

Task-based Parallel H.264 Video Encoding for Explicit Communication Architectures

Michail Alvanos, George Tzenakis, Dimitrios S. Nikolopoulos[†], and Angelos Bilas[†]

Institute of Computer Science (ICS)

Foundation for Research and Technology - Hellas (FORTH)

100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece

{alvanos,tzenakis,dsn,bilas}@ics.forth.gr

Abstract—Future multi-core processors will necessitate exploitation of fine-grain, architecture-independent parallelism from applications to utilize many cores with relatively small local memories. We use *c264*, an end-to-end H.264 video encoder for the Cell processor based on *x264*, to show that exploiting fine-grain parallelism remains challenging and requires significant advancement in runtime support. Our implementation of *c264* achieves speedup between 4.7× and 8.6× on six synergistic processing elements (SPEs), compared to the serial version running on the power processing element (PPE). We find that the programming effort associated with efficient parallelization of *c264* at fine granularity is highly non-trivial. Hand optimizations may improve performance significantly but are limited eventually by the code restructuring they require. We assess the complexity of exploiting fine-grain parallelism in realistic applications, by identifying optimizations of *c264* and the effort they require.

I. INTRODUCTION

Multi-core processors with many simple cores and small, explicitly managed memories are an important design point for high performance computing architectures [1], [2]. These processors present several challenges to software developers: First, software needs to manage efficiently the memory hierarchy, with little or no involvement from programmers. Second, they require fine-grain task parallelism. Third, exploiting fine-grain parallelism necessitates efficient runtime support.

Video encoders are essential components for real-time video processing on portable devices, such as cell phones, PDAs, and video cameras, as well as on personal computers and servers. H.264 video encoding is a complex, multi-phase process, with high memory bandwidth requirements, challenging to parallelize efficiently both at the algorithmic and system level. H.264 video encoding has been parallelized in the past for shared memory multiprocessors, using coarse-grain, frame-based parallelization strategies [3], [4]. Furthermore, H.264 video encoding has been parallelized on the Cell processor using pipelining [5] and static partitioning [6]. These strategies are either architecture-dependent, therefore not portable, or limit the amount of parallelism that can be extracted. We present *c264*, a fine-grain task-level parallel implementation of H.264 which is architecture-independent, as parallelism is

expressed in terms of native application kernels and problem size, achieves good load balancing, and can utilize many cores.

The goal of this paper is to investigate the performance and programming effort associated with fine-grain parallelization of *c264*, on heterogeneous multi-core processors with explicit memory management. Prior work has extensively analyzed the performance of video *decoding* on Cell [7], [8]. Video encoding is significantly more challenging than video decoding due to higher memory requirements and dynamic behavior. Other prior work on the parallelization of video encoding on Cell has considered only the compute-intensive kernels [5], [6]. By contrast, we consider the entire video encoding path, and present an end-to-end implementation, based on task parallelism.

Our parallel implementation uses a master-worker execution model, which maps efficiently to the Cell Control code for generating tasks, tracking task dependencies, and scheduling runs on the control-efficient PowerPC core (PPE), whereas the computationally-intensive components are offloaded on the compute-efficient synergistic processing elements (SPEs). The small size of the local memories prevents frame-level parallelization. We address this using fine-grain macroblock-level parallelization, which allows efficient management of local memories by privatizing and replicating data structures.

Overall, this paper makes the following contributions:

- We present *c264*, an end-to-end implementation of the H.264 encoder for the Cell processor. We present a thorough, quantitative analysis of the application, along with several optimizations.
- We explore the implications of fine-grain task-based parallelism on multi-core processors with small local memories and explicit communication. We show that the impact of TLB misses, the waste of on-chip memory, and task imbalance have tremendous impact on the performance.
- We compare our fine grain task-based implementation with the coarse grain running on a x86 multi-core processor. We show that performance of a x86 processor of the same technology as the Cell is similar to the processor when the implementations are both well optimized.
- We analyze the programming effort associated with parallelizing *c264* at a fine granularity. We show that the effort for privatization of data structures is significant.

