# A Highly Scalable Parallel Implementation of H.264*

Arnaldo Azevedo[1], Ben Juurlink[1], Cor Meenderinck[1], Andrei Terechko[2], Jan Hoogerbrugge[3], Mauricio Alvarez[4], Alex Ramirez[4,5], and Mateo Valero[4,5]

[1] Delft University of Technology, Delft, Netherlands
{Azevedo,Benj,Cor}@ce.et.tudelft.nl
[2] Vector Fabrics, Eindhoven, Netherlands
andrei@vectorfabrics.com
[3] NXP, Eindhoven, Netherlands
jan.hoogerbrugge@nxp.com
[4] Technical University of Catalonia (UPC), Barcelona, Spain
{alvarez,mateo}@ac.upc.edu
[5] Barcelona Supercomputing Center (BSC), Barcelona, Spain
alex.ramirez@bsc.es

**Abstract.** Developing parallel applications that can harness and efficiently use future many-core architectures is the key challenge for scalable computing systems. We contribute to this challenge by presenting a parallel implementation of H.264 that scales to a large number of cores. The algorithm exploits the fact that independent macroblocks (MBs) can be processed in parallel, but whereas a previous approach exploits only intra-frame MB-level parallelism, our algorithm exploits intra-frame as well as inter-frame MB-level parallelism. It is based on the observation that inter-frame dependencies have a limited spatial range. The algorithm has been implemented on a many-core architecture consisting of NXP TriMedia TM3270 embedded processors. This required to develop a subscription mechanism, where MBs are subscribed to the *kick-off lists* associated with the reference MBs. Extensive simulation results show that the implementation scales very well, achieving a speedup of more than 54 on a 64-core processor, in which case the previous approach achieves a speedup of only 23. Potential drawbacks of the 3D-Wave strategy are that the memory requirements increase since there can be many frames in flight, and that the frame latency might increase. Scheduling policies to address these drawbacks are also presented. The results show that these policies combat memory and latency issues with a negligible effect on the performance scalability. Results analyzing the impact of the memory latency, L1 cache size, and the synchronization and thread management overhead are also presented. Finally, we present performance requirements for entropy (CABAC) decoding.

## 1 Introduction

The demand for computational power increases continuously in the consumer market as it forecasts new applications such as Ultra High Definition (UHD)

* This work was performed while the fourth author was with NXP Semiconductors.

P. Stenström (Ed.): Transactions on HiPEAC IV, LNCS 6760, pp. 111–134, 2011.

video [1], 3D TV [2], and real-time High Definition (HD) video encoding. In the past this demand was mainly satisfied by increasing the clock frequency and by exploiting more instruction-level parallelism (ILP). Due to the inability to increase the clock frequency much further because of thermal constraints and because it is difficult to exploit more ILP, multicore architectures have appeared on the market.

This new paradigm relies on the existence of sufficient thread-level parallelism (TLP) to exploit the large number of cores. Techniques to extract TLP from applications will be crucial to the success of multicores. This work investigates the exploitation of the TLP available in an H.264 video decoder on an embedded multicore processor. H.264 was chosen due to its high computational demands, wide utilization, and development maturity and the lack of "mature" future applications. Although a 64-core processor is not required to decode a Full High Definition (FHD) video in real-time, real-time encoding remains a problem and decoding is part of encoding. Furthermore, emerging applications such as 3DTV are likely to be based on current video coding methods [2].

In previous work [3] we have proposed the 3D-Wave parallelization strategy for H.264 video decoding. It has been shown that the 3D-Wave strategy potentially scales to a much larger number of cores than previous strategies. However, the results presented there are analytical, analyzing how many macroblocks (MBs) could be processed in parallel assuming infinite resources, no communication delay, infinite bandwidth, and a constant MB decoding time. In other words, our previous work is a limit study.

In this paper, we make the following contributions:

- We present an implementation of the 3D-Wave strategy on an embedded multicore consisting of up to 64 TM3270 processors. Implementing the 3D-Wave turned out to be quite challenging. It required to dynamically identify inter-frame MB dependencies and handle their thread synchronization, in addition to intra-frame dependencies and synchronization. This led to the development of a subscription mechanism where MBs subscribe themselves to a so-called *Kick-off List* (KoL) associated with the MBs they depend on. Only if these MBs have been processed, processing of the dependent MBs can be resumed.
- A potential drawback of the 3D-Wave strategy is that the latency may become unbounded because many frames will be decoded simultaneously. A policy is presented that gives priority to the oldest frame so that newer frames are only decoded when there are idle cores.
- Another potential drawback of the 3D-Wave strategy is that the memory requirements might increase because of large number of frames in flight. To overcome this drawback we present a frame scheduling policy to control the number of frames in flight.
- We analyze the impact of the memory latency and the L1 cache size on the scalability and performance of the 3D-Wave strategy.

- The experimental platform features hardware support for thread management and synchronization, making it relatively light weight to submit/retrieve a task to/from the task pool. We analyze the importance of this hardware support by artificially increasing the time it takes to submit/retrieve a task.
- The 3D-Wave focuses on the MB decoding part of the H.264 decoding and assumes an accelerator for entropy decoding. We analyze the performance requirements of the entropy decoding accelerator not to harm the 3D-Wave scalability.

Parallel implementations of H.264 decoding and encoding have been described in several papers. Rodriguez et al. [4] implemented an H.264 encoder using Group of Pictures (GOP)- (and slice-) level parallelism on a cluster of workstations using MPI. Although real-time operation can be achieved with such an approach, the latency is very high.

Chen et al. [5] presented a parallel implementation that decodes several B frames in parallel. However, even though uncommon, the H.264 standard allows to use B frames as reference frames, in which case they cannot be decoded in parallel. Moreover, usually there are no more than 2 or 3 B frames between P frames. This limits the scalability to a few threads. The 3D-Wave strategy dynamically detects dependencies and automatically exploits the parallelism if B frames are not used as reference frames.

MB-level parallelism has been exploited in previous work. Van der Tol et al. [6] presented the exploitation of intra-frame MB-level parallelism and suggested to combine it with frame-level parallelism. If frame-level parallelism can be exploited is determined statically by the length of the motion vectors, while in our approach it is determined dynamically.

Chen et al. [5] also presented MB-level parallelism combined with frame-level parallelism to parallelize H.264 encoding. In their work, however, the exploitation of frame-level parallelism is limited to two consecutive frames and independent MBs are identified statically. This requires that the encoder limits the motion vector length. The scalability of the implementation is analyzed on a quad-core processor with Hyper-Threading Technology. In our work independent MBs are identified dynamically and we present results for up to 64 cores.

This paper is organized as follows. Section 2 provides an overview of MB parallelization technique for H.264 video decoding and the 3D-Wave technique. Section 3 presents the simulation environment and the experimental methodology to evaluate the 3D-Wave implementation. In Section 4 the implementation of the 3D-Wave on the embedded many-core is detailed. Also a frame scheduling policy to limit the number of frames in flight and a priority policy to reduce latency are presented. Extensive simulation results, analyzing the scalability and performance of the baseline 3D-Wave, the frame scheduling and frame priority policies, as well as the impacts of the memory latency, L1 cache size, parallelization overhead, and entropy decoding, are presented in Section 5. Conclusions are drawn in Section 6.

## 2 Thread-Level Parallelism in H.264 Video Decoding

Currently, H.264 [7] is one of the best video coding standard, in terms of compression and quality [8]. It has a compression improvement of over two times compared to previous standards such as MPEG-4 ASP, H.262/MPEG-2, etc. The H.264 standard was designed to serve a broad range of application domains ranging from low to high bitrates, from low to high resolutions, and a variety of networks and systems, e.g., internet streams, mobile streams, disc storage, and broadcast.

