stages are decoupled by placing FIFO queues between the stages, buffering the indicated data structures. The Picture Info Buffer (PIB) and Decoded Picture Buffer (DPB) are not needed to pipeline the stages, but for the H.264 decoding algorithm. The PIB is used in the Entropy Decoding (ED) which needs the Picture Info (PI) of previous frames. A PIB entry consists of the motion vectors, MB types, and reference indices of an entire frame. A DPB entry contains an output frame and is used both as the reference and display buffer. The PIB and DPB buffer entries are not released in a FIFO manner, but when they are no longer needed.

The *read* stage reads the H.264 stream from memory or disk and outputs raw H.264 frames. The *parse* stage parses the header of the H.264 frame and allocates a PIB entry. The parsed header and the remainder of the H.264 frame are sent to the ED stage.

The *ED* stage reads the H.264 frame and produces a work unit for each MB of a frame. This stage includes CABAC decoding, filling prediction caches, motion vector calculation, deblocking filter parameter calculation, as well as other calculations. Copies of the motion vector, MB type, and reference indices of each MB are stored in the allocated PIB entry. The produced work units for an entire frame are

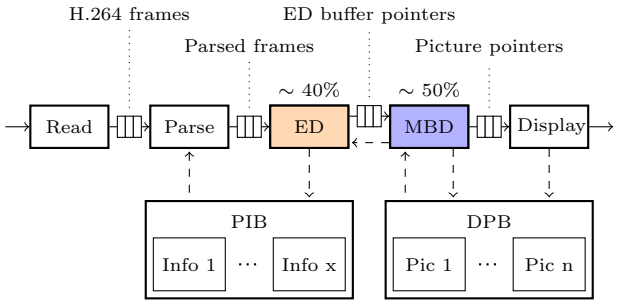

Figure 1: Each pipeline stage in the parallel H.264 decoder processes an entire frame. FIFO queues are placed between the stages to decouple them. Dashed arrows show the buffer release and allocation signals.

placed in an internal ED buffer entry. At the end of the ED stage, if the frames are no longer needed, one or more PIB entries are released and a pointer to the ED buffer entry is sent to the MBD stage. Pointers are passed as ED buffer entries are fairly large (43.5MB for QHD sequences). The internal ED buffer has multiple entries to be able to work ahead. This reduces the impact of dependency stalls when the ED stage temporarily takes more time than the MBD stage and vice versa. In this paper four entries are used as more did not improve performance.

The *MBD* stage processes the work units produced by the ED stage and performs the video decoding kernels that produce the final output frame. This includes intra prediction, motion compensation, deblocking filter, and other kernels. At the start of the MBD stage a DPB entry is allocated for the output frame. At the end of the stage the used ED buffer entry is released by sending a signal to the ED stage. Then one or more reference frames are marked as no longer referenced or released if they have already been displayed. Since the DBP functions both as the reference and as the display buffer, frames must be both displayed and no longer referenced before they can be released. Finally, a pointer to the produced frame is sent to the display stage.

The *display* stage reorders the output frames before displaying them because the decoding order and the display order are not the same in H.264. After a frame has been displayed, it is released if it is no longer referenced, otherwise it is marked as displayed.

Pipelining is effective as long as the pipelining overhead, caused by the buffering operations, does not dominate. The decoupling of the ED and the MBD stage requires an ED buffer of 43.5 MB. This is too large to stay in the cache which causes capacity misses. Further pipelining the ED or the MBD stages would cause even more capacity misses, and has, therefore, not been performed. Instead DLP is exploited in the ED and MBD stages, which is not impaired by the buffering penalty. As indicated in Figure 1, the ED and MBD stages of the H.264 decoder take approximately 40% and 50% of the total execution time, respectively. These percentages have been measured on the SMP platform with the QHD Park Joy sequence.

3.2 Entropy Decoding Stage

In the ED stage, the CABAC decoding is performed, which does not exhibit DLP within a frame. The ED stage, however, does exhibit DLP between frames, but frames are not fully independent. MBs in B-frames can have a *direct* encoding mode. In this mode the motion vectors of the MB are not encoded in the stream, but instead the motion vectors of the co-located MB in the closest reference frame are reused. A potential dependency pattern and the parallelism between frames are illustrated in Figure 2.

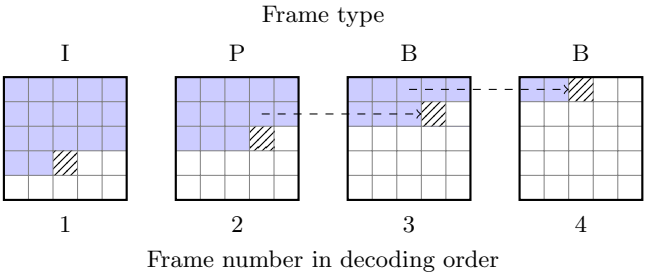

Figure 2: Parallel ED of consecutive frames. Colored MBs have been entropy decoded. Hashed MBs are currently being decoded in parallel

Figure 2 shows that frames can be decoded in parallel as long as the co-located MBs have been decoded before. This is ensured by Entropy Ring (ER) strategy illustrated in Figure 3, which is similar to the strategy used by Cho et al. [8]. In this strategy there are n Entropy Decoding Threads (EDTs) and EDT_i decodes frames i, $n+i$, $2n+i$,... etc. Each EDT performs the same function as the the single threaded ED stage and has four ED buffers entries to be able to work ahead. The Dist thread distributes the frames over the EDTs. The EDTs are organized in a ring structure to ensure that the co-located MB is decoded before the MB that depends on it. To ensure this, at any time EDT_{i+1} is not allowed to have processed more MBs than EDT_i.