The rest of this paper is organized as follows. Section II presents the necessary background on H.264 encoding and

[†]Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR-71409, Greece.

This work was partially supported by the European Commission through the SARC IP (contract no. SARC-27648), the HIPEAC NoE (contract no. IST-004408 and IST-217068) and the ENCORE (contract no. 248647) projects.

x264 video encoding library. Section III presents our design for *c264*. Section IV presents our methodology and experimental results. Section V presents related work. We discuss future directions and draw our conclusions in Section VI.

II. BACKGROUND

H.264 [9] is a video compression standard, also known as MPEG-4 Advanced Video Coding (AVC). Video encoders take as input a raw, uncompressed video stream and process it frame by frame. For each frame, the encoder identifies differences from one or more previously processed frames, called reference frames. The resulting output is an encoded video stream including the reference frames and the differences required to reconstruct all dependent frames. Each frame in H.264 is divided into non-overlapping macroblocks (MBs) of 16×16 pixels. Within each frame, macroblocks are grouped in slices. The output, encoded stream contains information about frames, slices, and macroblocks.

Most MPEG video encoders consist of three main functional units: a temporal model, a spatial model, and an entropy encoder. Figure 1(a) shows the H.264 video encoder block diagram. The temporal model identifies similarities between macroblocks in a single or multiple neighboring frames using *motion estimation*. Motion estimation determines motion vectors that describe how one macroblock is derived by transforming one or more reference macroblocks. Motion estimation algorithms vary both in the way they select motion vectors as well as the shape of the region they explore in each reference frame. Motion estimation identifies a region in the reference frames that minimizes a matching criterion and marks it as the “best match”. The H.264 standard allows the use of up to 16 reference frames for motion estimation.

The encoder, in addition to the temporal model, also employs intra-frame analysis and selects intra encoding when its bitrate cost is lower than that of inter-frame analysis to improve the overall coding efficiency. Intra-frame analysis tries to locate similarities between the current macroblock and its neighbors within the same frame. The video encoder subtracts the selected, best matching region in the reference frame(s) from the current macroblock to produce a new macroblock (*motion compensation*), which in turn is encoded and transmitted together with a motion vector describing the position of the best matching region.

The spatial model transforms macroblock differences using an integer spatial block transformation based on the discrete cosine transformation and generates a set of coefficients that are then quantized. In addition, the spatial model includes a complete H.264 decoder. If an encoded frame needs to be used as reference for encoding other frames, it is better to use a version of the frame that is derived by decoding the encoded frame, rather than the original raw input frame. This approach leads to better quality video streams. Thus, the encoder, after the spatial model, decodes and stores the reconstructed frame in memory to use it as reference for subsequent frames. A *deblocking filter* is applied to every decoded macroblock to reduce blocking distortion, created from quantization. This filter aims at improving visual quality

and prediction performance by smoothing sharp edges between macroblocks.

Macroblocks encoded using *only* macroblocks of the same frame are called intra coded (I-type) macroblocks, while macroblocks that are encoded using macroblocks of other frames *also* are called either predicted-type (P-type) or bidirectionally-predictive (B-type) macroblocks. P-type frames use temporal redundancy from past I- or P-frames, whereas B-type macroblocks use both past and future reference frames, and consequently achieve the highest degree of compression. Each P- and B-type frame can contain I-type macroblocks.

Finally, an *entropy encoder* combines the quantized coefficients and motion vectors in a single stream and encodes it using either context-based adaptive variable length coding (CAVLC) or context-based adaptive binary arithmetic coding (CABAC). The video encoder encapsulates the output stream in packets called Network Abstraction Layer (NAL) units.

