Performance Evaluation of Macroblock-level Parallelization of H.264 Decoding on a cc-NUMA Multiprocessor Architecture

Mauricio Alvarez
Universitat Politècnica de Catalunya
Barcelona - Spain
alvarez@ac.upc.edu

Alex Ramírez
Barcelona Supercomputing Center
Barcelona - Spain
alex.ramirez@bsc.es

Mateo Valero
Barcelona Supercomputing Center
Barcelona - Spain
mateo.valero@bsc.es

Arnaldo Azevedo
Delft University of Technology
Delft, The Netherlands
azevedo@ce.et.tudelft.nl

Cor Meenderinck
Delft University of Technology
Delft, The Netherlands
cor@ce.et.tudelft.nl

Ben Juurlink
Delft University of Technology
Delft, The Netherlands
B.H.H.Juurlink@tudelft.nl

ABSTRACT

This paper presents a study of the performance scalability of a macroblock-level parallelization of the H.264 decoder for High Definition (HD) applications on a multiprocessor architecture. We have implemented this parallelization on a cache coherent Non-uniform Memory Access (cc-NUMA) shared memory multiprocessor (SMP) and compared the results with the theoretical expectations. Three different scheduling techniques were analyzed: static, dynamic and dynamic with tail-submit. A dynamic scheduling approach with a tail-submit optimization presents the best performance obtaining a maximum speed-up of 9.5 using 24 processors. A detailed profiling analysis showed that thread synchronization is one of the limiting factors for achieving a better parallel scalability. The paper includes an evaluation of the impact of using blocking synchronization APIs like POSIX threads and POSIX real-time extensions. Results showed that macroblock-level parallelism as a very fine-grain form of Thread-Level Parallelism (TLP) is highly affected by the thread synchronization overhead generated by these APIs. Other synchronization methods, possibly with hardware support, are required in order to make MB-level parallelization more scalable.

1. INTRODUCTION

Video applications had become a very important workload in multiple computing environments, ranging from mobile media players to Internet servers. In order to deliver the increasing levels of quality and compression efficiency that new multimedia applications are demanding, in the recent years a new generation of video coding standards have been defined [1, 2]. Furthermore, the trend towards high quality video systems has pushed the adoption of High Definition (HD) digital video and even higher definitions are being proposed [3, 4]. The combination of the complexity of new video Codecs and the higher quality of HD systems has resulted in an important increase in the computational requirements of the emerging video applications [5, 6]. Parallelization at different levels has emerged as one of the solutions for providing the required performance.

At the same time there, there is a paradigm shift in computer architecture towards chip multiprocessors (CMPs). In the past performance has improved mainly due to higher clock frequencies and architectural approaches to exploit instruction-level parallelism (ILP). It seems, however, that these sources of performance gains are exhausted. New techniques to exploit more ILP are showing diminishing results while being costly in terms of area, power, and design time. Also clock frequency increase is flattening out, mainly because pipelining has reached its limit. As a result, the increasing number of transistors per chip are now dedicated to include multiple cores per chip. Following the current trend in multicore designs it is expected that the number of cores on a CMP will double every processor generation, resulting in hundreds of cores per die in the near future [7, 8]. A central question is whether applications scale to benefit from the increasing number of cores in the future CMPs.

CMPs are good candidates for speeding-up video codecs, but only in the case that the later can be effectively parallelized. As a result, an important research effort has been made in the last years for developing techniques for parallelization of advanced video codecs like H.264 [9, 10, 11]. One of the most promising techniques is the parallelization at the level of macroblocks (MBs), in which small blocks of the video frame are processed in parallel. This type of parallelization has been presented in theoretical and simulation analysis as scalable and efficient, but there has not been an extensive analysis on real parallel platforms. In this paper we investigate the performance scalability of Macroblock-level parallelization for the H.264 decoder on a cache coherent Non-uniform Memory Access (cc-NUMA) Shared Memory Multiprocessor (SMP). Although cc-NUMA SMPs have a different architecture constraints than CMPs they share common features allowing this study to present useful insights for porting video codec applications to multicore architectures.

The paper is organized as follows. First, in section 2 we present a brief introduction to the H.264 video codec standard and we discuss the parallelization strategy that we have used. Then, in section 3 we present the methodology used for this study including the benchmarks and computing platform. In section 4 we present results obtained in the SMP machine. An finally, in section 6 we present the conclusions and future work.

2. PARALLELIZATION OF H.264

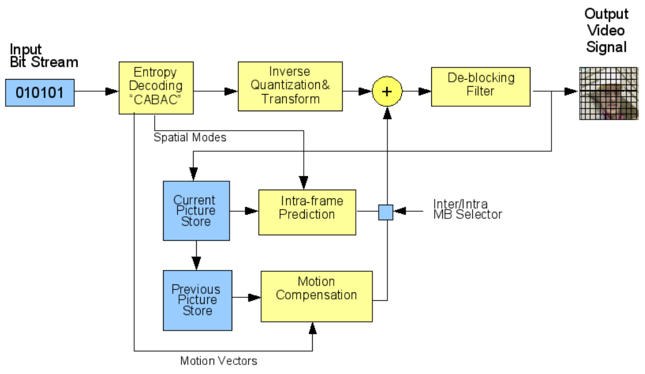


Figure 1: Block diagram of the decoding process.

