Parallel H.264 Decoding on an Embedded Multicore Processor

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

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

Abstract. In previous work the 3D-Wave parallelization strategy was proposed to increase the parallel scalability of H.264 video decoding. This strategy is based on the observation that inter-frame dependencies have a limited spatial range. The previous results, however, investigate application scalability on an idealized multiprocessor. This work presents an implementation of the 3D-Wave strategy on a multicore architecture composed of NXP TriMedia TM3270 embedded processors. The results show that the parallel H.264 implementation scales very well, achieving a speedup of more than 54 on a 64-core processor. Potential drawbacks of the 3D-Wave strategy are that the memory requirements increase since there can be many frames in flight, and that the latencies of some frames might increase. To address these drawbacks, policies to reduce the number of frames in flight and the frame latency are also presented. The results show that our policies combat memory and latency issues with a negligible effect on the performance scalability.

1 Introduction

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

Compared to [3], 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.

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 and it introduces a frame scheduling policy to limit the number of frames in flight and describes a priority policy to reduce latency. The results of the 3D-Wave, the frame scheduling and frame priority policies are presented in Section 5. Conclusions are drawn in Section 6.

2 Thread-Level Parallelism in H.264 Video Decoding

Currently, one of the best video coding standard, in terms of compression and quality is H.264 [7]. The coding efficiency gains of advanced video codecs such as H.264 come at the price of increased computational requirements. The demands for computing power increases also with the shift towards high definition resolutions. As a result, current high performance uniprocessor architectures are not capable of providing the required performance for real-time processing [8,9]. 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 algorithm 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. Due to space limitations 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 a set of dependencies shown as arrows in Figure 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.

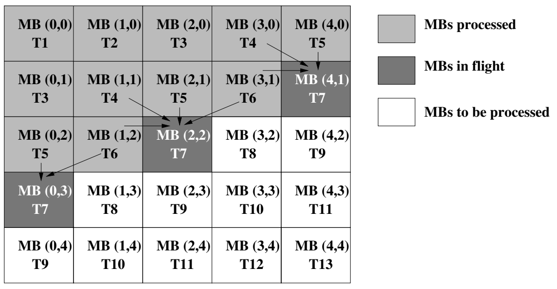


Fig. 1. 2D-Wave approach for exploiting MB parallelism. The arrows indicate dependencies.

Figure 1 illustrates the 2D-Wave for a 5×5 MBs image (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 of decoding a frame 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 (1920x1088) there are at most 60 independent MBs during 9 time slots.

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

However, MB-level parallelism has some disadvantages. The first one is that the entropy decoding cannot be parallelized using data decomposition, due to the fact that 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 the decoding of a frame.

None of the approaches described 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 MB-level with frame-level parallelism and which reveals the large amount of parallelism required to effectively use future many-core CMPs. The key points are described below.

In the decoding process the dependency between frames is in the Motion Compensation (MC) module only. 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 decoding the next frame. The decoding process of the next frame can be started after the reference areas of the reference frames are decoded. Figure 2 illustrates this way of parallel decoding of frames, called 3D-Wave strategy.

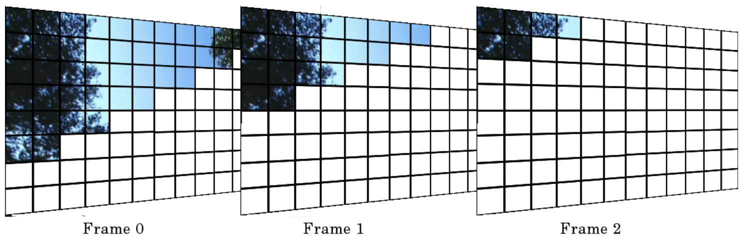


Fig. 2. 3D-Wave strategy: frames can be decoded in parallel because inter frame dependencies have limited spatial range

In our previous study the H.264 decoder was modified to analyze the available parallelism for real movies. The sequences of the HD-VideoBench [10] were used as inputs for the experiment. For each MB its dependencies were analyzed. Then for each timeslot we analyzed the number of MBs that can be processed in parallel during that time slot. The experiments did not consider any practical or implementation issues, but explored the limits to the parallelism available in the application.