x264 [10] is an open source library for encoding H.264 video streams. *x264* supports matching blockings of different size during motion estimation from 16×16 down to 4×4 . It also supports several optimized motion estimation algorithms, such as diamond (DIA), hexagon (HEX), uneven multi-hexagon (UMH), and exhaustive search. The encoder performs mostly integer operations and has been optimized, using vector instructions, for various architectures. Figure 1(b) shows an execution profile for the serial *x264* using manual code instrumentation, with one reference frame, 128×128 motion estimation search area, UMH search algorithm, and the Power Processing Element (PPE) on Cell. We observe that most of the execution time is spent in the analysis and encoding phases (spatial and temporal model). These modules cover about 70% and 85% of the serial execution time on the PPE for DIA and UMH motion estimation algorithms, respectively. We also observe that the input video resolution does not affect significantly the percentage of execution time spent in analysis and encoding for the same motion estimation algorithm. Entropy encoding, accounts for an additional 5–15% of the total execution time of the encoder. The deblocking filter accounts for 3–5% of total execution time. The remaining execution time on the PPE is for frame initialization, memory copy operations between buffers, and control code.

III. DESIGN AND IMPLEMENTATION OF *c264*

We parallelize *c264* with a task-based programming model and runtime, *Tagged Procedure Calls (TPC)* [11]. The runtime aims at reducing task management overhead by eliminating off-chip memory accesses for task initiation and processing of completion notifications. TPC hides all operations associated with data transfers behind a simple task abstraction with one or more arguments. The runtime system uses only on-chip operations when initiating and completing tasks, although argument data may require off-chip transfers.

The *x264* encoder uses coarse grain, frame-based parallelization, where different threads process different frames. When processing a frame, each thread determines the frame type, calculates rate control, and spawns a thread for this frame. Each thread proceeds to process all macroblocks

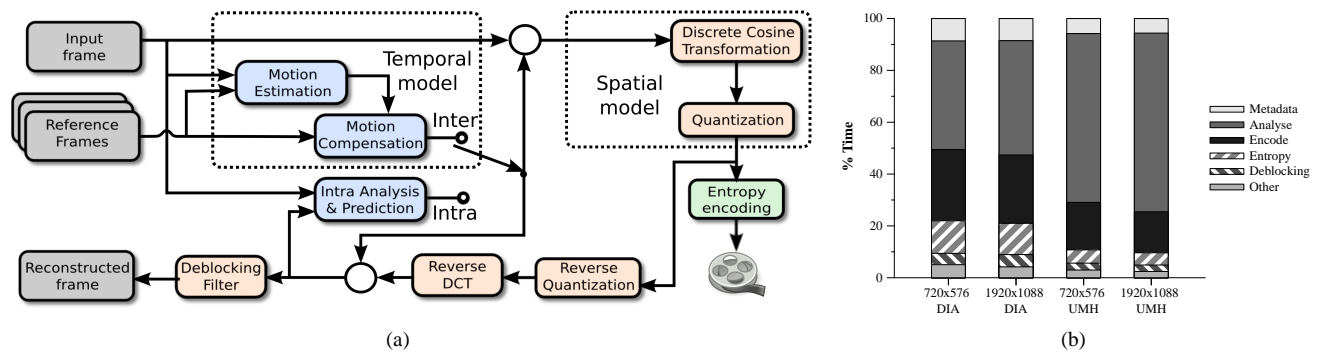


Fig. 1. (a) Block diagram of H.264 video encoding. (b) Normalized execution time breakdown of *x264* for different resolutions and motion estimation algorithms.

through the encoding process, including entropy encoding. Each thread uses locks to wait for the completion of the appropriate line of pixels, when it accesses a not-yet complete part of another frame. The limited size of local stores on Cell prevents frame-level parallelization. An alternative, slice-based parallelization [12] is also not appropriate for the Cell processor due to the limited intra-frame parallelism and the increased memory requirements. A more appropriate, fine grain approach for processors with small local stores is parallelization at the macroblock-level [13]. This approach reduces memory requirements on the SPEs, however it increases communication significantly: the search region is required by all cores when processing a single macroblock, resulting in replication of data in multiple local stores over time.