The coding efficiency gains of advanced video codecs such as H.264 come at the price of increased computational requirements. The computing power demand increases also with the shift towards high definition resolutions. As a result, current high performance uniprocessor architectures are not capable of providing the performance required for real-time processing [9,10]. Therefore, it is necessary to exploit parallelism. The H.264 codec can be parallelized either by a task-level or data-level decomposition.

In a *task-level decomposition* the functional partitions of the application such as vector prediction, motion compensation, and deblocking filter are assigned to different processors. Scalability is a problem because it is limited to the number of tasks, which typically is small. In a *data-level decomposition* the work (data) is divided into smaller parts and each part is assigned to a different processor. Each processor runs the same program but on different (multiple) data elements (SPMD). In H.264 data decomposition can be applied to different levels of the data structure. Only MB-level parallelism is described in this work; a discussion of the other levels can be found in [3].

In H.264, the motion vector prediction, intra prediction, and the deblocking filter kernels use data from neighboring MBs defining the dependencies shown in Fig. 1. Processing MBs in a diagonal wavefront manner satisfies all the dependencies and allows to exploit parallelism between MBs. We refer to this parallelization technique as 2D-Wave, to distinguish it from the 3D-Wave proposed in [3] and for which implementation results are presented in this work.

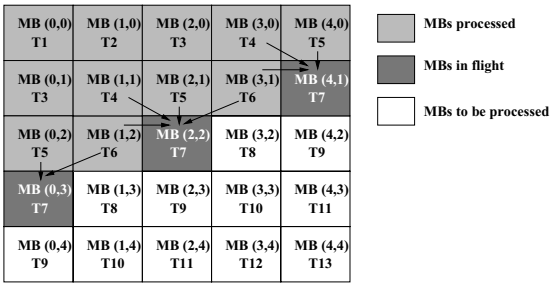



**Fig. 1.** 2D-Wave approach for exploiting MB parallelism. The arrows indicate dependencies.

Fig. 1 illustrates the 2D-Wave for an image of 5×5 MBs (80×80 pixels). At time slot T7 three independent MBs can be processed: MB(4,1), MB(2,2), and MB(0,3). The number of independent MBs varies over time. At the start it increases with one MB every two time slots, then stabilizes at its maximum, and finally decreases at the same rate it increased. For a low resolution like QCIF there are at most 6 independent MBs during 4 time slots. For Full High Definition (1920×1088) there are 60 independent MBs during 9 time slots.

MB-level parallelism has several advantages over other H.264 parallelization schemes. First, this scheme can have good scalability, since the number of independent MBs increases with the resolution of the image. Second, it is possible to achieve good load balancing if dynamic scheduling is used.

MB-level parallelism also has some disadvantages, however. The first is that entropy decoding can only be parallelized using data-level decomposition at slice-level, since the lowest level of data that can be parsed from the bitstream are slices. Only after entropy decoding has been performed the parallel processing of MBs can start. This disadvantage can be overcome by using special purpose instructions or hardware accelerators for entropy decoding. The second disadvantage is that the number of independent MBs is low at the start and at the end of decoding a frame. Therefore, it is not possible to sustain a certain processing rate during frame decoding.

The 2D-Wave technique, however, does not scale scales to future many-core architectures containing 100 cores or more, unless extremely high resolution frames are used. We have proposed [3] a parallelization strategy that combines intra-frame MB-level parallelism with inter-frame MB-level parallelism and which reveals the large amount of TLP required to harness and effectively use future many-core CMPs. The key points are described below.

In H.264 decoding there is only an inter-frame dependency in the Motion Compensation module. When the reference area has been decoded, it can be used by the referencing frame. Thus it is not necessary to wait until a frame is completely decoded before starting to decode the next frame. The decoding process of the next frame can start after the reference areas of the reference frames have been decoded. Fig. 2 illustrates this strategy called the 3D-Wave.

In our previous study the FFMPEG H.264 decoder [3] was modified to analyze the available parallelism for real movies. The experiments did not consider any practical or implementation issues, but explored the limits to the parallelism

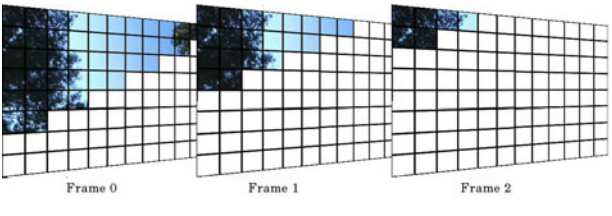



**Fig. 2.** 3D-Wave strategy: frames can be decoded in parallel because inter-frame dependencies have a limited spatial range

available in the application. The results show that the number of parallel MBs exhibited by the 3D-Wave ranges from 1202 to 1944 MBs for SD resolution ($720 \times 576$), from 2807 to 4579 MBs for HD ($1280 \times 720$), and from 4851 to 9169 MBs for FHD ($1920 \times 1088$). To sustain this amount of parallelism, the number of frames in flight ranges from 93 to 304 depending on the input sequence and the resolution. So, theoretically, the parallelism available on 3D-Wave technique is huge. There are many factors in real systems, however, such as the memory hierarchy and bandwidth, that could limit its scalability. In the next sections the approach to implement the 3D-Wave and exploit this parallelism on an embedded manycore system is presented.

## 3 Experimental Methodology

In this section the tools and methodology to implement and evaluate the 3D-Wave technique are detailed. Components of the many-core system simulator used to evaluate the technique are also presented.

An NXP proprietary simulator based on SystemC is used to run the application and collect performance data. Computations on the cores are modeled cycle-accurate. The memory system is modeled using average transfer times with channel and bank contention. When channel or bank contention is detected, the traffic latency is increased. NoC contention is supported. The simulator is capable of simulating systems with up to 64 TM3270 cores with shared memory and its cache coherence protocols. The operating system is not simulated.

The TM3270 [11] is a VLIW-based media-processor based on the Trimedia architecture. It addresses the requirements of multi-standard video processing at standard resolution and the associated audio processing requirements for the consumer market. The architecture supports VLIW instructions with five guarded issue slots. The pipeline depth varies from 7 to 12 stages. Address and data words are 32 bits wide. the unified register file has 128 32-bit registers. 2×16-bit and 4×8-bit SIMD instruction are supported. The TM3270 processor can run at up to 350 MHz, but in this work the clock frequency is set to 300 MHz. To produce code for the TM3270 the state-of-the-art highly optimizing NXP TriMedia C/C++ compiler version 5.1 is used.

The modeled system features a shared memory using MESI cache coherence. Each core has its own L1 data cache and can copy data from other L1 caches through 4 channels. The 64Kbyte L1 data cache has 64-byte lines and is 4-way set-associative with LRU replacement and write allocate. The instruction cache is not modeled. The cores share a distributed L2 cache with 8 banks and an average access time of 40 cycles. The average access time takes into account L2 hits, misses, and interconnect delays. L2 bank contention is modeled so two cores cannot access the same bank simultaneously.

The multi-core programming model follows the task pool model. A Task Pool (TP) library implements submissions and requests of tasks to/from the task pool, synchronization mechanisms, and the task pool itself. In this model there is a main core and the other cores of the system act as slaves. Each slave runs

a thread by requesting a task from the TP, executing it, and requesting another task. The task execution overhead is low. The time to request a task is less than 2% of the MB decoding time.