The results show that the amount of MB-level parallelism exhibited by the 3D-Wave ranges from 1202 to 1944 MBs for SD resolution (720×576), from 2807 to 4579 MBs for HD resolution (1280×720), and from 4851 to 9169 MBs for FHD resolution (1920×1088). 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.

The theoretically available parallelism exhibited by the 3D-Wave technique is huge. However, there are many factors in a real system, 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 multicore system is presented.

3 Experimental Methodology

In this section the tools and methodology to implement and evaluate the 3D-Wave technique are detailed. Also components of the many-core system simulator used to evaluate the technique are 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 data transfer times with channel and bank contention detection that modifies the latency to transfer the data. The simulator is capable of simulating systems with 1 to 64 TM3270 cores with shared memory and its cache coherence protocols. The simulator does not simulate the operating system.

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 issue slots. Each one is guarded. The pipeline depth varies from 7 to 12 stages. Address and data words are 32 bits wide. It features a unified register file with 128 32-bit registers. The SIMD capabilities are 2 x 16-bit and 4 x 8-bit. The 64Kbyte 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 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 protocol. Each core has its own L1 data cache and can copy data from other L1 caches through 4 channels. 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 one 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. This allows low task execution overhead of less than 2% of the average MB decoding time for task request.

The experiments focus on the baseline profile of the H.264 standard. The baseline 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. In this profile, 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, maximally 16 reference frames, weighted prediction, hexagonal motion estimation algorithm with maximum search range 24, and one slice per frame. The experiments use all four videos from the HD-VideoBench [10], Blue_Sky, Rush_Hour, Pedestrian, and Riverbed, in the three available resolutions, SD, HD and FHD.

The 3D-Wave technique focuses on the thread-level parallelism 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.

4 Implementation

In our previous work we used the FFmpeg decoder, but since we are using the Trimedia simulator for this implementation, we use the NXP H.264 decoder. The 2D-Wave parallelization strategy has already been implemented in this decoder [13], 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. However, 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 Decoder

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 inverse transform 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. Figure 1 shows that 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 on which it depends. When a MB is finished, 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.

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 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. Figure 3 depicts pseudo-code for MB decoding after the tail submit optimization has been performed.

```
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. 3. Tail submit

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.

In the 3D-Wave implementation multiple frames are decoded concurrently which requires modifications to the Reference Frame Buffer (RFB). The RFB stores the decoded frames that are going to be used as reference. As it can service only one frame in flight, the 3D-Wave would require multiple RFBs. Instead, in this proof of concept implementation, the RFB was modified such that a single instance can service 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.

Figure 4 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.