A. Exploiting parallelism

Exploiting parallelism at the macroblock level requires satisfying the data dependencies between macroblocks. For a given macroblock, the following operations need data from neighboring macroblocks:

- Inter-frame prediction requires motion vectors of neighboring macroblocks to determine the final motion vector.
- Intra-frame prediction uses pixels from neighboring blocks of the current macroblock.
- The deblocking filter, which is applied on the reconstructed frame after the transformation, uses pixel values of neighboring left and upper macroblocks.
- The entropy encoding selects one out of four look-up tables to use for encoding coefficients. The selected table depends on the number of non-zero coefficients in the previously encoded macroblock.

To take advantage of macroblock-level parallelism we create tasks that are related to macroblock processing for the three phases of encoding: analyze and encode, entropy encoding, and deblocking. The simplest way to manage task dependencies is to issue all macroblocks in an antidiagonal-based manner and wait before issuing the next antidiagonal. This technique is also known as 2D-wavefront parallelism [14]. Although motion estimation is independent, *x264* reads the results from neighboring macroblocks and uses them as hints to minimize the overall search time.

The rest of the application code is mostly control code that cannot be parallelized. Parts of this code can run in parallel with other tasks, whereas other parts are dependent on all tasks and need to run between groups of concurrent tasks serially. We offload the first type of code to an SPE as a single task that runs concurrently to other tasks, whereas the second type of code is executed on the PPE.

We issue the entropy encoding task after the completion of the analyze/encode stage of the current macroblock and the entropy encoding stage of the previous macroblock in scan order. Entropy encoding is offloaded as a single task to an SPE but can run concurrently to other tasks of the same frame. Entropy encoding is control-intensive and runs slightly faster on the PPE than the SPE, due to the presence of dynamic branch prediction on the PPE. We execute the entropy encoding on the PPE at the end of each frame, when there are no additional parallel tasks to execute.

The third task for each macroblock applies the deblocking filter. In *c264*, each task applies the deblocking filter in the macroblock line of the reconstructed frame to avoid redundant data transfers. In addition to the macroblock related tasks, we offload concurrent and independent memcopy operations to SPEs. For example, we copy the input raw frame from the read buffer to memory aligned buffers of the encoder. Figure 2 depicts the set of dependencies between macroblocks within a frame. Arrows show dependencies of outstanding macroblocks to previously encoded macroblocks in an antidiagonal manner.

B. Optimizations

a) *Scheduling*: Heuristics in the motion estimation algorithms and non-predictable encoding time may introduce load imbalance. To address the issue we use *dynamic scheduling* instead of *static scheduling*, using per-macroblock state to maintain the dependencies. We issue tasks from any antidiagonal as soon as their individual dependencies are satisfied.

b) *Memory optimizations*: The Cell processor requires attention to several aspects of memory management. We apply a number of optimizations to increase the performance of DMA transfers. First, we use data prefetching through multi-buffering whenever applicable. However, the space available in the local stores for data prefetching in *c264* is limited. The SPE binary file of the application is about 145 KBytes, which leaves

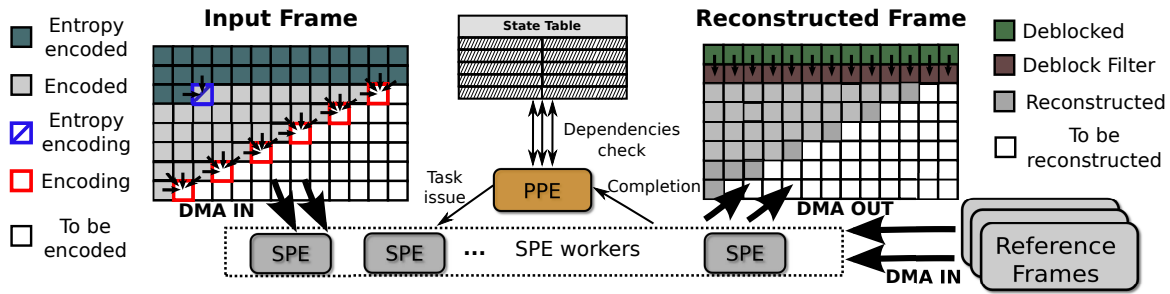


Fig. 2. Design of *c264*, arrows inside frames express the dependencies among different tasks.