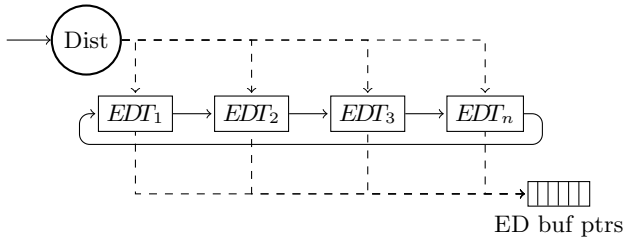

Figure 3: In the ER strategy the EDTs are organized in a ring to maintain dependencies.

The parallelism in the ER strategy scales with the frame size, since there can be as many EDTs as MBs in a frame. In addition, the synchronization overhead is low, since it consists of incrementing a counter containing the number of decoded MBs. Its efficiency is not optimal, however, due to load imbalance. Figure 4 depicts the time it takes to entropy decode each frame in the QHD stream Park Joy [26]. It shows that I- and P-frames take longer to entropy decode, which could cause the EDTs that decode B-frames to stall.

To address this load imbalance, we introduce a slightly more complex B-Ring (BR) strategy, which is illustrated in Figure 5. In this strategy the Split thread splits the I- and P-frames from the B-frames. As depicted in Figure 2, only B-frames have dependencies, since I- and P-frames do not have MBs with a direct encoding. Because B-frames have a relatively constant entropy decoding time, the number of

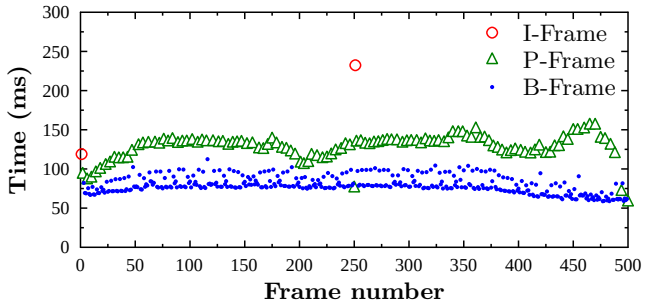

Figure 4: Entropy decoding times of the different frames in the QHD Park Joy sequence.

dependency stalls is reduced, increasing the efficiency. Furthermore, this strategy also exploits that I- and P-frames can be decoded fully in parallel and out-of-order.

The DistB thread distributes the B-frames in a round-robin fashion over the B-frame EDTs. It stalls when a B-frame has a dependency to a not completed I- or P-frame, and then waits for the Reorder thread to signal its completion. The Reorder thread is responsible for reordering the produced ED buffers of the I-, P- and B-frames to their original decode order, before signaling them to the DistB thread and submitting them to the MBD stage. The reordering abstracts the parallel entropy decoding of frames from the MBD stage, thereby reducing the overall complexity and increasing modularity.

The maximum number of parallel B-frame EDTs is equal to the number of MBs in a frame. As this number is very large, we choose to signal the next B-frame EDT after completing an entire MB line, instead of each MB to reduce the synchronization overhead.

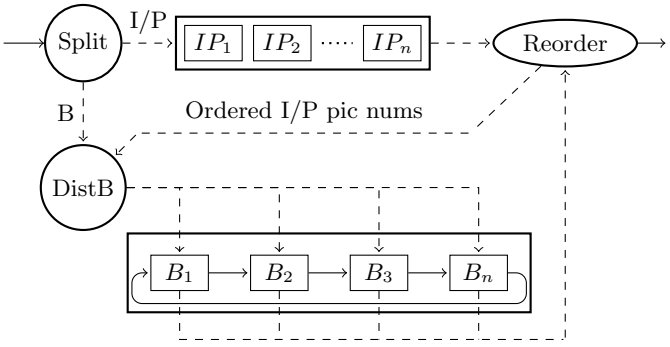

Figure 5: B-Ring strategy. IP denotes an EDT that processes I/P-frames and B denotes an EDT that processes B-frames

3.3 Macroblock Decoding Stage

The MBD stage exhibits DLP within frames as well as between frames, also referred to as spatial and temporal MB-level parallelism, respectively. In our previous work [7] we introduced the Ring-Line (RL) strategy, which exploits only spatial MB-level parallelism. The spatial MB dependencies and parallelism are illustrated in Figure 6. For every MB the data dependencies are satisfied if their upper right and left MB have been decoded. Due to these dependencies, at most one MB per MB line can be decoded in parallel.

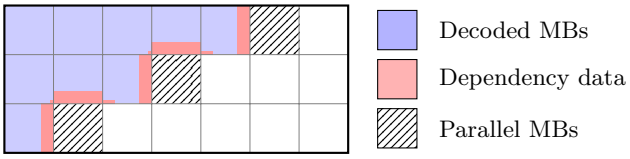

Figure 6: Illustration of spatial MB-level parallelism and dependencies. To decode a MB, data of adjacent MBs is required. The data is available after the upper right and left MB have been decoded.

In this paper, an improved version of the RL strategy is introduced, referred to as the Multi-frame Ring-Line (MRL) strategy. Figure 7 illustrates the MRL strategy. In the MRL strategy macroblock decoding threads (MBTs) are organized in a ring. Each MBT decodes a MB line of the frame. By decoding the lines from left to right the dependency to the left MB is implicitly resolved. The dependency to the upper right MB is satisfied if MBT_{i+1} "stays behind" MBT_i. More specifically, at any time MBT_i must have processed at least two more MBs than MBT_{i+1}. The MBT processing the last line of a frame informs the Release thread of the frame completion. The Release thread releases the ED buffer and one or more reference frames if they are no longer needed. Finally, it signals the decoded picture to the *display* stage. A separate Release thread is used to be able to quickly continue with the next frame.