Currently, H.264 is one of the the best video coding standards in terms of compression and quality. It is used in HD-DVD and blu-ray Disc, and many countries are using/will use it for terrestrial television broadcast, satellite broadcast, and mobile television services. It has a compression improvement of over two times compared to previous standards such as MPEG-4 ASP and MPEG-2 [1]. Figure 1 depicts a block diagram of the decoding process of H.264. The main kernels are Prediction (intra prediction or motion estimation), Discrete Cosine Transform (DCT), Quantization, Deblocking filter, and Entropy Coding. These kernels operate on macroblocks (MBs), which are blocks of 16×16 pixels, although the standard allows some kernels to operate on smaller blocks, down to 4×4. H.264 uses the YCbCr color space with mainly a 4:2:0 subsampling scheme. More details about H.264 video standard can be found in [12, 13].

H.264/AVC is based on the same block-based motion compensation and transform-based coding framework of prior MPEG video coding standards. It provides higher coding efficiency through added features and functionality that in turn entail additional complexity. In order to provide the required performance for HD applications it is necessary to exploit Thread Level Parallelism (TLP)

Macroblock(MB)-level parallelization has been proposed as a scalable and effective technique for parallelizing the H.264 decoder [9, 14, 11, 15, 16]. It can scale to a large number of processors without depending on coding options of the input videos, and without affecting significantly the latency of the decoding process.

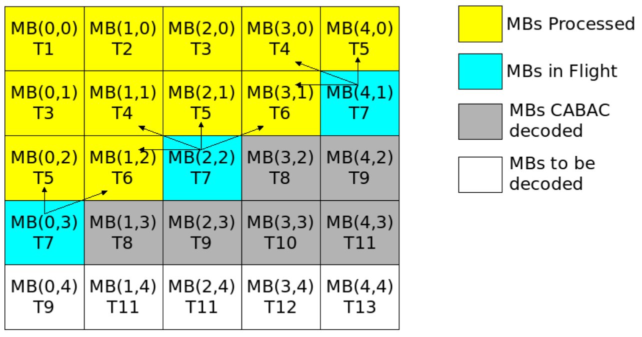


Figure 2: MB-level Parallelism Inside a Frame

In H.264 usually MBs in a frame are processed in scan order, which means starting from the top left corner of the frame and moving to the right, row after row. To exploit parallelism between MBs inside a frame it is necessary to take into account the dependencies between them. In H.264, motion vector prediction, intra prediction, and the deblocking filter use data from neighboring MBs defining a structured set of dependencies. These dependencies are shown in Figure 2. MBs can be processed out of scan order provided these dependencies are satisfied. Processing MBs in a diagonal wavefront manner satisfies all the dependencies and at the same time allows to exploit parallelism between MBs. We refer to this parallelization technique as 2D-Wave [9].

It is important to note that, due to the sequential behavior of the entropy decoding kernel it should be decoupled from the rest of macroblock decoding. In our implementation, first CABAC entropy decoding is performed for all the MBs in a frame and the results are stored in an intermediate buffer. Once entropy decoding is performed, the decoding of MBs is executed in parallel using the 2D-wave strategy.

2.1 Theoretical Maximum Speed-up

Assuming that the time to process each MB is constant, that there is not overhead for thread synchronization and that there is an unlimited number of processors then we can estimate the theoretical maximum performance from the 2D-wave MB parallelization. The maximum speed-up can be calculated as follows: Let *mb_width* and *mb_height* be the width and height of the frame in macroblocks respectively. Then, the time for processing all the macroblocks in sequential and parallel way are T_{seq} and T_{par} respectively:

$$T_{seq} = mb\_width * mb\_height$$

$$T_{par} = mb\_width + (mb\_height - 1) * 2)$$

Taken that into account, the maximum theoretical speed-up and the maximum number of processors can be calculated as follows:

$$Max\_speed\_up = \frac{T\_seq}{T\_par}$$

$$Max\_proc = round((mb\_width + 1)/2)$$

In table 1 the values of T_{seq} and T_{par} for different frame sizes are shown. As can be seen, the parallelization scales with the

Resolution	hz pix	ver pix	mb_width	mb_height	T_{seq}	T_{par}	max-speed-up	max-proc.
QCIF	176	144	11	9	99	27	3.67	6
CIF	352	288	22	18	396	56	7.07	11
STD	720	576	45	36	1620	115	14.09	23
HD	1280	720	80	45	3600	168	21.43	40
FHD	1920	1088	120	68	8160	254	32.13	60

Table 1: Computational work, Critical Path and Maximum Speed up of Parallelization for Different Resolutions

resolution of the image. For Full High Definition (FHD) resolution the theoretical maximum speed-up is 32.13X when using 60 processors.

3. EVALUATION PLATFORM

Throughout this paper we use the HD-VideoBench [17], a benchmark for testing HD video coding applications. The bechmark includes four video test sequences with different motion and spatial characteristics. All movies are available in three formats: 720×576 (SD), 1280×720 (HD), 1920×1088 (FHD). Each movie has a frame rate of 25 frames per second and has a length of 100 frames.

H264 encoding was done with the X264 encoder using the following options: 2 B-frames between I and P frames, 16 reference frames, weighted prediction, hexagonal motion estimation algorithm (hex) with maximum search range 24, one slice per frame, and adaptive block size transform. Movies encoded with this set of options represent the typical case for HD content. The H.264 decoder is a modified version of FFmpeg [18] for supporting 2D-wave parallelization.