limited space for storing both the working set of the active task and prefetching data. Second, we optimize DMA transfers using addresses aligned at 128-bytes boundaries for both source and destination. In addition, we adjust the stride of non contiguous arguments to avoid memory bank conflicts [15]. Third, *c264* shows high numbers of TLB misses, due to strided memory access on the uncompressed frame data. To reduce the number of TLB misses we use large page sizes of 16 MBytes for the raw, uncompressed frame data via the `hugetlbfs` facility in the Linux kernel. We allocate the aforementioned large pages in the initialization phase of the encoder, thus removing the overhead of continuously allocating and freeing pages during the encoding phase. Finally, replication of data structures required for parallelization, due to the existence of multiple outstanding macroblocks, increases significantly the memory footprint. To mitigate the impact of the increased memory requirements we use a custom, pre-allocated memory pool to recycle application data structures.

c) *SPE code optimizations:* *x264* already provides vectorized versions of the kernels for the PPE using AltiVec extensions. In addition, we manually vectorize SPE code, eliminate branches, and partially unroll loops in the encoder kernels. We also expand scalar variables to vectors, where possible, which reduces the overhead of loads and stores of local variables. For example, narrow stores require a read, a scalar insert, and a write operation.

C. Limitations of *c264*

Macro-block level parallelization can increase the available degree of concurrency by exploiting parallelism across frames, using 3D-wavefront parallelization. Although, intra- and inter-frame macroblock-level parallelization are orthogonal, allowing tasks across frames to execute concurrently would require postponing (moving) on the next frame the metadata handling of issued tasks from the current frame. Furthermore, issuing tasks from different frames to the same SPE would require using quantization tables for different frame types, which is prohibitive given the small local store size. In our implementation we wait for all outstanding tasks to complete at the end of each frame before proceeding to the next frame and we leave inter-frame parallelization for future work.

Our implementation has less flexibility in encoding options than *x264* due to the limited size of the local stores: *c264* supports only 16×16 block size for motion estimation/compensation and only inter-frame encoding in B-frames. We modify the calculation of the limits for motion vector

search to avoid producing motion vectors out of the reference region, which limits search window sizes to 128×128 . Although these limitations result in larger video streams for the same video quality, we believe that quality can be improved by using the missing components of *x264*, such as rate-distortion optimization and small matching blockings. In our work, we port only the CAVLC encoder, although *x264* supports both the CABAC and the CAVLC algorithms.

IV. EXPERIMENTAL EVALUATION

A. Experimental Platform and Methodology

We present results from experiments on a Playstation3 console with one 3.2 GHz Cell processor and 256-MBytes of main memory. In this platform, user programs have access to only six of the eight SPEs. We use several video sequences from HD-VideoBench [16]: *blue_sky*, *pedestrian*, *riverbed*, and *rush_hour*. Each video stream consists of 100 frames. In our evaluation we vary three H.264 parameters that affect application behavior significantly: (a) We use two *resolutions* that affect the number of tasks: 720×576 (small) and 1920×1080 (large). (b) We use two *motion estimation algorithms* that affect the computational workload of each task: *diamond* (DIA) and *uneven multi-hexagon* (UMH).

For the rest of the encoding parameters, we keep the quantization parameter constant at 26, we use four B-frames between I- and P-frames and we do not use B-frames as reference frames. We disable the adaptive selection of B-frames number, because the code responsible for the decision is computationally intensive and becomes the bottleneck. We use 128×128 pixels as the search region for the motion estimation process. Finally, we use one reference frame for P-frame encoding and two for B-frame encoding due to the local store size limitation. We always distribute tasks in a round-robin manner across SPEs, provided that there is a free slot available in the SPE task queue; otherwise the SPE is skipped. In our experiments we use one empty task queue slot per SPE, for reasons we explain in Section IV-D.