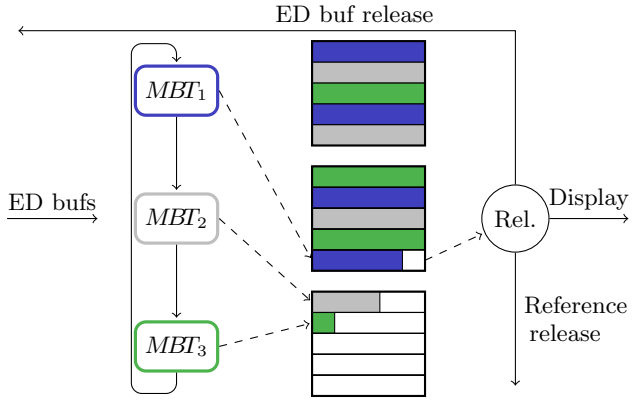

Figure 7: Illustration of the MRL strategy.

The previous RL strategy uses a barrier between consecutive frames. This results in recurring ramp-up and ramp-down inefficiency for each frame, because there are only a few parallel MBs at the beginning and the end of each frame. The new MRL strategy eliminates this inefficiency by overlapping the execution of consecutive frames. Previously this was not possible because the ED stage was not executed parallel to the MBD stage, which is solved in this paper.

Overlapping the MBD stage of consecutive frames, however, may introduce additional temporal dependencies when using too many MBTs, because the required reference picture data for the motion compensation might not be completely available. To ensure that all required reference data is available, the number of in-flight MB lines, thus the number of MBTs, needs to be restricted. The maximum number

of MBTs, MBT_{max}, is given by the following equation:

$$MBT_{max} = \lceil (H - MMV)/16 \rceil, \quad (1)$$

where H is the vertical resolution in pixels and MMV is the maximum motion vector length in pixels. For QHD, assuming that the MMV of QHD will be twice that of FHD, MBT_{max} is $(2160-1024)/16 = 71$. Additionally, the picture border needs to be extended directly after decoding a MB line, because areas outside the actual picture can be used as reference data in H.264.

4. EXPERIMENTAL SETUP

For the evaluation QHD sequences of Xiph.org Test Media [26] are used. These sequences have a framerate of 50 frames per second (fps) and use a YUV 4:2:0 color space. The sequences are 500 frames long, but for the evaluation they have been extended to 10000 frames by replicating them 20 times. The sequences have been encoded with x264 [25], using settings based on the High 5.1 profile. The encoding properties are listed in Table 1. The average bitrates of the encoded QHD sequences varied between 77.6 and 259.8 Mbps. In comparison, 16 Mbps FHD sequences with 25 fps are considered high quality. The parallel H.264 decoder has been evaluated using two QHD sequences, Park Joy and Ducks Take Off, which have a bitrate of 117.8 Mbps and 259.8 Mbps, respectively. For conciseness only the results for the Park Joy sequence, which represents the average case, are provided. In general higher bitrate sequences translate to lower framerates, but higher speedups compared to lower bitrate sequences.

Table 1: X264 encode setting for the Ducks and Park QHD sequences.

Option	Value	Brief description
–cfr	23	Quality-based variable bitrate
–partition	all	All MB partition allowed
–b-frames	16	Number of consecutive B-frames
–b-adapt	2	Adaptive number of B-frames
–b-pyramid	normal	Allow B-frames as reference
–direct	auto	Spatial and Temporal Direct MB encoding
–ref	16	Up to 16 reference frames
–slices	1	Single slice per frame

The parallel H.264 decoder is evaluated on three platforms with significantly different memory architectures. An overview of the platforms is provided in Table 2. To determine the performance the wall clock time of the entire H.264 decoder is measured. This includes all stages depicted in Figure 1, except the display stage. The display stage is disabled since the evaluation platforms do not provide this feature.

The baseline implementation is the widely-used and open-source FFmpeg transcoder [9]. FFmpeg offers a high performance H.264 decoder implementation with, among others, SSE and AltiVec optimizations for the MBD kernels and an optimized entropy decoder. It is one of the fastest single threaded implementations [2]. The FFmpeg framework, however, does not allow a clean implementation of our parallelization strategy. The provided codec interface enforces that only a single frame is in-flight at a time. To solve this, FFmpeg has been dismantled of everything not related to H.264 decoding and rebuilt in a lightweight parallel version using the POSIX thread library facilities for parallelization and synchronization. Based on the decoupled code also a new sequential version is developed which serves as the baseline performance.

Table 2: Platform specifications.

	SMP	cc-NUMA	Cell
Processor	Xeon X5365	Xeon 7560	PowerXCell 8i
Sockets	2	8	2
Frequency	3 GHz	2.26 GHz	3.2 GHz
Cores	8	64	18
SMT	-	off	2-way PPE
Local store	-	-	4 MB
Last level $	16 MB	192 MB	1MB
Interconnect	FSB	QPI	FlexIO
Memory BW	8.5 GB/s	204.8 GB/s	25.6 GB/s
Linux kernel	2.6.28	2.6.36	2.6.18
GCC	4.3.3	4.4.3	4.1.1
Opt. level	-O2	-O2	-O2

5. BUS-BASED SMP

The first platform that we consider is a Symmetric MultiProcessor (SMP) platform. This platform has 8 homogeneous cores with symmetric memory access via a single memory controller through a shared Front Side Bus (FSB). While it is possible to extend this architecture with more cores and memory controllers, the shared FSB constitutes a scalability bottleneck. The programming effort for such a system, however, is relatively low as there are no or few specific optimizations required for the memory architecture. Some general optimizations have been performed to minimize false sharing, such as duplicating the motion compensation scratch pad and the upper border buffers, which improves performance on all cache coherent architectures.