3.1 Architecture

The application was tested on a SGI Altix which is a distributed shared memory (DSM) machine with a cc-NUMA architecture. The basic building block is this DSM system is the blade. Each blade has two dual-core Intel Itanium processors, 8GB of RAM and an interconnection module. The interconnection of blades is done using a high performance interconnect fabric called NUMAlink-4 capable of 6.4 GB/s peak bandwidth through two 3.2 GB/s unidirectional links (see figure 3.1). The complete machine has 32 nodes with 2 dual-core processors per node for a total of 128 cores and a total of 1TB of RAM.

Each processor in the system is a Dual-Core Intel Itanium2 processor running at 1.6 GHz. This processors has a 6-wide, 8-stage deep, in order pipeline. The resources consist of six integer units, six multimedia units, two load and two store units, three branch units, two extended-precision floating point units, and one additional single-precision floating point unit per core. The hardware employs dynamic prefetch, branch prediction, a register scoreboard, and non-blocking caches. All the three levels of cache are located on-die. The L3 cache is accessed at core speed, providing up to 8.53 GB/s of data bandwidth. The Front Side Bus (FSB) runs at a frequency of 533 MHz.

The compiler used was gcc 4.1.0 and the operating system was Linux with kernel version 2.6.16.27. Profiling information was taken using the Paraver tool with traces generated using the source-to-source Mercurium compiler and the Mintaka trace generator [19].

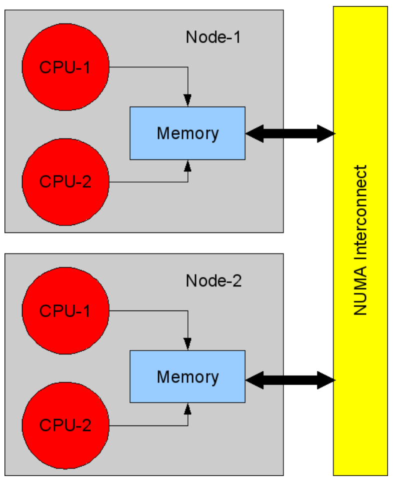


Figure 3: Architecture of the cc-NUMA Multiprocessor

The application was executed on the SMP machine using "cpusets" and "dplace" options. Cpusets are objects in the linux kernel that enable to partition the multiprocessor machine by creating separated execution areas. By using them the application has exclusive use of all the processors. With the dplace tool, memory allocation is done taking into account the cc-NUMA architecture and data is allocated in a NUMA friendly way.

3.2 Programming Model

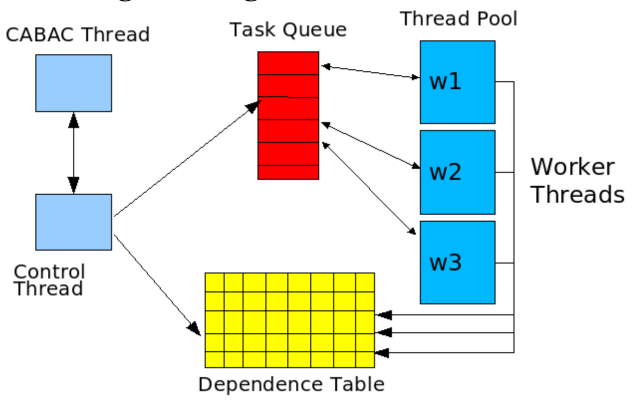


Figure 4: Dynamic Task Model Diagram

Our implementation is based on a dynamic task model in which a set of threads is activated when a parallel region is encountered. In the case of the 2D-wave parallelization a parallel region is the decoding of all MBs in a frame. Each parallel region is controlled by a frame manager, which consist of a thread pool, a task queue, a dependence table and a control thread as showed in Figure 3.2.

The thread pool consists of a group of worker threads that wait for work on the task queue. The generation of work on the task queue is dynamic and asynchronous. The dependencies of each MB are expressed in a dependence table. When all the dependencies for a MB are resolved a new task is inserted on the task queue. Any thread from the thread pool can take this task and process it. When a thread finish the processing of a MB it updates the table of dependencies and if it founds that another MB has resolved its dependencies it can insert a new task into the task queue. Finally, the control thread is responsible for handling all the initialization and finalization tasks that are not parallelizable.

As a further optimization step a tail-submit method has been implemented in which each worker thread can process MBs directly without passing through the task queue. With the tail-submit optimization a thread that has decoded a MB checks whether the dependencies of its neighboring right and down-left MBs has been resolved. If at least one of these neighbor MBs is ready to be processed the current worker thread continues decoding that MB directly. If both of them are ready, the worker thread submit one to the task queue and process the other directly. If there are not MBs ready, the worker thread access the task queue to wait for a new MB to be decoded. This technique has the advantage of dynamic distributed control, low thread management overhead, and also exploits spatial locality, reducing the data communication overhead, as we will show later.

The dynamic task model was implemented using POSIX threads. Synchronization between threads and access to the task queue was implemented using blocking synchronization with POSIX real-time semaphores. Atomicity of the read and write operations is guaranteed by using mutexes. Both synchronization objects are blocking, which means that the operating system (OS) is responsible for the activation of threads. The access to the table of dependencies was implemented with atomic instructions. An atomic operation *dec_and_fetch* is used for decrementing the counter of dependencies of a particular MB and detecting if that count has reached the zero value.