We show execution time breakdowns from both the PPE and SPEs. We break execution time on the PPE in four sections: *PPE issue*: time spent in the runtime system for issuing tasks, *Queue stall*: time waiting for an empty queue slot in the task queues of SPEs, *Sync wait*: time waiting for specific task or tasks to complete, and *Application*: time spent running application code. Similarly, SPE breakdowns consist of *SPE Task*: the compute time, *SPE transfer*: runtime library and

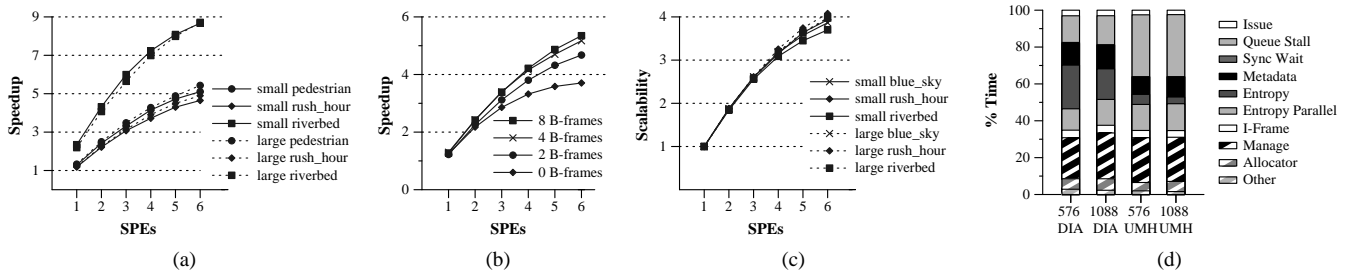


Fig. 3. Speedup of *c264* calculated over PPE-only execution time (a), for different numbers of B-frames for blue_sky with the large resolution (b), and scalability computed over the PPE + SPE execution time (c) using the UMH motion estimation algorithm, 128×128 search region. Breakdowns of *c264* PPE time using six SPEs and different motion estimation algorithms/resolutions (d).

communication time, and *SPE Idle* time. Moreover in some PPE breakdowns we analyze further the application part into: *Metadata*: time for the management of motion vectors, number of coefficients, and calculating pointers to reference pictures. *Entropy Parallel*: time spent executing entropy tasks on the PPE instead of an SPE, if there are no empty task queue slots in any SPE. *Entropy*: time spent at the end of each frame, when there are no additional available parallel tasks to execute. *I-frame*: time spent encoding one I-frame. *Manage*: main loop for dependency checks and proper scheduling of tasks. *Memory allocator*: the time to update offsets/pointers for the recycling of data structures. As a reference point, we include in our results application execution time on the PPE only. Note that we omit I/O time spent in reading the input from and writing the output to the disk. SPE code is transferred to each SPE once at application start-up.

B. Overall speedup and scalability

First, we compare our implementation with the serial version of *x264*. We slightly modify the native encoder to execute the same code path as the parallelized version and we disable features of the serial version that are also disabled in *c264*. In all cases we use the optimized version of kernels for the PPE and SPE architecture. Figure 3(a) shows overall *c264* speedups as the execution time of the serial version running on the PPE only, divided by the execution time of the parallel version running on the PPE and a variable number of SPEs. We achieve speedup between $4.7\times$ and $8.6\times$ on six SPEs compared to the serial version on the PPE. Moreover, our implementation benefits from 20% up to 100% using one SPE. The most interesting case is the riverbed video that achieves super-linear speedup. The riverbed video sequence fails to perform well using the PPE only. In this input set, the temporal model fails to effectively reduce redundant information, since there are limited similarities between neighboring video frames. Thus, the encoder uses mostly intra-frame only encoding, which is expensive due to accesses on the pixels of neighboring macroblocks. Moreover, intra-frame encoding increases significantly the bitrate of the output video and creates extra computation overhead at the final phase of processing, the entropy encoding.