The experiments focus on the baseline profile of the H.264 standard. This profile only supports I and P frames and every frame can be used as a reference frame. This feature prevents the exploitation of frame-level parallelization techniques such as the one described in [5]. However, this profile highlights the advantages of the 3D-Wave, since the scalability gains come purely from the application of the 3D-Wave technique. Encoding was done with the X264 encoder [12] using the following options: no B-frames, at most 16 reference frames, weighted prediction, hexagonal motion estimation algorithm with a maximum search range of 24, and one slice per frame. The experiments use all four videos from the HD-VideoBench [13], Blue Sky, Rush Hour, Pedestrian, and Riverbed, in the three available resolutions, SD, HD and FHD.

The 3D-Wave technique focuses on the TLP available in the MB processing kernels of the decoder. The entropy decoder is known to be difficult to parallelize. To avoid the influence of the entropy decoder, its output has been buffered and its decoding time is not taken into account. Although not the main target, the 3D-Wave also eases the entropy decoding challenge. Since entropy decoding dependencies do not cross slice/frame borders, multiple entropy decoders can be used. We analyze the performance requirements of an entropy decoder accelerator in Section 5.7.

## 4 Implementation

In this work we use the NXP H.264 decoder. The 2D-Wave parallelization strategy has already been implemented in this decoder [14], making it a perfect starting point for the implementation of the 3D-Wave. The NXP H.264 decoder is highly optimized, including both machine-dependent optimizations (e.g. SIMD operations) and machine-independent optimizations (e.g. code restructuring).

The 3D-Wave implementation serves as a proof of concept thus the implementation of all features of H.264 is not necessary. Intra prediction inputs are deblock filtered samples instead of unfiltered samples as specified in the standard. This does not add visual artifacts to the decoded frames or change the MB dependencies.

This section details the 2D-Wave implementation used as the starting point, the 3D-Wave implementation, and the frame scheduling and priority policies.

### 4.1 2D-Wave Implementation

The MB processing tasks are divided in four kernels: vector prediction (VP), picture prediction (PP), deblocking info (DI), and deblocking filter (DF). VP calculates the motion vectors (MVs) based on the predicted motion vectors of the neighbor MBs and the differential motion vector present in the bitstream. PP performs the reconstruction of the MB based on neighboring pixel information

(Intra Prediction) or on reference frame areas (Motion Compensation). Inverse quantization and the inverse DCT are also part of this kernel. DI calculates the strength of the DF based on MB data, such as the MBs type and MVs. DF smoothes block edges to reduce blocking artifacts.

The 2D-Wave is implemented per kernel. By this we mean that first VP is performed for all MBs in a frame, then PP for all MBs, etc. Each kernel is parallelized as follows. Fig. 1 shows that within a frame each MB depends on at most four MBs. These dependencies are covered by the dependencies from the left MB to the current MB and from the upper-right MB to the current MB, i.e., if these dependencies are satisfied then all dependencies are satisfied. Therefore, each MB is associated with a reference count between 0 and 2 representing the number of MBs it depends on. For example, the upper-left MB has a reference count of 0, the other MBs at the top edge have a reference count of 1, and so do the other MBs at the left edge. When a MB has been processed, the reference counts of the MBs that depend on it are decreased. When one of these counts reaches zero, a thread that will process the associated MB is submitted to the TP. Fig. 3 depicts pseudo C-code for deblocking a frame and for deblocking a MB.

When a core loads a MB in its cache, it also fetches neighboring MBs. Therefore, locality can be improved if the same core also processes the right MB. To increase locality and reduce task submission and acquisition overhead, the

```
int deblock_ready[w][h];          // matrix of reference counts

void deblock_frame() {
   for (x=0; x<w; x++)
      for (y=0; y<h; y++)
         deblock_ready[x][y] = initial reference count; // 0, 1, or 2
   tp_submit(deblock_mb, 0, 0);  // start 1st task MB<0,0>
   tp_wait();
}

void deblock_mb(int x, int y){
   // ... the actual work

   if (x!=0 && y!=h-1){
      new_value = tp_atomic_decrement(&deblock_ready[x-1][y+1], 1);
      if (new_value==0)
         tp_submit(deblock_mb, x-1, y+1);
   }
   if (x!=w-1){
      new_value = tp_atomic_decrement(&deblock_ready[x+1][y], 1);
      if (new_value==0)
         tp_submit(deblock_mb, x+1, y);
   }
}
```

**Fig. 3.** Pseudo-code for deblocking a frame and a MB

```
void deblock_mb(int x, int y){
again:
   // ... the actual work

   ready1 = x>=1 && y!=h-1 && atomic_dec(&deblock_ready[x-1][y+1])==0;
   ready2 = x!=w-1 && atomic_dec(&deblock_ready[x+1][y])==0;

   if (ready1 && ready2){
      tp_submit(deblock_mb, x-1, y+1);    // submit left-down block
      x++;
      goto again;                         // goto right block
   }
   else if (ready1){
      x--; y++;
      goto again;                         // goto left-down block
   }
   else if (ready2){
      x++;
      goto again;                         // goto right block
   }
}
```

**Fig. 4.** Tail submit

2D-Wave implementation features an optimization called *tail submit*. After the MB is processed, the reference counts of the MB candidates are checked. If both MB candidates are ready to execute, the core processes the right MB and submits the other one to the task pool. If only one MB is ready, the core starts its processing without submitting or acquiring tasks to/from the TP. In case there is no neighboring MB ready to be processed, the task finishes and the core request another one from the TP. Fig. 4 depicts pseudo-code for MB decoding after the tail submit optimization has been performed. `atomic_dec` atomically decrements the counter and returns its value. If the counter reaches zero, the MB dependencies are met.

### 4.2 3D-Wave Implementation

In this section the 3D-Wave implementation is described. First we note that the original structure of the decoder is not suitable for the 3D-Wave strategy, because inter-frame dependencies are satisfied only after the DF is applied. To implement the 3D-Wave, it is necessary to develop a version in which the kernels are applied on a MB basis rather than on a slice/frame basis. In other words, we have a function `decode_mb` that applies each kernel to a MB.

Since the 3D-Wave implementation decodes multiple frames concurrently, modifications to the Reference Frame Buffer (RFB) are required. The RFB stores the decoded frames that are going to be used as reference. As it can serve only one frame in flight, the 3D-Wave would require multiple RFBs. In this proof of

```
void decode_mb(int x, int y, int skip, int RMB_start){
    IF (!skip) {
        Vector_Prediction(x,y);
        RMB_List = RMB_Calculation(x,y);
    }
    FOR RMB = RMB_List.table[RMB_start] TO
                              RMB_List.table[RMB_last]{
        IF !RMB.Ready {
            RMB.Subscribe(x, y);
            return;
        }
    }
    Picture_Prediction(x,y);
    Deblocking_Info(x,y);
    Deblocking_Filter(x,y);
    Ready[x][y] = true;

    FOR MB = KoL.start TO KoL.last
        tp_submit(decode_mb, MB.x, MB.y, true, MB.RMB_start);
    //TAIL_SUBMIT
}
```

**Fig. 5.** Pseudo-code for 3D-Wave

concept implementation, the RFB was modified such that a single instance can serve all frames in flight. In the new RFB all the decoded frames are stored. The mapping of the reference frame index to RFB index was changed accordingly.

Fig. 5 depicts pseudo-code for the `decode_mb` function. It relies on the ability to test if the reference MBs (RMBs) of the current MB have already been decoded or not. The RMB is defined as the MB in the bottom right corner of the reference area, including the extra samples for fractional motion compensation. To be able to test this, first the RMBs have to be calculated. If an RMB has not been processed yet, a method is needed to resume the execution of this MB after the RMB is ready.