4. PERFORMANCE ANALYSIS

One of the main factors that affects the scalability of the 2D-wave parallelization is the allocation (or scheduling) of MBs to processors. In this paper we have evaluated three different scheduling algorithms. The first one is a static scheduling which assumes constant processing time. Second, a dynamic scheduling mechanism based on the task queue model in which the processing order is defined dynamically. And finally, dynamic scheduling was enhanced with tail submit optimization in which processors can decode MBs directly without passing through the task queue.

The application was executed for the three scheduling algorithms with 1 to 32 processors. Although the machine has 128 processors, executions with more than 32 processors were not carried out because the speed-up limit is always reached before that point. Speed-up is calculated against the original sequential version. The speed-up reported (unless the contrary is explicitly said) is for the parallel part which corresponds to the decoding of all MBs in a frame without considering entropy decoding. In all the speed-up results the value for one thread is the normalized execution time for the sequential version. The other points represent the execution time for the parallel version. N-thread number means 1 control thread and N-1 decoder threads.

In figure 4 the average speed-up for the different scheduling approaches is presented. In the next sections each one is going to be discussed in detail.

4.1 Static scheduling

Static scheduling means that the decoding order of MBs is fixed and a master thread is responsible for sending MBs to the decoder threads following that order. The predefined order is a zigzag scan order which can lead to an optimal schedule if MB processing time is constant. When the dependencies of an MB are not ready the master thread waits for them.

Figure 4 shows the speed-up of static scheduling. The maximum available speed-up reached is 2.51 when using 8 processors (efficiency of 31%). Adding more processors do not result in more performance, instead there is a big reduction in speed-up. This is due to the fact that MB processing time is variable and the static schedule fails to discover the available parallelism. Most of the time the master thread is waiting for other threads to finish.

4.2 Dynamic scheduling

In this scheme production and consumption of MBs is made through the centralized task queue. The task queue has been implemented using blocking synchronization. This makes the programming easier but also involves the OS in the scheduling process. Since processing MBs is very fine grained, the interaction of the OS causes a significant overhead as we will show later.

In figure 4 the speed-up for the dynamic scheduling version is shown. A maximum speed-up of 2.42 is found when 10 processors are used. This is lower than the maximum speed-up for the static scheduling. Although the dynamic scheduling is able to discover more parallelism, the overhead for submitting MBs to and getting MBs from the task queue is so big that it jeopardizes the parallelization gains. In the next section we are going to analyze in detail the different sources of this overhead.

4.2.1 Execution phases

As mentioned, the parallel system consists of a combination of a master thread and a group of worker threads. In order to analyze the performance of each worker thread we divided the execution of each one into seven phases. In Figure 6 the pseudo-code of each worker thread is presented illustrating the different phases. The resulting phases are:

- *wait_start_signal*: Synchronization point at the beginning of the frame.

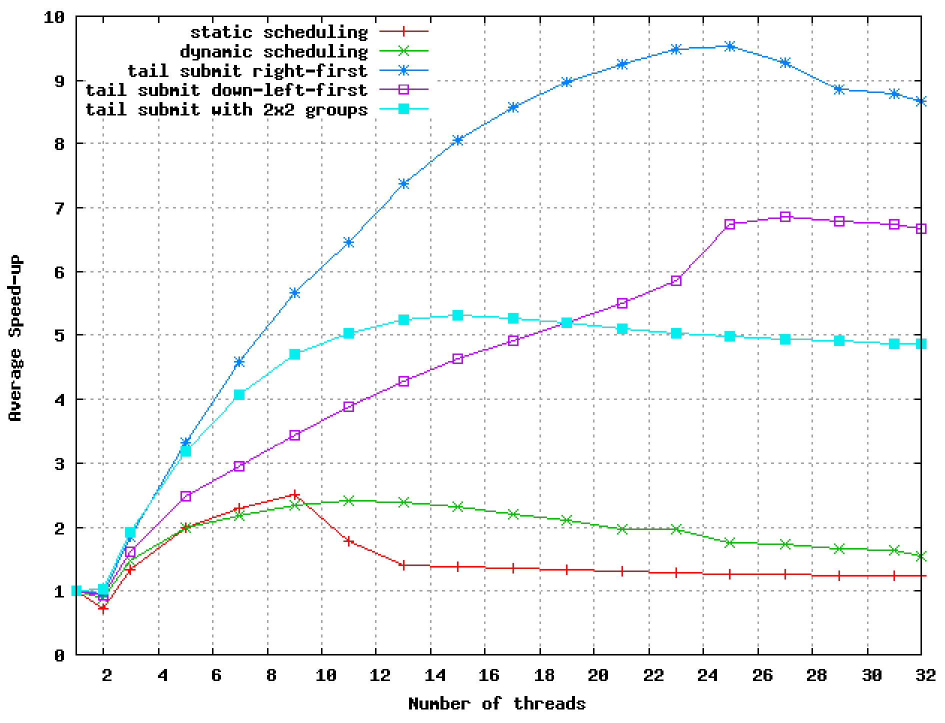


Figure 5: Speed-up of Macroblock Decoding using Different Scheduling Approaches

```
decode_mb_worker:
    FOR-EACH frame-in-sequence:
        wait_start_signal()
        WHILE no-end-of-frame
            get_mb()
            copy_mb()
            decode_mb()
            update_dep()
            ready_mb()
            submit_mb()
```

Figure 6: Pseudo-code of 2D-wave with Dynamic Scheduling

- *get_mb*: Take one MB from the task queue, if there are no MBs available remains blocked until another thread puts one MB.
- *copy_mb*: Copy of CABAC values from the intermediate buffer to the local thread structures.
- *decode_mb*: Actual work of macroblock decoding.
- *update_dep*: Update the table of MB dependencies.
- *ready_mb*: Analysis of each new ready to process MB.
- *submit_mb*: Put one MB into the task queue. If the task queue is full the thread remains blocked until another thread removes one MB.