```
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_start TO RMB_List.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 KoL.start TO KoL.last tp_submit(MB);
    //TAIL_SUBMIT
}
```

Fig. 4. Pseudo-code for 3D-Wave

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 8x8 pixel areas and each one of them can be partitioned in up to four 4x4 pixel blocks The 4x4 blocks in an 8x8 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 Figure 1) is added to the list.

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 Figure 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) which contains the parameters of the MBs subscribed to this RMB. When an 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 Figure 3.

4.3 Frame Scheduling Policies

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 Task Priorities

In video decoding, latency is an important characteristic of the system. The frame scheduling policy described in the previous section reduces the frame latency. However, as 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 decoding latency.

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 the submission to the TP is made using the 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, the impact on the memory and bandwidth requirements, and the results of the frame scheduling and priority policies. The experiments were carried out according to the methodology described in Section 3. To evaluate the 3D-Wave technique, one second (25 frames) of each sequence was decoded using the enhanced NXP decoder. Due to long simulation times and the large number of simulations, more frames could not be simulated. The four sequences of the HD-VideoBench using three resolutions were evaluated. Due to space limitations only the results for the Rush_Hour sequence are presented which are close to the average. The results for the other sequences vary less than 5%.

5.1 Scalability

The scalability results are for 1 to 64 cores. More cores could not be simulated due to limitations of the simulator. Table 1 depicts the scalability results, i.e., the speedup of the parallel implementation running on p processors over the parallel implementation running on a single core, of the 2D-Wave (columns labeled 2D-W) and 3D-Wave (columns labeled 3D-W) implementations. In addition, it shows the speedup of the 3D-Wave running on p cores over the 2D-Wave on p cores (columns labeled 3D vs 2D). On a single core, 2D-Wave can decode 39 SD, 18 HD, and 8 FHD frames per second, respectively.

Table 1. 2D-Wave and 3D-Wave speedups for the 25-frame Rush Hour sequences

Cores	SD			HD			FHD		
	2D-W	3D-W	3D vs 2D	2D-W	3D-W	3D vs 2D	2D-W	3D-W	3D vs 2D
1	1.00	1.00	0.92	1.00	1.00	0.92	1.00	1.00	0.92
2	1.77	2.00	1.05	1.77	2.00	1.04	1.78	2.00	1.04
4	3.22	4.00	1.14	3.27	3.99	1.13	3.31	4.00	1.11
8	5.56	7.80	1.29	5.78	7.83	1.25	5.96	7.88	1.22
16	8.19	14.63	1.65	9.31	14.75	1.46	9.92	15.21	1.41
32	8.42	27.78	3.04	11.78	28.44	2.22	14.40	28.94	1.85
64	8.32	49.32	5.47	11.53	53.16	4.25	15.35	54.78	3.28

On a single core the 3D-Wave implementation takes 8% more time than the 2D-Wave implementation due to administrative overhead. The 3D-Wave implementation scales almost perfectly up to 8 cores, while the 2D-Wave implementation incurs a 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 it 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.

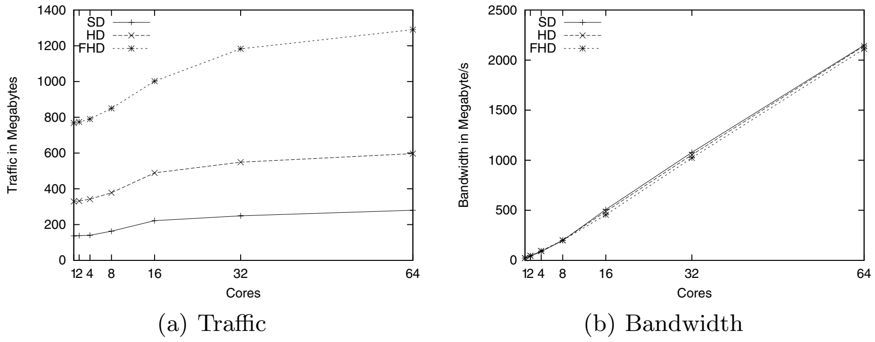
For SD sequences the 2D-Wave technique saturates at a speedup of just over 8 for 16 cores and beyond. This happens because of the limited amount of MB parallelism inside the frame and the dominant ramp up and ramp down of the availability of parallel MBs. The 3D-Wave technique for the same resolution continuously scales up to 64 cores with a parallelization efficiency just below 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 implementation in Table 1 just slightly increase for higher resolutions. The 2D-Wave implementation on the other hand, 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, for large number of cores cache trashing occurs, as will be shown in the next section, which results in a large number of memory stalls. 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 wasted during these two periods. In a real sequence with many frames it would be negligible.

5.2 Bandwidth Requirements