The performance and speedup results are depicted in Figure 8 for the Park sequence. Each bar in the figure is labeled by n-m, where n denotes the number of EDTs and m the number of MBTs. For conciseness, n denotes the combined number of EDTs and, therefore, has a minimum of two, corresponding to one IP-frame and one B-frame decoding thread. The ratio of the number of IP-frame EDTs to B-frame EDTs does not differ very much and is about 1 to 2 for all platforms. The read, parse, display, split, distribute, reorder and release threads are not taken into account in this total thread number. There is one of each such threads.

The SMP platform exhibits reasonable performance and scalability. A maximum speedup of 4.5× is achieved, with a performance of 25.9fps. The sequential decoder, which is based on the decoupled code used for the SMP parallel version, is slightly slower than the original FFmpeg code in which the ED and MBD stages are merged. The difference is around 15% and is observed on all platforms. The performance degradation is caused by additional cache misses introduced by using the large ED buffers needed to decouple the ED and MBD stages, as mentioned in Section 3.1.

The figure shows that using more than 4 MBTs reduces performance considerably. The reason for this is as follows. Since there are more threads than cores, some MBTs will be temporarily descheduled. Because MBTs depend on each other, however, this will stall other MBTs. Here it needs

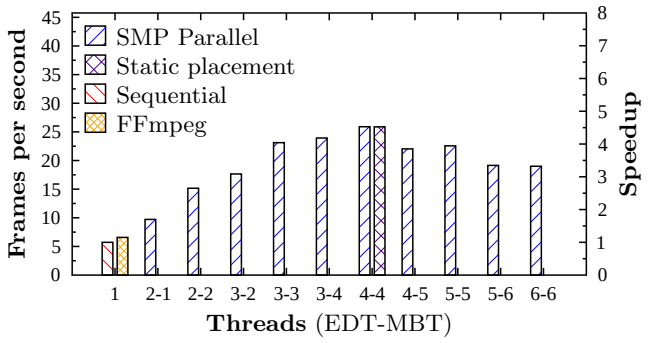

Figure 8: Performance and scalability on the 8-core SMP for the Park sequence.

to be remarked that thread synchronization has been implemented using busy waiting, because it incurs lower overhead than blocking. The EDTs are less likely to stall than MBTs since, as shown in Figure 6, the MBT that decodes a certain MB line has to stay at least two MBs behind the MBT that processes the previous MB line. Therefore, MBTs can tolerate running out of pace for only a few MBs compared to a few MB lines for EDTs.

The reduced scaling efficiency is mostly caused by the limited memory bandwidth of this platform. To show that the FSB is not the bottleneck, Figure 8 also depicts the results for the *Static placement* version. In the Static placement version, consecutive MBTs are placed on the same node to reduce cache coherence misses and, therefore, FSB traffic. No performance improvement is observed, however. Other possible causes for the saturated scalability are insufficient application parallelism and threading overhead. If either of these is the cause, it would also limit the scalability on the other platforms, which is shown to be not the case in the following sections.

6. CACHE COHERENT NUMA

Our second evaluation platform is an 8-socket cc-NUMA machine [11] based on the Nehalem-EX architecture. Each socket contains 8 homogeneous cores, for a total of 64 cores. Each socket is also a memory node as it accommodates an individual memory controller. Inter-node cache coherence and memory traffic use the QPI network with an aggregate bandwidth of 307.2 GB/s. Together with an aggregate memory bandwidth of 204.8 GB/s, this platform offers very high communication bandwidth, per-core 3 to 5 × higher than the SMP platform. To exploit this communication bandwidth, however, NUMA specific optimizations are required. First, the NUMA optimizations performed to the SMP implementation are described, followed by the experimental results.

6.1 cc-NUMA Optimizations

To optimally utilize the NUMA memory hierarchy, the parallel H.264 decoder requires specific optimizations. Page placement on the cc-NUMA platform uses the "first touch" policy. This policy maps a page to the node that accesses it the first time. A poor initial thread placement can cluster large parts of the working set in a single memory node. A way to ensure a balanced memory distribution is to statically assign threads to cores. For this only the EDTs and MBTs are considered, since they access most of the working set.

Figure 9 illustrates the static thread placement strategy for a 4-socket configuration. In the figure, IP_i and B_i denote the IP- and B-frame EDTs, respectively. M_i denotes the MBTs.

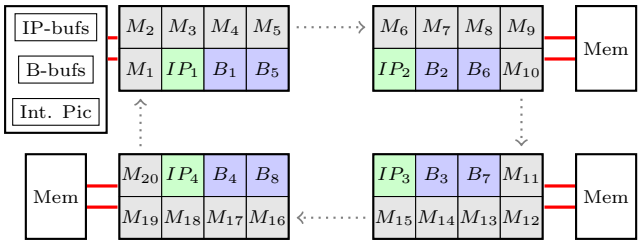

Figure 9: Static thread placement on the cc-NUMA platform.

The EDTs are placed in a round-robin fashion over the sockets. This ensures that the ED buffers are distributed evenly over the memory nodes, as the EDTs are the first to access them. This static thread placement also improves data locality, since the EDTs always find the ED buffer data in their local memory node. The MBTs are placed in a block distributed fashion over the available sockets. This ensures that the picture data is distributed evenly in a block interleaved manner over the memory nodes. Furthermore, placing consecutive MBTs on a single node increases locality as they share an overlapping part of the picture data. In this way most coherence traffic stays on the same node. Some MBTs still need to access a remote memory node, but contention is minimized and the node distance is always only one hop. In addition, the static placement reduces thread migration as threads are bound to cores. Thread migrations are expensive on cc-NUMA platforms [14], which can cause a lot of dependency stalls.

The static thread placement yields a page distribution that is optimal for the ED stage and MBD stage *separately*, but which is not *globally* optimal. A single EDT produces an entire ED buffer entry for a frame, but several parallel running MBTs consume this ED buffer to process the frame further. Since complete ED buffers are allocated in a single node, all MBTs will have to access this node at the same time, resulting in a temporal memory access hotspot.