The RMBs can only be calculated after motion vector prediction, which also defines the reference frames. Each MB can be partitioned in up to four 8×8 pixel areas and each one of them can be partitioned in up to four 4×4 pixel blocks The 4×4 blocks in an 8×8 partition share the reference frame. With the MVs and reference frames information, it is possible to calculate the RMB of each MB partition. This is done by adding the MV, the size of the partition, the position of the current MB, and the additional area for fractional motion compensation and by dividing the result by 16, the size of the MB. The RMB results of each partition is added to a list associated with the MB data structure, called the `RMB-list`. To reduce the number of RMBs to be tested, the reference frame of each RMB is checked. If two RMBs are in the same reference frame, only the one with the larger 2D-Wave decoding order (see Fig. 1) is added to the list.

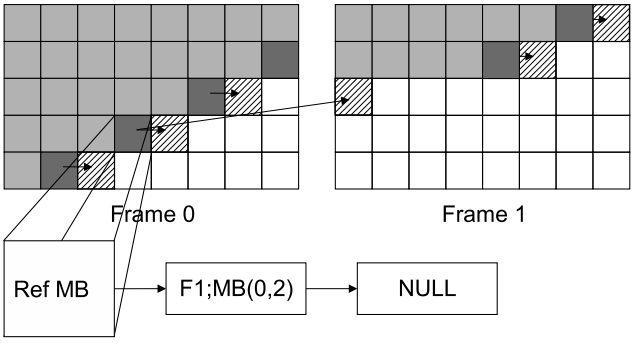



**Fig. 6.** Illustration of the 3D-Wave and the subscription mechanism

The first time `decode_mb` is called for a specific MB it is called with the parameter `skip` set to `false` and `RMB_start` set to `0`. If the decoding of this MB is resumed, it is called with the parameter `skip` set to `true`. Also `RMB_start` carries the position of the MB in the RMB-list to be tested next.

Once the RMB-list of the current MB is computed, it is verified if each RMB in the list has already been decoded or not. Each frame is associated with a MB ready matrix, similar to the `deblock_ready` matrix in Fig. 3. The corresponding MB position in the ready matrix associated with the reference frame is atomically checked. If all RMBs are decoded, the decoding of this MB can continue.

To handle the cases where a RMB is not ready, a RMB subscription technique has been developed. The technique was motivated by the specifics of the TP library, such as low thread creation overhead and no sleep/wake up capabilities. Each MB data structure has a second list called the Kick-off List (KoL) that contains the parameters of the MBs subscribed to this RMB. When a RMB test fails, the current MB subscribes itself to the KoL of the RMB and finishes its execution. Each MB, after finishing its processing, indicates that it is ready in the ready matrix and verifies its KoL. A new task is submitted to the TP for each MB in the KoL. The subscription process is repeated until all RMBs are ready. Finally, the intra-frame MBs that depend on this MB are submitted to the TP using tail submit, identical to Fig. 4.

Fig. 6 illustrates this process. Light gray boxes represent decoded MBs and dark gray boxes MBs that are currently being processed. Hatched boxes represent MBs available to be decoded while white boxes represent MBs whose dependencies have not yet been resolved. In this example MB(0,2) of frame 1 depends on MB(3,3) of frame 0 and is subscribed to the KoL of the latter. When MB(3,3) is decoded it submits MB(0,2) to the task pool.

### 4.3 Frame Scheduling Policy

To achieve the highest speedup, all frames of the sequence are scheduled to run as soon as their dependencies are met. However, this can lead to a large number

of frames in flight and large memory requirements, since every frame must be kept in memory. Mostly it is not necessary to decode a frame as soon as possible to keep all cores busy. A frame scheduling technique was developed to keep the working set to its minimum.

Frame scheduling uses the RMB subscription mechanism to define the moment when the processing of the next frame should be started. The first MB of the next frame can be subscribed to start after a specific MB of the current frame. With this simple mechanism it is possible to control the number of frames in flight. Adjusting the number of frames in flight is done by selecting an earlier or later MB with which the first MB of the next frame will be subscribed.

### 4.4 Frame Priority

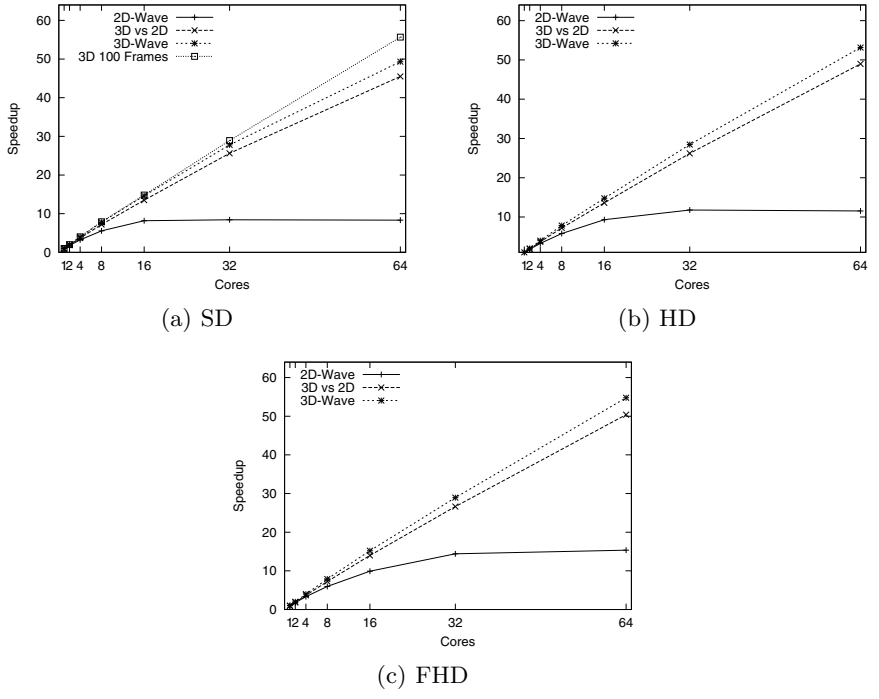
Latency is an important characteristic of video decoding systems. The frame scheduling policy described in the previous section reduces the frame latency, since the next frame is scheduled only when a part of the current frame has been decoded. However, when a new frame is scheduled to be decoded, the available cores are distributed equally among the frames in flight. A priority mechanism was added to the TP library in order to reduce the frame latency even further.

The TP library was modified to support two levels of priority. An extra task buffer was implemented to store high priority tasks. When the TP receives a task request, it first checks if there is a task in the high priority buffer. If so this task is selected, otherwise a task in the low priority buffer is selected. With this simple mechanism it is possible to give priority to the tasks belonging to the frame "next in line". Before submitting a new task the process checks if its frame is the frame "next in line". If so the task is submitted with high priority. Otherwise it is submitted with low priority. This mechanism does not lead to starvation because if there is not sufficient parallelism in the frame "next in line" the low priority tasks are selected.

## 5 Experimental Results

In this section the experimental results are presented. The results include the scalability results of the 3D-Wave (Section 5.1), results of the frame scheduling and priority policies (Section 5.2), impact on the memory and bandwidth requirements (Section 5.3), influence of memory latency (Section 5.4), influence of L1 data cache size on scalability and performance (Section 5.5), the impact of parallelism overhead on scalability (Section 5.6), and the requirements for the CABAC accelerator to leverage a 64-core system (Section 5.7).