The impact of the 3D-Wave technique on memory and communication bandwidth has also been analyzed. First the data traffic between the L1 and L2 data caches is measured. Figure 5(a) depicts the traffic for the three resolutions. The graph shows that 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. This increase in traffic is the result of cache thrashing. 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



(a) Traffic (b) Bandwidth

Fig. 5. Traffic and bandwidth for FHD Rush_Hour sequence

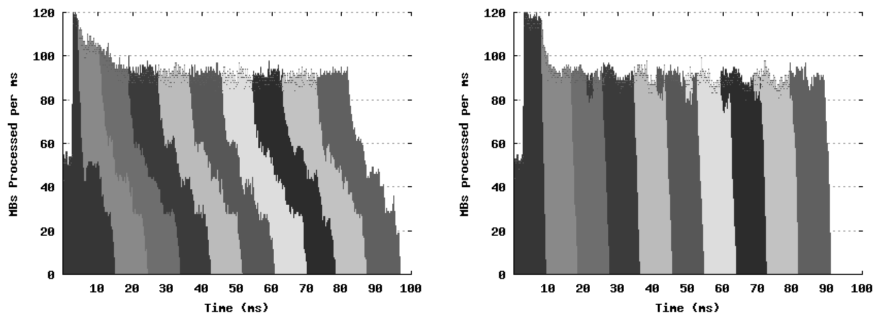
submit strategy). However, because 3D-Wave exploits inter-frame data locality, it results in an average 18% less traffic than 2D-Wave.

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. Figure 5(b) depicts the bandwidth results for different numbers of cores.

The total amount of intra chip bandwidth required for 64 cores is 2.1 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. This represents a workload more than 16 times higher than required for real time decoding, but it indicates what can be necessary for future applications such as 3D TV.

5.3 Frame Scheduling

Figure 6(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 3 frames in flight while there is a small performance loss of about 5%. This performance loss can be explained by the short sequence used. In these short sequences the time of ramp up and ramp down has a non-negligible impact on the overall performance.



(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. 6. Results for frame scheduling and priority policy for FHD Rush_Hour in a 16-core processor. Different colors represent different frames.

In the current state of development, 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.

5.4 Priority Policy

The priority mechanism, presented in Section 4.4, strongly reduces the latency of the frame to be decoded. In the original 3D-Wave implementation the latency of the first frame 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 almost the same as the latency of the 2D-Wave, which decodes frames one-by-one. Figure 6(b) depicts the number of MBs processed per *ms* when this feature is used.

6 Conclusions

In this work an implementation of the 3D-Wave parallelization technique on an embedded CMP has been presented and evaluated. The implementation requires to identify intra-frame MB dependencies dynamically, which led to the development of a mechanism where MBs subscribe themselves to the MBs in the reference areas they depend upon. We have also presented policies for reducing the number of frames in flight and the frame latency.

The results show that the 3D-Wave implementation can leverage a multicore system with up to 64 cores. While the 2D-Wave has a speedup about 8 for 16 cores or more, for SD resolution, the 3D-Wave has a speedup of almost 45 on 64 cores. These results were achieved for sequences with no frame-, slice- or GOP-level parallelism.

Future work includes the development of an automatic frame scheduling technique, the implementation of the 3D-Wave on general purpose processors, and the implementation of the 3D-Wave in the encoder. A 3D-Wave implementation of the encoder can be applied for high definition, low latency encoding on multi-processors.

Acknowledgment

This work 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. NAB Broadcast Engineering Conference, 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: IEEE Inter. Conf. on Proc. Image Processing, pp. 2977–2980 (2006)
3. Meenderinck, C., Azevedo, A., Alvarez, M., Juurlink, B., Ramirez, A.: Parallel Scalability of H.264. In: Proc. First Workshop on Programmability Issues for Multi-Core Computers (January 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., Li, E., 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. Oelbaum, T., Baroncini, V., Tan, T., Fenimore, C.: Subjective Quality Assessment of the Emerging AVC/H.264 Video Coding Standard. In: Proc. Inter. Broadcast Conference (IBC) (2004)
8. 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 Symposium, pp. 24–33 (2005)
9. 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)
10. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In: IEEE Int. Symp. on Workload Characterization (2007)
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: Proc. 38th Inter. Symp. on Microarchitecture (MICRO), pp. 331–342 (2005)
12. X264. A Free H.264/AVC Encoder, `http://developers.videolan.org/x264.html`
13. Hoogerbrugge, J., Terechko, A.: A Multithreaded Multicore System for Embedded Media Processing. Transactions on High-Performance Embedded Architectures and Compilers 4(2) (to Appear, 2009)