This hotspot is avoided by letting the MBTs first touch the ED buffers, instead of the EDTs. This ensures that both the input ED buffer entry pages and DPB entry output pages of each MBT are distributed evenly, as illustrated in Figure 10. The downside, however, is that now each EDT will have to write to remote memory nodes. But because the overall contention is reduced and because read latency is more important than write latency, this page placement improves the overall performance.

In addition to letting the MBTs first touch the ED buffer entries, the MBTs are assigned to process the MB lines corresponding to the pages they touched. Without this, depending on the number of MB lines per frame and the number of MBTs, the MBTs process different MB lines in each frame. For example, when there are 8 MB lines in a frame and 3 MBTs, MBT_1 decodes MB lines 1, 4, and 7 of the first frame, MB lines 2, 5, and 8 of the second frame, etc. MB line 2 of the second frame, however, resides in a different memory node as it is first touched by MBT_2, which results in a lot of inter-node memory accesses.

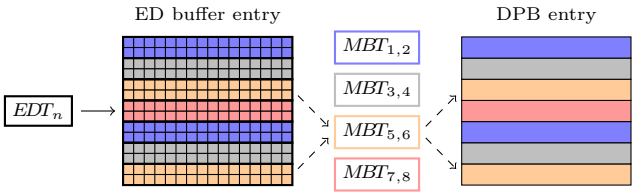

Figure 10: Illustration of the globally optimized page placement and the MBT to MB lines binding. The colors denote the thread and page placement to different nodes.

6.2 cc-NUMA Experimental Results

Four versions of the parallel H.264 decoder have been evaluated on the cc-NUMA platform. The first version, referred to as "SMP parallel", is the same as the one used on the SMP platform. The second version, referred to as "Interleaved" employs a round-robin page placement policy instead of first touch. The third version, referred to as "Static placement" uses the static thread placement presented in Section 6.1. The fourth version, referred to as "NUMA optimized", applies in addition to the static thread placement, the globally optimized page placement of the ED buffers and the MBT to line binding.

Figure 11 shows the performance and scalability of each version for 1, 2, 4, and 8 sockets. The figure shows the results obtained using the best performing thread configurations, which have been found through a design space exploration. An exception to this is the Interleaved version, which uses the same thread configuration as the SMP parallel version. The optimal thread configurations are depicted in Table 3. Figure 11 shows that the parallel H.264 decoder is able to scale to very high performance levels. The maximum achieved frame rate is 200 fps with a speedup of 33.4×.

While the performance is very high, the scaling efficiency decreases with more sockets. For example, the SMP parallel and Interleaved versions exhibit reasonable scaling up to 2 sockets, with a speedup of 11.6× on 16 cores. However, they become less efficient when deploying 4 and 8 sockets for which a speedup of 26.6× is observed on 64 cores. When using a static thread placement the performance and scalability increase considerably for 4 and 8 sockets. For example, for 8 sockets the performance of the Static placement version is 200 fps versus 157 fps for the SMP parallel version.

The NUMA optimized version performs slightly better for 4 sockets and slightly worse for 8 sockets compared to the Static placement version. The reason why the NUMA optimized version is slightly slower on 8 sockets is that only 56 threads (EDTs + MBTs) are used versus 64 threads for the Static placement version. Because the number of MBTs is less flexible due to the static binding of MB lines to MBTs the optimal performance is obtained with a smaller thread configuration. To increase the performance of the MBD stage the number of MBTs have to be increased from 27 to 34. This, however, leaves no cores to increase the number of EDTs. We expect that the performance difference between the NUMA optimized version and the Static placement version would increase with more sockets and/or cores.

The impact of the NUMA optimized version is, however, visible in the thread configurations. With more sockets the ratio between the number of EDTs and the number of MBTs changes in favor of the number of MBTs in the Static placement version. The efficiency of scaling the number of MBTs, therefore, decreases considerably with more sockets due to increased contention when reading from an ED buffer entry. For the NUMA optimized version this ratio remains fairly constant because in the globally optimized page placement the ED buffer entries are read from all memory nodes simultaneously, thereby avoiding contention.

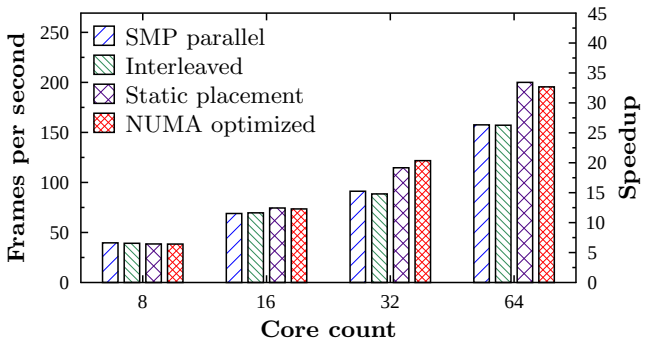

Figure 11: Performance and scalability on a 8-socket cc-NUMA machine for the Park sequence.

Table 3: Optimal thread configuration for the Park sequence. E denotes the combined number of EDTs, M denotes the number of MBTs.

	1		2		4		8	
	E	M	E	M	E	M	E	M
SMP parallel	5	4	8	8	12	19	23	40
Interleaved	5	4	8	8	12	19	23	40
Static placement	5	3	8	8	14	18	24	40
NUMA optimized	5	3	9	7	16	15	29	27

Optimizing the thread mapping and page placement yields performance improvements of up to 27.3%. A static thread placement, however, is undesirable because other programs might map their threads to the same cores, while there are other cores available. Our results indicate, however, that techniques that give priority to locality over load balancing, such as resource partitioning, locality-aware scheduling [22], and runtime page migration [17], can provide significant performance benefits, when increasing the number of cores.