To evaluate the 3D-Wave, one second (25 frames) of each sequence was decoded. Longer sequences could not be used due to simulator constraints. The four sequences of the HD-VideoBench using three resolutions were evaluated. Since the result for Rush Hour sequence are close to the average and other sequences vary less than 5% only its results will be presented.




(a) SD

(b) HD

(c) FHD

**Fig. 7.** 2D-Wave and 3D-Wave speedups for the 25-frame sequence Rush Hour for different resolutions

### 5.1 Scalability

The scalability results are for 1 to 64 cores. More cores could not be simulated due to limitations of the simulator. Figs. 7(a), 7(b), and 7(c) depict the 3D-Wave scalability on $p$ processors ($T_{3D}(1)/T_{3D}(p)$), 2D-Wave scalability ($T_{2D}(1)/T_{2D}(p)$), and 3D-Wave versus 2D-Wave on a single core ($T_{2D}(1)/T_{3D}(p)$), labeled as 3D vs 2D. On a single core, 2D-Wave can decode 39 SD, 18 HD, and 8 FHD frames per second.

On a single core the 3D-Wave implementation takes 8% more time than the 2D-Wave implementation due to administrative overhead for all resolutions. The 3D-Wave scales almost perfectly up to 8 cores, while the 2D-Wave incurs an 11% efficiency drop even for 2 cores due to the following reason. The tail submit optimization assigns MBs to cores per line. At the end of a frame, when a core finishes its line and there is no other line to be decoded, in the 2D-Wave the core remains idle until all cores have finished their line. If the last line happens to be slow the other cores wait for a long time and the core utilization is low. In the 3D-Wave, cores that finish their line, while there is no new line to be decoded, will be assigned a line of the next frame. Therefore, the core utilization as well as the scalability efficiency of the 3D-Wave is higher. Another advantage of the 3D-Wave over the 2D-Wave is that it increases the efficiency of the Tail Submit

optimization. In the 2D-Wave the low available parallelism makes the cores stall more due to unsolved intra-frame dependencies. In the 3D-Wave, the available parallelism is much larger which increases the distance between the MBs being decoded, minimizing intra-frame dependency stalls.

For SD sequences, the 2D-Wave technique saturates at 16 cores, with a speedup of only 8. This happens because of the limited amount of MB parallelism inside the frame and the dominant ramp up and ramp down of the availability of parallel MBs. The 3D-Wave technique for the same resolution continuously scales up to 64 cores, with a parallelization efficiency of almost 80%. For the FHD sequence, the saturation of the 2D-Wave occurs at 32 cores while the 3D-Wave continuously scales up to 64 cores with a parallelization efficiency of 85%.

The scalability results of the 3D-Wave increase slightly for higher resolutions. On the other hand, the 2D-Wave implementation achieves higher speedups for higher resolutions since the MB-level parallelism inside a frame increases. However, it would take an extremely large resolution for the 2D-Wave to leverage 64 cores, and the 3D-Wave implementation would still be more efficient.

The drop in scalability efficiency of the 3D-Wave for larger number of cores has two reasons. First, cache trashing occurs for large numbers of cores, leading to many memory stalls, as will be show in the next section. Second, at the start and at the end of a sequence, not all cores can be used because little parallelism is available. The more cores are used, the more cycles are relatively wasted during these two periods. It would be negligible in a real sequence with many frames. To show this Fig. 7(a) also shows the scalability results for 100 frames of the Rush Hour SD sequence. Simulation with HD or FHD sequences with more than 25 frames are not possible because the simulator cannot allocate the required data structures.

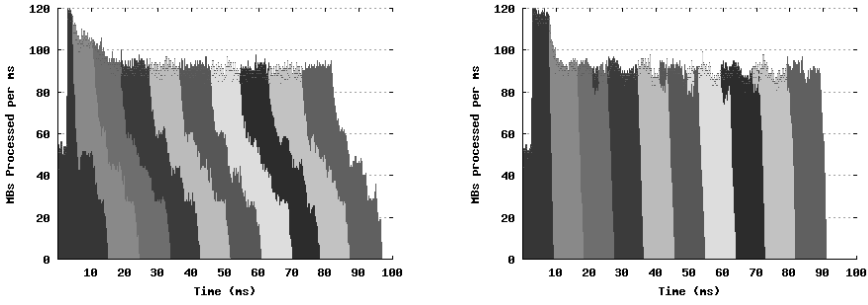
For 64 cores the scalability grows from 49.32 to 55.67 when processing 100 instead of 25 frames. The effects of ramp up and ramp down times are minimized when more frames are used. In this case, the scalability results are closer to the results that would be achieved in a real life situation.

### 5.2 Frame Scheduling and Priority

In this section, experimental results for the frame scheduling and priority policies are presented. The effectiveness of these policies is presented first, then the impact of these policies on the 3D-Wave efficiency.

Fig. 8(a) presents the results of the frame scheduling technique applied to the FHD Rush Hour sequence using a 16-core system. This figure presents the number of MBs processed per $ms$. It also shows to which frame these MBs belong. In this particular case, the subscribe MB chosen is the last MB on the line that is at 1/3rd of the frame. For this configuration there are at most 3 frames in flight. Currently, the selection of the subscribe MB must be done statically by the programmer. A methodology to dynamically fire new frames based on core utilization needs to be developed.

The priority mechanism presented in Section 4.4 strongly reduces the frame latency. In the original 3D-Wave implementation, the latency of the first frame




(a) Number of MBs processed per ms using frame scheduling and frames to which these MBs belong

(b) Number of MBs processed per ms using frame scheduling and the priority policy

**Fig. 8.** Results for frame scheduling and priority policy for FHD Rush Hour on a 16-core processor. Different gray scales represent different frames.

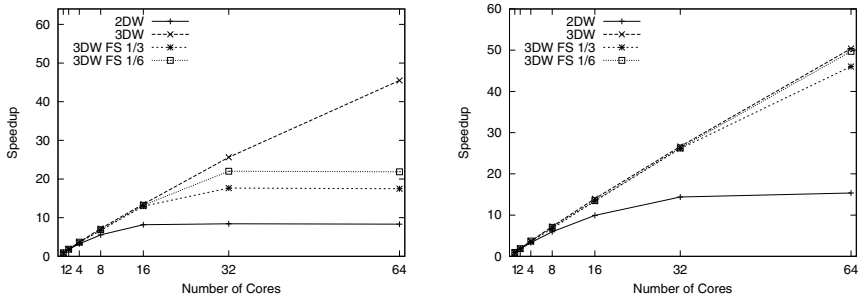
is 58.5 *ms*, using the FHD Rush Hour sequence with 16 cores. Using the frame scheduling policy, the latency drops to 15.1 *ms*. This latency is further reduced to 9.2 *ms* when the priority policy is applied together with frame scheduling. This is 0.1 *ms* longer than the latency of the 2D-Wave, which decodes frames one-by-one. Fig. 8(b) depicts the number of MBs processed per *ms* when this feature is used.

Two scenarios were used to analyze the impact of frame scheduling and priority on the scalability. The chosen scenarios use 3 and 6 frames in flight, with and without frame priority. Figs. 9(a) and 9(b) depict the impact of the presented techniques on the scalability. 2D-Wave (2DW) and 3D-Wave (3DW) scalability results are presented as guidelines. In Fig. 9, FS refers to the frame scheduling. The addition of frame priority has no significant impact on the scalability and therefore not shown, as it would decrease legibility. The reported scalability is based on the 2D-Wave execution time on a single core.