In Table 2 the execution time of the different phases is presented in absolute time (us). The first conclusion that can be extracted is that the decoding time for each MB increases with the number of processors. When executing with 16 processors the time spent in actual decoding increases by a factor of 2.72 compared to the single thread case. This is mainly due to the cc-NUMA architecture of the machine. The dynamic scheduling algorithm do not consider data locality when assigning tasks to processors. It might be the case that a processor take a MB which has its data dependencies in another processor in a remote node, then all the memory access should cross the NUMA interconnection network and pay a high latency for that. As more processors are involved in the parallel decoding more dispersed is going to be the data. A similar behaviour occurs for the *copy_mb* phase. The source area of these copies is the CABAC intermediate buffer that resides in the main memory which, in turn, is distributed across the nodes. The copy from a remote node implies a costly off-chip access.

The phases that exhibit a major increase in execution time are *get_mb* and *submit_mb*. For 32 processors they have a slowdown of 42.82 and 39.36 respectively. This enormous increase in the execution time reveals a contention problem. In dynamic scheduling all the worker threads get MBs from (and submit MBs to) the task queue creating an important pressure on it. This can be illustrated with the time required to submit one MB to the task queue. Since the size of the queue is big enough to put all the MBs in a frame there are not threads that have to wait for an empty slot. Because of that, all the time spent in *submit_mb* is overhead and contention. Submitting one element to the queue when there is only one worker thread takes 7 us, but when there are 32 threads it takes 40 times more.

Threads	decode_mb	copy_mb	get_mb	update_mb	ready_mb	submit_mb	overhead-ratio
1t	22.65	1.62	4.89	1.01	2.38	5.03	0.67
4t	33.09	2.95	9.71	1.36	2.86	12.44	1.30
8t	41.88	3.90	16.67	1.61	3.02	20.84	2.05
16t	61.78	5.94	55.95	2.25	3.55	80.28	6.57
32t	78.75	7.25	209.37	2.70	4.36	201.01	18.88

Table 2: Average Execution Time for Worker Threads with Dynamic Scheduling (time in us)

The last column in the table shows the ratio of actual computation and overhead. For less than 4 processors the overhead is in the range of the average MB processing time, but there is an enormous increase in the overhead with the number of processors. For more than 24 processors the overhead is more than 10 times the processing time. From this we can conclude that the centralized task queue which allows only a single reader and a single writer becomes the bottleneck. A more distributed algorithm like tail submit or work stealing could help to reduce this contention.

4.3 Dynamic scheduling with Tail submit

As a way to reduce the contention on the task queue, the dynamic scheduling approach was enhanced with a tail submit optimization [20]. With this optimization not all MBs have to pass across the queue. When a thread found a ready to process MB it process that MB directly without any further synchronization. There are two ordering options for doing the tail submit process as shown in figure 4.3. In the first one, the MB that is executed directly is the right neighbor of the current MB; in the second one, the selected block is the down-left MB.

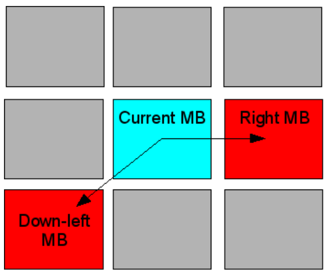


Figure 7: Right-first and Down-left-first orders for doing Tail-submit

In Figure 4 the speed-up of tail-submit implementations is presented. In the down-left-first version a maximum speed-up of 6.85 is achieved with 26 processors (efficiency of 26%). From this point the speed-up is saturated and decreases a bit. In the same figure, it is shown the speed-up of the tail-submit version using the right-first order, a maximum speed-up of 9.53 is achieved with 24 processors (efficiency of 39.7%). Going beyond 24 processors results in diminishing benefits. These results show that the tail-submit version with right-first MB order is more scalable than with the down-left MB order. This is due to the fact the right-first order exploits better the data locality between MBs. Data from the left block is required by the deblocking filter stage and by using the right-first order the values of the previous MB remain in the cache.

4.3.1 Execution Phases

```
decode_mb_worker:
    FOR-EACH frame-in-sequence:
        wait_start_signal()
        WHILE no-end-of-frame
            get_mb()
            DO:
               copy_mb()
               decode_mb()
               update_dep()
               ready_mb()
               submit_mb()
            WHILE ready_mbs
```

Figure 8: Pseudo-code of 2D-wave with Dynamic Scheduling and Tail Submit

In Figure 8 the pseudo-code of the dynamic scheduling with tail submit version is shown. The main difference with the code of dynamic scheduling (see Figure 6) is that the decoding of MBs do not require always a *get_mb* operation.

In Table 3 the profiling results for tail submit version are shown. Different from the case of static scheduling, in tail submit MB decoding time remains almost constant with the number of threads. Decoding one MB takes 21.7 us in average and each thread has a slow-down of 1.39 when executed with 32 threads. The same applies for the *copy_mb* operation. Because the tail submit with right-first order exploits the data locality of neighbor MBs the decode and copy operations are not affected by data dispersion as the dynamic scheduling version.