7. CELL BROADBAND ENGINE

Our final platform has a local store memory architecture, and consists of two Cell Broadband Engines processors with 2 PPE cores and 16 SPE cores. The Cell architecture is very different from the previous two platform, as it exposes the on-chip memory hierarchy to the programmer. On the one hand, the programmer is given control of regulating the data flow between the cores and the off-chip memory. On the other hand, the programmer is now responsible for fitting the data structures in the on-chip memory, which is performed transparently by the hardware in cache-based processors.

On the Cell architecture the same parallel H.264 decoding strategy is used. The differences are in the implementations of the ED and MBD stages. As most of the time is spent in these stages, it is necessary to port both of them to the SPEs to gain overall speedup. The other stages of the decoder and the control threads run on the PPEs using the Pthread base code. The implementations and optimizations of the ED

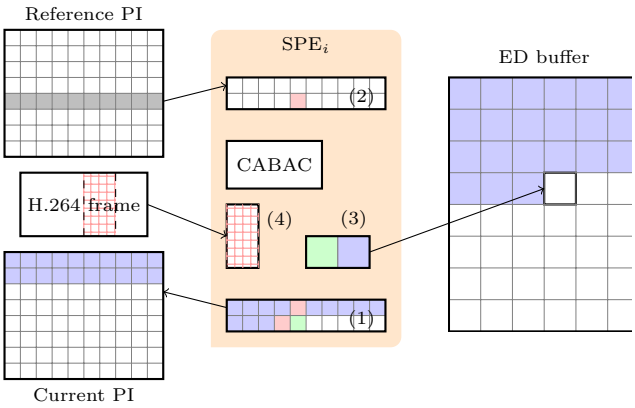

Figure 12: Overview of the SPE EDT implementation. Data structures in the orange background are located in the local store. The other data structures reside in the main memory.

and MBD stages on the Cell SPEs are discussed in the next two sections, followed by the experimental results.

7.1 Entropy Decoding on the SPE

From the threads in the ED stage depicted in Figure 5 only the I/P- and B-frame entropy decoding threads are executed on the SPEs. Although the I/P- and B-frame decoding threads process different types of frames, their SPE implementations are quite similar. Therefore, the base SPE EDT implementation is presented first and the differences between the two are described later. Figure 12 depicts a simplified overview of the EDT implementation. The color of the structures denote the "state" of the data. Blue denotes that it has been produced in this frame, gray denotes that is has been produced in a previous frame, red denotes that it is used for the ED of the current MB, and green denotes that it is produced by the ED of the current MB.

Each EDT requires access to several data structures that do not fit in the local store. The required input data structures are the CABAC tables and buffers, H.264 frame data, and the reference Picture Info (PI). The output data structures are the PI and the ED buffer of the current frame. The CABAC tables and buffers are able to fit in the local store. The other data structures are too large and, furthermore, their size increases with the resolution.

Close examination of the ED algorithm reveals that there is little reuse of data. Performing the ED of a MB only uses the PI data produced by the ED of the upper and left neighboring MBs. From the reference PI only the data corresponding to the co-located MB is used. This allows keeping only a small window (1) of the PI data in the local store. With a window of two MB lines in the local store, the PI data produced by decoding the current MB line can be written back during the decode of the next MB line. Furthermore, the data of the upper MB stays in the local store until it is used for decoding the lower MB. For the reference PI also a buffer of two MB lines (2) is allocated in the local store to be able to prefetch the next MB line. The motion vectors of the reference PI, however, cannot be prefetched for a complete MB line due to local store size constraints. Instead, the motion vectors of 4 MBs are prefetched at a time.

The ED buffer elements are not reused by the EDT. Therefore, only two buffer elements (3) are required in the local store to perform a double buffered write back. The only data that is not double buffered is the H.264 frame window (4), because the total amount of traffic for reading the H.264 frame is small. We have, therefore, decided to decrease the local store usage and code complexity, by keeping a single H.264 frame window with a size of 4KB in the local store.

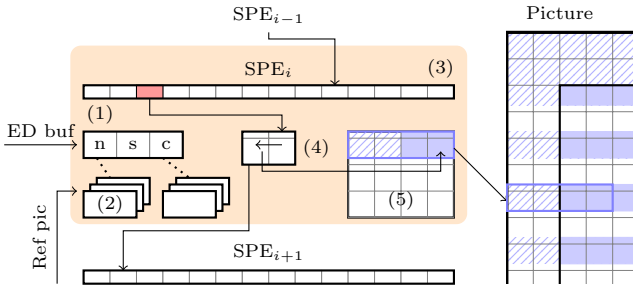

Figure 13: Overview of the SPE MBT implementation. Data structures in the orange background are located in the local store of SPE$_i$.

The total local store footprint of the described EDT implementation for QHD resolutions is 238 kB, of which 63 kB is program code. The ED implementation for I/P-frames does not require a reference PI window. In the B-frame EDT implementation, after decoding each line a signal is sent to the next EDT in the B-ring to maintain the dependencies between co-located blocks.

7.2 Macroblock Decoding on the Cell

Similar to the ED stage, only the MBTs of the MBD stage are mapped to the SPEs. The problems of porting the code and performing the data partitioning have already been solved for a large part in our previous work [7]. Some improvements are necessary to support the QHD resolution and to overlap execution of consecutive frames. A simplified overview of the data allocation in the SPE MBT implementation is shown in Figure 13.

Each MBT uses an ED buffer entry and one or more reference pictures as input to produce the output picture data. As is the case for the EDTs, these data structures are too large to fit completely in the local store and several smaller data windows are allocated in the local store to hold only the active part of the data structures.