Fig. 9(a) shows that 6 frames in flight are not enough to leverage a 64-core system when decoding an SD sequence. The maximum speedup of 23 is the result of the relatively low amount of MB-level parallelism of SD frames. As presented in Fig. 7(a), the 2D-Wave has a maximum speedup of 8. For HD (figure not shown), the performance when 6 frames in flight are allowed is already close to the performance of the original 3D-Wave. The maximum speedups are 24 and 45, for three and six frames in flight, respectively. The latter is 92% of the maximum 3D-Wave speedup. For FHD, depicted in Fig. 9(b), allowing three frames in flight provides a speedup of 46. When 6 frames are used, the difference between the frame scheduler enabled and the original 3D-Wave is only 1%.

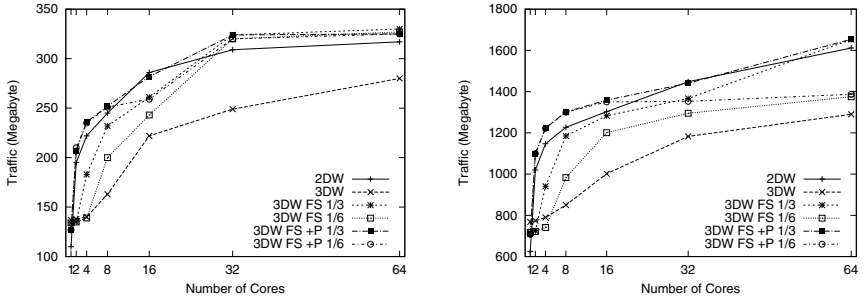
### 5.3 Bandwidth Requirements

In this section, the intra-chip bandwidth requirements for the 3D-Wave and its frame scheduling and priority policies are reported. The amount of data traffic




(a) Scalability for SD (b) Scalability for FHD

**Fig. 9.** Frame scheduling and priority scalability results of the Rush Hour 25-frame sequence




(a) SD (b) FHD

**Fig. 10.** Frame scheduling and priority data traffic results for Rush Hour sequence

between L2 and L1 data caches is measured. Accesses to main memory are not reported by the simulator.

The effects of frame scheduling and priority policies on data traffic between L2 and L1 data caches are depicted in Fig. 10(a) and 10(b). The figures depict the required data traffic for SD and FHD resolutions, respectively. In the figures, FS refers to the frame scheduling while P refers to the use of frame priority.

Data locality decreases as the number of cores increases, because the task scheduler does not take into account data locality when assigning a task to a core (except with the tail submit strategy). This decrease in the locality contributes to traffic increase. Due to these effects, the 3D-Wave increases the data traffic by approximately 104%, 82%, and 68% when going from 1 to 64 cores, for SD, HD, and FHD, respectively.

Surprisingly, the original 3D-Wave requires the least communication between L2 and L1 data caches for 8 cores or more. It is approximately 20% to 30% (from SD to FHD) more data traffic efficient than the original 2D-Wave, for 16 cores or more. This is caused by the high data locality of the original 3D-Wave technique.

The 3D-Wave implementation fires new frames as soon as their dependencies are met. This increases the probability of the reference areas of a MB to be present in the system. The probability increases because nearby area of several frames are decoded together, so the reference area is still present in data caches of other cores. This reduces the data traffic because the motion compensation (inter-frame dependency) requires a significant portion of data to be copied from previous frames.

The use of FS and Priority has a negative impact on the L2 to L1 data cache traffic. The use of FS and Priority decreases the data locality, as they increase the time between processing MBs from co-located areas of consecutive frames. However, when the number of frames are enough to sustain a good scalability, the increased data traffic when using FS and Priority is still lower than the data traffic of 2D-Wave implementation. For SD, the data traffic for FS and Priority is higher than the 2D-Wave when the available parallelism is not enough to leverage for 32 and 64 cores. The same happens for the HD using only 3 frames in flight. For FHD, 2D-Wave is the technique that requires most data traffic, together with FS for 3 frames in flight. When the number of frames in flight are enough to leverage to 32 or 64 cores, FS is 4 to 12% more efficient than 2D-Wave. FS and Priority can be 3 to 6% data traffic less efficient than 2D-Wave in the cases when number of frames in flight are insufficient to leverage to the number of cores.

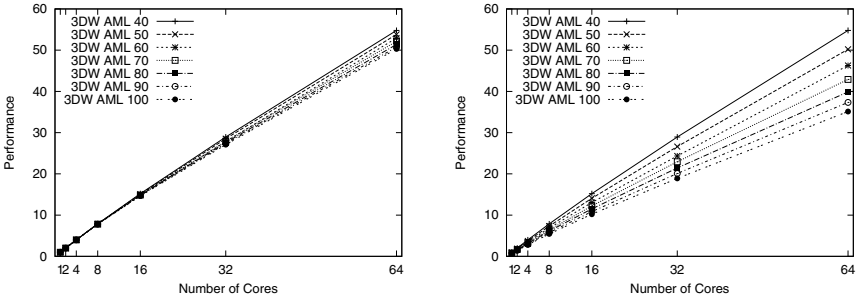
With the data traffic results it is possible to calculate the L2 to L1 bandwidth requirements. The bandwidth is calculated by dividing the total traffic by the time to decode the sequence in seconds. The total amount of intra chip bandwidth required for 64 cores is 21 GB/s for all resolutions of Rush Hour sequence. The bandwidth is independent of the resolution because the number of MBs decoded per time unit per core is the same.

### 5.4 Impact of the Memory Latency

The type of interconnection used, and the number of cores in the system both influence the memory latency. For increasing number of cores, also the latency of a L2 to L1 data transfer increases. In this section we analyze the impact of this latency on the performance. One second (25 frames) of the Rush Hour sequence, in all three available resolutions, was decoded while with several average memory latencies.

In the previous experiments the average L2 data cache latency was set to 40 cycles. In this experiment the Average Memory Latency (AML) ranges from 40 to 100 cycles in steps of 10 cycles. The latency of the interconnect between L1 and L2 is modelled by adding additional delay cycles to the AML of the L2 cache.

Fig. 11(a) depicts the scalability results for FHD resolution. That is, for each AML, the performance using X cores is compared to the performance using one core. The results show that the memory latency does not significantly affect the scalability. For 64 cores, increasing the AML from 40 to 100 cycles decreases the scalability by 10%. However, the scalability does not equal the absolute




(a) Scalability (b) Performance

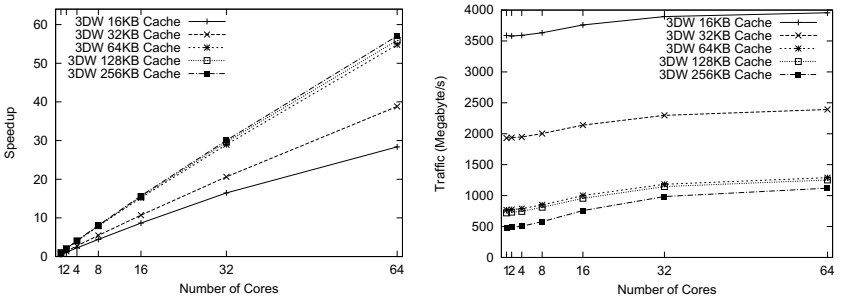
**Fig. 11.** Scalability and performance for different Average Memory Latency (AML) values, using the 25 frame Rush Hour FHD sequence. In the scalability graph the performance is relative to the execution on a single core, but with the same AML. In the performance graph all performances are relative to the execution on a single core with an AML of 40 cycles.

performance. In Fig. 11(b) the performance is depicted using the execution time one a single core with an AML of 40 cycles as baseline. The performances still scale the same, but the performance for one core is different for every line (in contrast to Fig. 11(a)). The graph shows that in total the system's performance is decreased significantly. That means that large systems might be infeasible if the memory latency increases too much.