On of the most important effects of the tail submit optimization is the reduction in the time spent in *submit_mb*. This time increases with the number processors but the absolute value is less than the dynamic scheduling version. The version with 32 threads spend 3.3 times more than the single thread version, but compared the dynamic scheduling it takes 8 times less. With tail submit there is less contention because with there are less submissions to the task queue. The last column in table 3 shows the percentage of MBs that are processed via tail submit, that mean, the MBs that do not pass through the task queue. For one thread, only at the end of the row the decoder thread goes to the queue and ask for a new MB resulting in a 90% of the MBs are processed by tail submit. When more than one thread comes into play then the number of tail submits reduces. That's because threads do not wait for other to finish, instead if there is not direct work to do they go to the task queue and wait there for new MBs. As a result, when there are 32 processors only 48% of the MBs are processed by tail submit, and at this point more MBs are processed through the queue than by tail submit. This is one of the main reasons for not getting more speed-up.

Threads	decode_mb	copy_mb	get_mb	update_dep	ready_mb	submit_mb	overhead-ratio	% of tail submit
1t	21.7	1.5	6.1	1.0	1.0	7.5	0.17	90.8
4t	24.2	1.9	55.9	1.1	1.1	7.8	0.22	79.8
8t	24.9	2.1	132.4	1.3	1.1	8.6	0.30	75.2
16t	27.5	2.4	265.3	1.6	1.1	10.1	0.68	58.5
32t	30.1	2.8	853.1	2.1	1.1	24.8	1.85	48.4

Table 3: Average Execution Time for Worker Threads in Dynamic Scheduling with Tail-Submit (time in us)

```
get_mb_from_taskq(int mb_xy):
    sem_wait(full_sem);
    mutex_lock(read_mutex);
    mb_xy = read_queue(task_queue);
    mutex_unlock(read_mutex);
    sem_post(empty_sem);
    return mb_xy
```

Figure 9: Pseudo-code of get_mb function for obtaining MBs from the task queue

The most significant contributor to the execution time is *get_mb* which represent the time waiting for and getting MBs to appear in the task queue. Time waiting on the task queue is almost 140 times bigger on 32 processors than on a single thread. For 32 threads this function represents 40.7% percent of the total thread time. In this case, the increase in the waiting time is due to the lack of parallel MBs because threads only go to the task queue when they do not found MBs ready to be processed. That means that with 24 processors the scalability limit of the tail submit version has been found.

4.3.2 Thread synchronization overhead

The previous analysis have shown that operations of getting and submitting MBs to/from the task queue have a significant overhead on the final performance and scalability of the parallelization. In order to understand the sources of this overhead we have performed a more detailed profiling analysis. The code in Figure 9 shows the implementation of the *get_mb* function which takes one MB from the task queue. Our implementation is based on two semaphores (using POSIX real-time semaphores): one for available elements in the queue (full_sem) and the other for empty slots in the queue (empty_sem). Reading from the queue requires a critical section which has been implemented using mutexes (from POSIX threads). The implementation for the *submit_mb* function is very similar to this one with the only difference that there is a write operation instead of the read and the semaphores are used in an inverse order.

In Figure 10 the average execution time of the *get_mb* and *submit_mb* functions is presented. The most important part for the first one is related to the semaphore at the beginning (*sem_wait*) which represents the waiting time for elements in the task queue. As the number of processors increases this value becomes bigger. From 1 to 2 threads the execution time increase by a factor of 16.35. For 32 processors this time is 400X bigger than for 1 thread. In the case of a large number of processors this big waiting time indicates that there are no more MBs to process and that the parallelism of the application has reached its limit. But for a small number of processors when there are independent MBs most of the time spent in *sem_wait* is overhead due to the intervention of the OS. A similar behavior can be noted with the increase in the execution time of the release operation in the output semaphore. This operation signals the availability of a new element in the task queue, when the number of thread increases the number of waiters in the task queue increases as well. It seems that the time required to signal the availability of one MB depends on the number of waiters, which is not a desirable feature. What should be appropriated is to take one of the waiter threads (the first one in a queue, for example) and activate it independent of the number of other waiters in the queue. This is an implementation issue of the *sem_post* API.

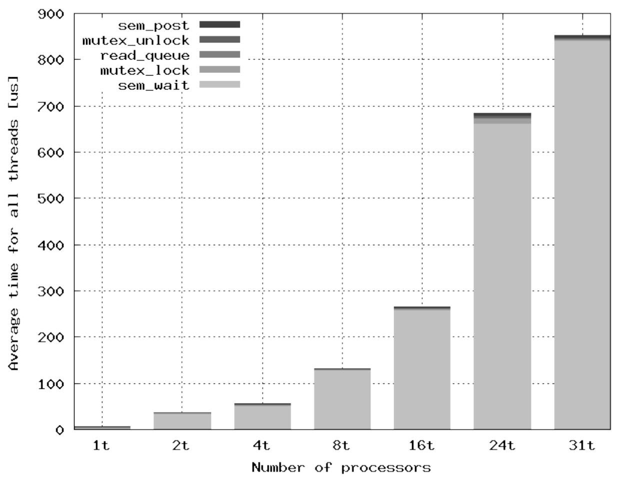
5. RELATED WORK

Several works deal with problem of parallelization of the H.264 video Codec. The H.264 codec can be parallelized either by a task-level or data-level decomposition. In task-level decomposition individual tasks of the H.264 Codec are assigned to processors while in data-level decomposition different portions of data are assigned to processors running the same program.