The MBD algorithm only requires one ED buffer element at a time to decode a MB. Three ED buffer elements (1) and two motion data buffers (2) are allocated in the local store to be able to prefetch both the ED buffer elements and the motion reference data. In Figure 13, element c denotes the element for the current MB, n denotes the element for the next MB, and s denotes the element for the second next. Element s and the motion data of element n can be prefetched, while element c is used to decode the current MB. After decoding each MB the roles of the elements rotate. Element n, of which the motion data has been prefetched, becomes the current element c, element s becomes the next element n, and element c can be reused to hold the new second next element.

To decode a MB, the picture data produced by decoding the upper-left to upper-right MBs is needed. Each SPE, therefore, has a buffer (3) to receive the filtered and unfiltered lower lines of these upper MBs from the previous MBT

in the ring. In this way the data is kept on chip, reducing the number of off-chip memory transfers. The buffer has 240 entries, one for each MB in a MB line of a QHD frame.

For the picture data, a working buffer with a size of 32×20 pixels (4) is needed to fit the picture data of two MBs and their upper borders. Before decoding the MB, the upper borders are copied into the working buffer. After decoding the MB, the data of the previously decoded MB, residing in the left side of the working buffer, is copied to the DMA buffer (5), then the picture data produced by decoding the current MB is copied to the left part of the working buffer to act as the left border of the next MB. The produced picture data cannot be copied directly to the DMA buffer as the deblocking filter not only modifies the picture data of the current MB, but also the picture data of the left MB and the received upper border data. Therefore, the write back of the picture data has to be delayed by one MB and also includes the lower lines of the upper MB.

In our previous implementation [7], the upper border buffer and the picture data buffer were joined to avoid the additional copy steps performed in the working buffer. This approach, however, required an entire MB line to be allocated in the local store, which is not feasible for QHD resolution. Another difference with our previous implementation is the DMA buffer. This buffer is enlarged to be able to perform the picture border extension directly after decoding a MB line to support the overlapped execution of two consecutive frames, as mentioned in Section 3.3.

In total, the local store footprint of the SPE MBT implementation is 197 kB, of which 121 kB is program code. As everything fits in the local store, techniques such as code overlaying, which have been used in other implementations [5, 8], are not required in our implementation.

7.3 Cell Experimental Results

To show the efficacy of the optimizations described in the previous sections, two versions of the Cell implementation are evaluated. The *Non-blocking* version employs the DMA latency hiding, double buffering techniques described in the previous section. The *Blocking* version does not use these techniques, but blocks when fetching data. Furthermore, in order to evaluate the impact of the available memory bandwidth, both versions are evaluated with only one and both memory controllers (MCs) enabled.

Figure 14 presents the performance and scalability results for the Cell platform. The figure shows that the *Non-blocking* version achieves a near ideal speedup of 16.5×. The speedup is relative to the single-threaded version (without multi-threading code) running on one PPE. The results are shown for 4 to 16 SPEs in steps of 4 SPEs. The results for 18 threads are obtained by executing two additional I/P-frame EDTs on the PPEs. The near ideal speedup implies that the SPE EDT and MBT implementations are as fast as their PPE counterparts.

In the *Non-blocking* version almost all data transfers are completely overlapped with the computation, which results in an up to 34% higher performance than the *Blocking* version. Data transfer latencies only reduce the performance in the *Non-blocking* version when they actually take longer than the computation. In our implementation this does not not occur until the application starts to become bandwidth limited. The results show that the memory bandwidth of one MC is saturated at around 20 fps. The performance

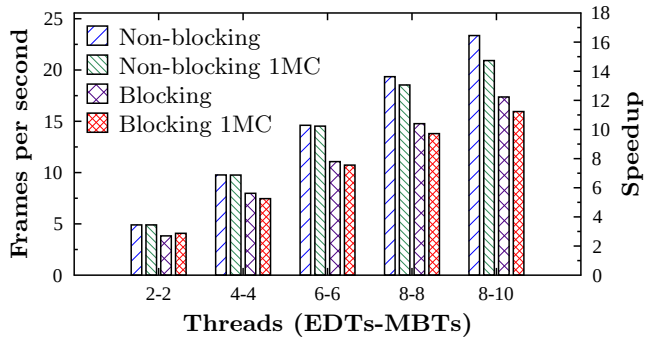

Figure 14: Performance and scalability on the Cell platform for the Park sequence.

of the *Blocking* version, however, is already reduced by disabling one MC at a lower frame rate, which indicates the effect of memory access contention.

For the Cell implementation additionally several FHD sequences are evaluated to be able to compare to the implementation of Cho et al. [8]. To be comparable, these FHD sequences are encoded using a 2-pass encoding to get an average bit rate of 16 Mbps instead of the constant quality mode used for the QHD sequences. Table 4 depicts the performance results of our Cell implementation and the results obtained by Cho et al. Compared to the work of Cho et al., the performance is between 2.5× and 3.3× higher. This difference is mostly caused by being able to use the SPEs for parallel entropy decoding, while the implementation of Cho et al. uses only the two PPEs for that stage.

Table 4: Performance comparison of the Cell implementation using 16 Mbps FHD sequences.

Sequence	EDT-MBT	Our decoder	Cho et al. [8]
Pedestrian	9-9	91 fps	37 fps
Tractor	10-8	81 fps	31 fps
Station 2	9-9	79 fps	24 fps
Rush Hour	8-10	88 fps	34 fps

8. CONCLUSIONS

In this paper a high-performance, fully parallel, QHD-capable H.264 decoder has been presented. The employed parallelization strategy exploits the available parallelism at two levels. First, function-level parallelism is exploited by pipelining the decoder stages. This allows several frames to be processed concurrently in different stages of the decoder. In addition, data-level parallelism is exploited within the entropy decoding (ED) and macroblock decoding (MBD) stages, as these two stages account for more than 90% of the total execution time. In the ED stage data-level parallelism between frames is exploited using a novel B-ring strategy. By separating the I- and P-frames from the B-frames, the I- and P-frames can be processed completely in parallel, while load balancing is improved for the B-frames. In the MBD stage mostly MB-level parallelism within a frame is exploited. Limited parallelism at the beginning and end of each frame is avoided by overlapping the execution of consecutive frames.