### 5.5 Impact of the L1 Cache Size

We analyzed the influence of the L1 data cache size on the scalability and the amount of L2-L1 traffic. The baseline system has L1 data caches of 64KB, 4-way set-associative, with LRU replacement, and write allocate. By modifying the number of sets in the cache systems with different cache sizes, i.e., 12, 32, 64, 128, and 256KB, were simulated. The results for FHD resolution are depicted in Fig. 12(a). The depicted performance are relative to the decoding time on a single core with the baseline 64KB L1 cache.

The systems with 16KB and 32KB caches have a large performance drop of approximately 45% and 30%, respectively, for any number of cores. The reason for this is depicted in Fig. 12(b), which presents the L1-L2 cache data traffic for FHD resolution. Compared to a system with 64KB caches, the system with 16KB caches has between 3.1 and 4.7 times more traffic while the system with 32KB caches has between 1.8 and 2.5 times more traffic. Those huge increases in data traffic are due to cache misses. For FHD resolution, one MB line occupies 45KB. Preferably, the caches should be able to store more then one MB line, as the data of each line is used in the decoding of the next line and serves as input for the motion compensation in the next frames. For FHD, the 16KB and 32 KB caches suffer greatly from data trashing. As a result there are a lot of write backs to the the L2 cache as well as reads. For smaller resolutions the effects are less.




(a) Scalability (b) Data traffic

**Fig. 12.** Impact of the L1 cache size on performance and L1-L2 traffic for a 25 frame Rush Hour FHD sequence

For example, for SD resolution using 16KB L1 caches the data traffic increases between 1.19 and 1.66 compared to the baseline with 64KB caches. With 32KB caches, the traffic increases only by approximately 7%.

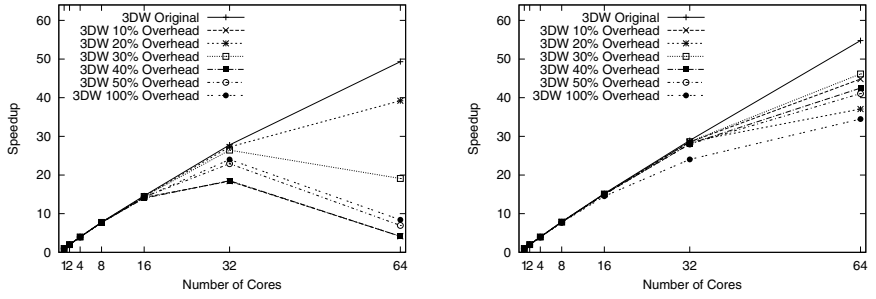
Using caches larger than 64KB provides small performance gains (up to 4%). The reason for this is again the size of a MB line. Once the dataset fits in the cache, the application behaves like a memory stream application and makes no use of the additional memory space. This is also reflected in the data traffic graph. For FHD, the system with 256KB caches has 13 to 27% less traffic than the 64KB system. For the lower resolutions, the traffic is reduced by at most 10% and the performance gain is at most 4%.

### 5.6 Impact of the Parallelization Overhead

Alvarez et al. [15] implemented the 2D-Wave approach on an architecture with 64 dual core IA-64 processors. Their results show that the scalability is severely limited by the thread synchronization and management overhead, i.e., the time it takes to submit/retrieve a task to/from the task pool. On their platform it takes up to 9 times as long to submit/retrieve a task as it takes to decode a MB. To analyze the impact of the TLP overhead on the scalability of the 3D-Wave, we replicate this TLP overhead by increasing the Task Pool Library by adding dummy calculation.

The inserted extra overheads are 10%, 20%, 30%, 40%, 50%, and 100% of the average MB decoding time, which is 4900 cycles. Because of the Tail Submit enhancement not every MB requests or submits a task to the Thread Pool. This causes a total performance overhead of only 3% for a single core when comparing the 100% TPL overhead against the baseline 3D-Wave. The effects of this increased overhead is depicted on Figs. 13(a), and 13(b), for SD and FHD resolutions, respectively.

The results for SD resolution show the impact of the increase overhead on the scalability. For 32 cores the scalability is considerably reduced when the overhead is 40% or more. For 64 cores the effects of the extra overhead reduces




(a) SD (b) FHD

**Fig. 13.** TPL overhead effects on scalability for Rush Hour frames

the scalability. The SD resolution is very affected with the increased overhead because the intra frame resolution is comparatively low and the lines are short, which increases task submition per frame. For the HD resolution (figure not shown) the increase in overhead limits the scalability to 32 cores while for FHD it slows down the scalability, but does not limit it. As the resolution increases the requests to TPL per MB ratio decreases and so the impact of the extra overhead. These results show the drastic effects of the overhead on the scalability, even with enhancements that reduces the requests to the parallelization support.

### 5.7 CABAC Accelerator Requirements

Broadly speaking, H.264 decoding consists of two parts: entropy (CABAC) decoding and MB decoding. CABAC decoding of a single slice/frame is largely sequential while in this paper we have shown that MB decoding is highly parallel. We therefore assumed that CABAC decoding is performed by a specific accelerator. In this section we evaluate the performance required of such an accelerator to allow the 3D-Wave to scale to a large number of cores.

Fig. 14 depicts the speedup as a function of the number of (MB decoding) cores for different speeds of the CABAC accelerator. The baseline accelerator, corresponding to the accelerator labeled "no speedup", is assumed to have the same performance as the TM3270 TriMedia processor. These results were obtained using a trace-driven, abstract-level simulator that schedules the threads given the CABAC and MB dependencies and their processing times. The traces have been obtained using the simulator described in Section 3 and used in the previous sections.

The results show that if CABAC decoding is not accelerated, then the speedup is limited to 7.5, no matter how many cores are employed. Quadrupling the speed of the CABAC accelerator improves the overall performance by a similar factor, achieving a speedup of almost 30 on 64 cores. When CABAC decoding is accelerated by a factor of 8, the speedup of 53.8 on 64 cores is almost the same as the results presented previously which did not consider CABAC. There are several proposals [16] that achieve such a speedup for CABAC decoding. This shows

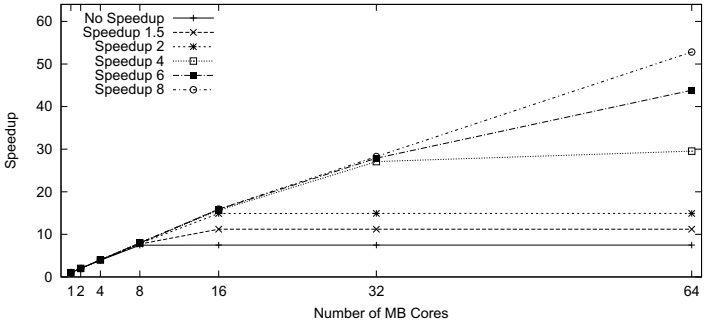



**Fig. 14.** Maximum scalability per CABAC processor and accelerators

that the CABAC processing does not pose a limitation on the scalability of the 3D-Wave technique. We remark that the 3D-Wave also allows employing multiple CABAC accelerators, since different slices/frames can be CABAC decoded in parallel, as entropy-decoding dependencies do not cross slice/frame borders.

## 6 Conclusions

Future CMPs will contain dozens if not hundreds of cores. For such systems, developing parallel applications that can harness them is the key challenge. In this paper we have contributed to this challenge by presenting a highly scalable parallel implementation of H.264 decoding. While a many-core is not necessary to achieve real-time FHD video decoding, it is likely that future video coding standards will be computationally more intensive and will be similarly block-oriented motion-compensation-based. Furthermore, decoding is part of encoding and real-time encoding is still a challenge. In addition, emerging applications such as 3D TV are likely to be based on current video coding standards.