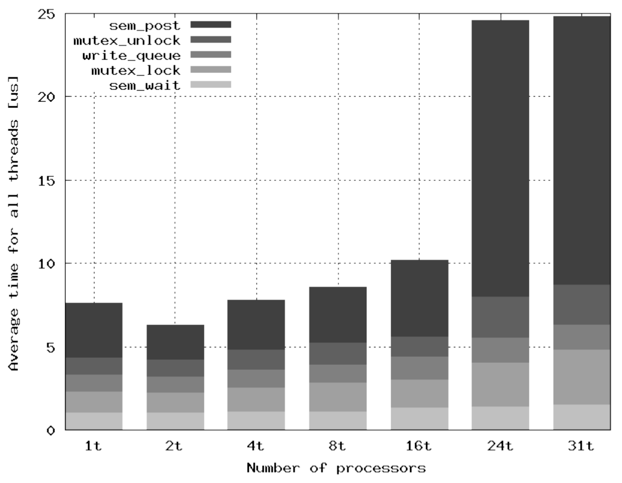
Some works have presented results for task-level parallelization [21, 22]. Task-level parallelism and macroblock pipelining works well for a reduced number of processors (less than 4 typically) but they do not scale well for CMP architectures because it is very complicated to further divide the mapping of tasks for using more processors. As the resolution of the image increase this approach is not enough for providing the required performance for real-time operation.

In the case of data-level decomposition there are different strategies depending on the granularity of the data division. Some works focus on coarse grained parallelization like Group-of-frames (GOP), frame and slice-level parallelism and combinations between them [23, 10, 24, 25]. These coarse-grain parallelization techniques are not scalable because dependencies between frames, or are not appropriate for CMPs due to to false sharing in the caches. Moreover all the proposals based on slice-level parallelism have the problem of the inherent loss of coding efficiency due to having a large number of slices.

MB-level parallelism has been proposed in several works. Van der Tol et al. [9] proposed the technique but they did not present an evaluation of it on a real or simulated platform. Chen et al. [14] evaluated an implementation on Pentium machines with a reduced number of processors. They don't present scalability results and work with lower resolutions. A combination of temporal and spatial MB-level parallelism for H.264 encoding has been discussed by Zhao et al. [15]. A limitation of this approach is that a static row scheduler is used resulting in poor load balancing. Chong et al [16] have proposed to add a prepass stage to the de-



(a) tail submit get mb



(b) tail submit submit mb

Figure 10: Distribution of Execution Time for the get_mb and submit_mb functions in Dynamic Scheduling with Tail submit

coder. Using the information from the preparsing pass a dynamic schedule of the MBs of the frame is calculated and MBs are assigned to processors dynamically. Although they present a dynamic scheduling algorithm it seems to be not able of discovering all the MB level parallelism that is available in a frame. Hoogerbrugge et al [20] have evaluated the 2D-wave parallelization for H.264 decoding using a simulated multicore. The architecture is composed of VLIW media processors with a cache coherent memory organization. They evaluated the original 2D-wave and proposed the tail submit optimization. The main point of this paper is on comparing different schemes for multithreading rather than evaluating the scalability of the H.264 decoding. Additionally the simulated system does not include a detailed model of the memory subsystem hiding the effect of accesses to the main memory. Moreover they have restricted the coding tools of the H.264 encoder in order to simplify the decoder. Our paper presents an evaluation of a real H.264 decoder (supporting most of the features of the standard) on a real platform. Because of that our results includes all the effects of the memory subsystem in the final performance.

Azevedo et al [26] have evaluated a technique that exploits intra- and inter-frame MB-level parallelism (called 3D-wave). They compared this technique with 2D-wave and showed that the former is more scalable than the later. Their analysis has been made on a simulation platform of embedded media processors that does not include a detailed model of the memory hierarchy. Anyway, those results are complementary to the presented in this paper. The architecture used for their simulations include a sophisticated hardware support for thread synchronization ad scheduling a achieving a performance close to the theoretical maximum. This confirms the results from our paper in which we show the necessity of special support for thread synchronization.

6. CONCLUSIONS

In this paper we have investigated the parallelization of the H.264 decoder at the macroblock level. We have implemented such a parallelization on a multiprocessor architecture.

From our implementation we can conclude that the best scheduling strategy is the combination of dynamic scheduling with tail submit. Dynamic scheduling deals with the problem of variable decoding time by assigning MBs to processors as soon as they have resolved all their dependencies. But dynamic scheduling suffers from a big contention overhead on a centralized task queue and inefficiencies in the thread synchronization API. By using tail submit the number of threads that access the task queue is reduced and most of the MBs are processed directly by the worker threads without requiring any further synchronization. Additionally tail submit with right-first order exploits data locality reducing the external memory pressure. Still there is a considerable overhead from thread synchronization which comes from the very low performance of the POSIX real-time semaphores on the evaluated architecture. The use of alternative synchronization mechanisms like lock-free algorithms, or other hardware supported synchronization mechanisms can help to increase the efficiency of the parallelization.