The parallel decoder has been implemented on three multicore platform with substantially different memory architectures. On the 8-core SMP platform the limited memory

bandwidth restricts the scalability to about 4.5×. Furthermore, the SMP parallel version is reasonably performance portable to the 64-core cc-NUMA platform as it achieves a speedup of 26.6×. On the cc-NUMA platform, due the non-uniform memory hierarchy and the large number of cores, specific optimizations are necessary to obtain the highest achievable performance and scalability. To efficiently exploit the distributed memory, a locality-aware static thread placement and page placement scheme have been presented. These optimizations yield additional improvements of up to 27.3% over the SMP parallel version, with a maximum performance of 200 fps. Scalability on the Cell platform is close to ideal with 16.5× on 18 cores. Due to vigorous overlapping of communication with computation, the Cell implementation is tolerant to DMA transfer latencies, which allows more efficient use of the memory bandwidth. Lack of portability and the required programming effort are known disadvantages of the Cell architecture, however.

The evaluation on the three platforms shows that our parallel H.264 decoding strategy scales well on a wide range of multicore architectures. Furthermore, the performance obtained on the cc-NUMA shows that multicores provide computational headroom that can be used to further innovation in the video coding domain. Finally, the performance results also show that exploiting the memory hierarchy becomes increasingly critical when the number of cores increases.

9. ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647. We would like to thank the Future SOC Lab of the Hasso Plattner Institut and the Mathematics department of TU Berlin for giving us access to their platforms. Finally, we would like to thank the anonymous reviewers for their constructive remarks.

10. REFERENCES

[1] M. Alvarez, A. Ramirez, A. Azevedo, C. Meenderinck, B. Juurlink, and M. Valero. Scalability of Macroblock-level Parallelism for H.264 Decoding. In *Proc. 15th Int. Conf. on Parallel and Distributed Systems*, 2009.

[2] M. Alvarez, E. Salami, A. Ramirez, and M. Valero. A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC. In *Proceedings IEEE Int. Symp. on Workload Characterization*, 2005.

[3] A. Azevedo, C. Meenderinck, B. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, and A. Ramirez. Parallel H.264 Decoding on an Embedded Multicore Processor. In *Proc. 4th Int. Conf. on High Performance Embedded Architectures and Compilers*, 2009.

[4] H. Baik, K.-H. Sihn, Y. il Kim, S. Bae, N. Han, and H. J. Song. Analysis and Parallelization of H.264 Decoder on Cell Broadband Engine Architecture. In *Proc. Int. Symp. on Signal Processing and Information Technology*, 2007.

[5] M. A. Baker, P. Dalale, K. S. Chatha, and S. B. Vrudhula. A Scalable Parallel H.264 Decoder on the Cell Broadband Engine Architecture. In *In Proc. 7th ACM/IEEE Int. Conf. on Hardware/Software Codesign and System Synthesis*, 2009.

[6] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proc. 37th Int. Symp. on Computer Architecture*, 2010.

[7] C. C. Chi, B. Juurlink, and C. Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proc. 24th Int. Conf. on Supercomputing*, 2010.

[8] Y. Cho, S. Kim, J. Lee, and H. Shin. Parallelizing the H.264 Decoder on the Cell BE Architecture. In *Proc. 10th Int. Conf on Embedded software*, 2010.

[9] The FFmpeg Libavcodec. `http://ffmpeg.org`.

[10] D. Finchelstein, V. Sze, and A. Chandrakasan. Multicore Processing and Efficient On-Chip Caching for H.264 and Future Video Decoders. *IEEE Trans. on Circuits and Systems for Video Technology*, 2009.

[11] Hewlett-Packard. HP ProLiant DL980 G7 server with HP PREMA Architecture. Technical report, 2010.

[12] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/AVC Baseline Profile Decoder Complexity Analysis. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), 2003.

[13] N. Iqbal and J. Henkel. Efficient Constant-Time Entropy Decoding for H.264. In *Proc. Conf. Design, Automation Test in Europe*, 2009.

[14] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *Proc. ACM/IEEE Conf. on Supercomputing*, 2007.

[15] N. Ling. Expectations and Challenges for Next Generation Video Compression. In *Proc. 5th IEEE Conf. on Industrial Electronics and Applications*, 2010.

[16] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez. Parallel Scalability of Video Decoders. *Journal of Signal Processing Systems*, 57, November 2009.

[17] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *Proc. 14th Int. Conf. on Supercomputing*, 2000.

[18] K. Nishihara, A. Hatabu, and T. Moriyoshi. Parallelization of H.264 video decoder for embedded multicore processor. In *Proc. IEEE Int. Conf. on Multimedia and Expo*, 2008.

[19] M. Roitzsch. Slice-balancing H.264 video encoding for improved scalability of multicore decoding. In *Proc. 7th Int. Conf. on Embedded software*, 2007.

[20] F. H. Seitner, R. M. Schreier, M. Bleyer, and M. Gelautz. Evaluation of Data-Parallel Splitting Approaches for H.264 Decoding. In *Proc. 6th Int. Conf. on Advances in Mobile Computing and Multimedia*, 2008.

[21] K.-H. Sihn, H. Baik, J.-T. Kim, S. Bae, and H. J. Song. Novel Approaches to Parallel H.264 Decoder on Symmetric Multicore Systems. In *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, 2009.

[22] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[23] E. van der Tol, E. Jaspers, and R. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.

[24] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7), 2003.

[25] X264. A Free H.264/AVC Encoder. `http://www.videolan.org/developers/x264.html`.

[26] Xiph.org. `http://media.xiph.org/video/derf/`.