While the idea behind the 3D-Wave was presented in our previous work, in this paper we have contributed by providing an actual implementation and by providing exhaustive simulation results. Implementing the 3D-Wave required, for example, developing a subscription mechanism where MBs are subscribed to a so-called Kick-off List associated with the MBs in the reference frame(s) they depend on. Several optimizations have been performed to reduce the overhead of this mechanism. For example, vector prediction is skipped if it has already been performed and if two reference MBs are in the same reference frame, only the one that will be decoded last is added to the list.

The simulation results show that the 3D-Wave implementation scales almost perfectly up to 64 cores. More cores could not be simulated due to limitations of the simulator. Furthermore, one of the main reasons why the speedup is slightly less than linear is that at the beginning and at the end of decoding a sequence of 25 frames, not all the cores can be used because little TLP is available. In a real sequence these periods are negligible. The presented frame scheduling and

priority policies reduce the number of frames in flight and the frame latency. By applying these policies, the frame latency of the 3D-Wave is only 0.1 ms (about 1%) longer than that of the 2D-Wave.

We also measured the amount of data traffic shared L2 and the private L1 data caches. Obviously, increasing the number of cores increases the L2-L1 data traffic, since the cores have to communicate via the L2 cache. 64 cores generate approximately the same amount of L2-L1 traffic as 32 cores, however, and both produce roughly twice as much traffic as a single core. To our initial surprise, the original 3D-Wave generates the least amount of L2-L1 data traffic. This is because the original 3D-Wave exploits the data reuse between a MB and its reference MBs, more so than the 2D-Wave and the 3D-Wave with frame scheduling and priority.

Next we have analyzed the impact of the memory latency and of the L1 cache size. While increasing the average memory latency (AML) hardly affects the scalability (i.e., the speedup of the 3D-Wave running on $p$ cores over the 3D-Wave running on a single core), it of course reduces the performance. Doubling the AML from 40 to 80 cycles reduces the performance on 64 cores by approximately 25%. The results for different L1 data cache sizes show that a 64KB data cache is necessary and sufficient to keep the active working set in cache. Smaller L1 data caches significantly reduce the performance, while larger L1 data caches provide little improvement. The reason is that a single line of MBs is 45KB for FHD and, therefore, caches larger than 45KB can exploit the data reuse between a MB line and the next MB line.

In addition, we have analyzed the impact of the parallelization overhead by artificially increasing the time it takes to submit/retrieve a task to/from the task pool. The 3D-Wave exploits medium-grain TLP (decoding a single MB takes roughly 5000 cycles on the TM3270), so task submission/retrieval should not take too much time. Because of the tail submit optimization, however, not for every MB a task is submitted to the task pool. The results show that even when the parallelization overhead is 50% of the MB decoding time (about 2500 cycles), the speedup on 64 cores is still higher than 41 for FHD. For SD, because it exhibits less TLP and therefore submits more tasks per MB, the effects are more dramatic.

Finally, we have analyzed the performance required of a CABAC accelerator so that CABAC decoding does not become the bottleneck that limits the scalability of the 3D-Wave. The results show that if CABAC decoding is performed by a core with the same speed as the other cores, then the speedup is limited to 7.5, no matter how many cores are employed. If CABAC decoding is accelerated by a factor of 8, however, the speedup for 64 cores is almost the same as when CABAC decoding is not considered but performed beforehand.

Future work includes the development of an automatic frame scheduling technique that only starts to decode a new frame if some cores are idle because of insufficient TLP, the implementation of the 3D-Wave on other platforms such as the Cell and GPGPUs, and the implementation of the 3D-Wave in the encoder. A 3D-Wave implementation of the encoder would provide high definition, low latency encoding on multicores.

## Acknowledgment

This work was performed while the first author was visiting NXP Semiconductors and it was partially supported by the European Commission in the context of the SARC integrated project #27648 (FP6), the Ministry of Science of Spain and European Union (FEDER funds) under contract TIC-2004-07739-C02-01, and the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). The authors would like to thank Anirban Lahiri from NXP for his collaboration on the experiments.

## References

1. Okano, F., Kanazawa, M., Mitani, K., Hamasaki, K., Sugawara, M., Seino, M., Mochimaru, A., Doi, K.: Ultrahigh-Definition Television System With 4000 Scanning Lines. In: Proc. of NAB Broadcast Engineering Conf., pp. 437–440 (2004)
2. Drose, M., Clemens, C., Sikora, T.: Extending Single-View Scalable Video Coding to Multi-View Based on H. 264/AVC. In: 2006 IEEE Inter. Conf. on Image Processing, pp. 2977–2980 (2006)
3. Meenderinck, C., Azevedo, A., Juurlink, B., Alvarez, M., Ramirez, A.: Parallel Scalability of Video Decoders. Journal of Signal Processing Systems (August 2008)
4. Rodriguez, A., Gonzalez, A., Malumbres, M.P.: Hierarchical Parallelization of an H.264/AVC Video Encoder. In: Proc. Int. Symp. on Parallel Computing in Electrical Engineering, pp. 363–368 (2006)
5. Chen, Y.K., Li, E.Q., Zhou, X., Ge, S.: Implementation of H.264 Encoder and Decoder on Personal Computers. Journal of Visual Communications and Image Representation 17 (2006)
6. van der Tol, E., Jaspers, E., Gelderblom, R.: Mapping of H.264 Decoding on a Multiprocessor Architecture. In: Proc. SPIE Conf. on Image and Video Communications and Processing (2003)
7. International Standard of Joint Video Specification (ITU-T Rec. H. 264— ISO/IEC 14496-10 AVC) (2005)
8. Oelbaum, T., Baroncini, V., Tan, T.K., Fenimore, C.: Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard. In: Int. Broadcast Conf., IBC (2004)
9. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: A Performance Characterization of High Definition Digital Video Decoding using H.264/AVC. In: Proc. IEEE Int. Workload Characterization Symp., pp. 24–33 (2005)
10. Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., Wedi, T.: Video Coding with H.264/AVC: Tools, Performance, and Complexity. IEEE Circuits and Systems Magazine 4(1), 7–28 (2004)
11. van de Waerdt, J., Vassiliadis, S., Das, S., Mirolo, S., Yen, C., Zhong, B., Basto, C., van Itegem, J., Amirtharaj, D., Kalra, K., et al.: The TM3270 Media-Processor. In: MICRO 2005: Proc. of the 38th Inter. Symp. on Microarchitecture, pp. 331–342 (November 2005)
12. X264. A Free H.264/AVC Encoder
13. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In: Proc. IEEE Int. Symp. on Workload Characterization (2007)

14. Hoogerbrugge, J., Terechko, A.: A Multithreaded Multicore System for Embedded Media Processing. Trans. on High-Performance Embedded Architectures and Compilers 4(2) (2009)
15. Alvarez, M., Ramirez, A., Valero, M., Meenderinck, C., Azevedo, A., Juurlink, B.: Performance Evaluation of Macroblock-level Parallelization of H.264 Decoding on a CC-NUMA Multiprocessor Architecture. In: Proc. of the 4CCC: 4th Colombian Computing Conf. (April 2009)
16. Osorio, R.R., Bruguera, J.D.: An FPGA Architecture for CABAC Decoding in Manycore Systems. In: Proc. of IEEE Application-Specific Systems, Architectures and Processors, pp. 293–298 (July 2008)