A possible solution, from the architecture point of view, is to include hardware support designed specifically for thread synchronization and scheduling. By doing that, it is possible to eliminate the overhead of thread synchronization and augment the efficiency of the parallelization. An important question that remains open is what is the required hardware support that can give high performance and, at the same time, can be scalable to a large number of processors and general for a range of applications.

Another important limitation of the parallelization scalability is the memory performance. In a cc-NUMA architecture not predictable accesses, like the ones performed in the motion compensation stage, result in off-chip accesses that have a high latency. In a multicore architecture this could be translated as a high demand of bandwidth for the external memory. This is an open issue and we are working on hardware optimizations strategies for augmenting the locality of the data accesses.

7. ACKNOWLEDGEMENTS

This work has been supported by the European Commission in the context of the SARC project (contract no. 27648), the Spanish Ministry of Education (contract no. TIN2007-60625) and HiPEAC (The European Network of Excellence on High-Performance Embedded Architecture and Compilation).

8. REFERENCES

[1] G. J. Sullivan and T. Wiegand, "Video Compression–From Concepts to the H.264/AVC Standard," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 18–31, Jan 2005.

[2] I. Richardson, *H.264 and MPEG-4. Video Compression for Next-generation Multimedia.* Chichester, England: Wiley, 2004.

[3] T. Sikora, "Trends and Perspectives in Image and Video Coding," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 6–17, Jan 2005.

[4] M. Sugawara, "Super hi-vision — research on a future ultra-hdtv system," European Broadcasting Union, Tech. Rep., 2008.

[5] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi, "Video Coding with H.264/AVC: Tools, Performance, and Complexity," *IEEE Circuits and Systems Magazine*, vol. 4, no. 1, pp. 7–28, Jan 2004.

[6] M. Horowitz, A. Joch, and F. Kossentini, "H.264/AVC Baseline Profile Decoder Complexity Analyis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 704–716, July 2003.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[8] S. Borkar, "Thousand core chips: a technology perspective," in *DAC '07: Proceedings of the 44th annual conference on Design automation.* New York, NY, USA: ACM, 2007, pp. 746–749.

[9] E. B. van der Tol, E. G. T. Jaspers, and R. H. Gelderblom, "Mapping of H.264 Decoding on a Multiprocessor Architecture," in *Proceedings of SPIE*, 2003.

[10] Y.-K. Chen, X. Tian, S. Ge, and M. Girkar, "Towards Efficient Multi-level Threading of H.264 Encoder on Intel Hyper-threading Architectures," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, Apr 2004.

[11] C. Meenderinck, A. Azevedo, M. Alvarez, B. Juurlink, and A. Ramirez, "Parallel Scalability of Video Decoders," *Journal Journal of Signal Processing Systems*, August 2008.

[12] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A.Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, July 2003.

[13] "ISO/IEC 14496-10 and ITU-T Rec H.264, Advanced Video Coding," 2003.

[14] Y. Chen, E. Li, X. Zhou, and S. Ge, "Implementation of H. 264 Encoder and Decoder on Personal Computers," *Journal of Visual Communications and Image Representation*, vol. 17, 2006.

[15] Z. Zhao and P. Liang, "Data partition for wavefront parallelization of H.264 video encoder," in *IEEE International Symposium on Circuits and Systems. ISCAS 2006*, 2006.

[16] J. Chong, N. R. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer, "Efficient parallelization of h.264 decoding with macro block level scheduling," in *2007 IEEE International Conference on Multimedia and Expo*, July 2007, pp. 1874–1877.

[17] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications," in *IEEE Int. Symp. on Workload Characterization*, 2007. [Online]. Available: http://people.ac.upc.edu/alvarez/hdvideobench

[18] "FFmpeg Multimedia System." 2005, http://ffmpeg.mplayerhq.hu/.

[19] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to Visualize and Analyze Parallel Code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, P. Nixon, Ed., mar 1995, pp. 17–31.

[20] J. Hoogerbrugge and A. Terechko, "A Multithreaded Multicore System for Embedded Media Processing," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 3, no. 2, pp. 168–187, June 2008.

[21] A. Gulati and G. Campbell, "Efficient Mapping of the H.264 Encoding Algorithm onto Multiprocessor DSPs," in *Proc. Embedded Processors for Multimedia and Communications II*, vol. 5683, no. 1, March 2005, pp. 94–103.

[22] O. L. Klaus Schoffmann, Markus Fauster and L. Böszörmeny, "An Evaluation of Parallelization Concepts for Baseline-Profile Compliant H.264/AVC Decoders," in *Lecture Notes in Computer Science. Euro-Par 2007 Parallel Processing*, August 2007.

[23] A. Rodriguez, A. Gonzalez, and M. P. Malumbres, "Hierarchical parallelization of an h.264/avc video encoder," in *Proc. Int. Symp. on Parallel Computing in Electrical Engineering*, 2006, pp. 363–368.

[24] T. Jacobs, V. Chouliaras, and D. Mulvaney, "Thread-parallel mpeg-2, mpeg-4 and h.264 video encoders for soc multi-processor architectures," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 1, pp. 269–275, Feb. 2006.

[25] M. Roitzsch, "Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding," in *Work-in-Progress Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

[26] A. Azevedo, C. Meenderinck, B. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, and A. Rammirez, "Parallel H.264 Decoding on an Embedded Multicore Processor," in *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers - HIPEAC*, Jan 2009.