Second, we vary the number of B-frames between I- and P-frames. This parameter can change the computation and communication characteristics of our implementation. Analyze/encode tasks originating from B-frames are more coarse

grain than the same tasks originating from P-frames, in both computation and communication manner. The aforementioned tasks of B-frames use two reference frames, in contrast with P-frames that use only one. Thus, the execution time of the serial version of P-frames is less than execution time of B-frames. Figure 3(b) shows that the speedup increases when using a higher number of B-type frames.

Next, we examine the scalability of *c264*, where we compare the execution time with the PPE and more than one SPEs to the execution time with the PPE and one SPE. Figure 3(c) shows the speedup for different resolutions and input streams. The encoder achieves a speedup between 3.8 and 4.2 on six SPEs. Moreover, we observe slightly better scalability using large resolution, due to a larger number of outstanding tasks in the middle of the frame, compared to the small resolution.

To understand the task overheads and serial parts, we present in Figure 3(d) PPE execution time breakdowns for different resolution and motion estimation algorithms from runs that use six SPEs. Issue time is less than 3% of the PPE execution time, without any dependence on the motion estimation algorithm or the video resolution. The average issue time is about 600 CPU core cycles per task. Queue stall accounts for around 18% and 35% for the DIA and UMH motion estimation algorithm respectively. The resolution of video has minor effect on the normalized stall time, in contrast with the selected algorithm. This indicates that the task granularity of the DIA motion estimation algorithm is much smaller than that of the UMH algorithm.

The serial part of entropy encoding using the DIA motion estimation algorithm is 24% and 17% for small and large resolutions respectively. In contrast, the serial entropy encoding ranges from 6% to 4% for small and large resolution respectively, when using the UMH algorithm. The encoder executes entropy encoding in scan order form in contrast to the wavefront algorithm. This means that we can not issue tasks from the next line until the previous line is fully analyzed and encoded. In our implementation we give priority to the entropy tasks via proper scheduling.

Metadata management accounts for 8% to 14% of PPE total execution time. Although, the number of memory accesses in this module is small, the access pattern is irregular and causes many cache misses. The task management loop (manage) overhead ranges from 20% to 24%. Memory allocation takes 4% and 6% of PPE execution time when using small and large resolution respectively, for the UMH algorithm. In high

resolutions the number of outstanding tasks and metadata increases, which in turn increases overall memory utilization. Queue stall and entropy take significant percentage of PPE time, which indicates that further scalability is possible.

Figure 4(a) shows elapsed time in the x-axis and execution time of different task types in the y-axis. We use the blue_sky video, small resolution, and 6 SPEs in this run. We collect results from only one SPE and we separate the file I/O operations using dotted lines in the graph. SPEs execute a limited number of tasks between two frames. Most of these tasks are memory copies between buffers and initialization of different structures. About 6% of total execution time of PPE is spent in frame initialization and finalization. The deblocking filter task of the luminance component is the most computationally intensive task, taking up to 180000 core cycles. The deblocking filter is partially vectorized and can be further improved. Analyze/encode and entropy encoding tasks also have significant variation in execution time. In B-frames the entropy tasks are smaller than in P-frames, because B-frames achieve better compression than P-frames. Thus, the encoded input information for the entropy encoder is small and the algorithm terminates earlier than the entropy encoding of P-frames. The variation in execution time of tasks leads to load imbalance.

The analysis of data reveals that there are three fundamental bottlenecks: task management overhead, serial parts, and architecture limitations. First, the high task management overhead, up to 40%, is a result of dynamic scheduling and dependency checking, memory management, and macroblock metadata handling. Second, the 2D-wavefront algorithm has some serialization points: frame initialization, frame finalization, and the remaining entropy encoding. Third, there are some architecture limitations related to the size of local stores, that have negative impact in overall performance. The performance of SPE code suffers because we can not enable some compiler optimizations (loop unrolling) and practically eliminates the prefetching of arguments. Moreover, the SPE code takes more than half of the available local store memory. In our design we require multiple copies of the same code region in the SPEs, that waste memory resources.

C. Impact of optimizations

In this section we evaluate the impact of each group of optimizations as described in Section III. Figure 4(b) shows the impact of each group of optimizations on execution time. We group optimizations in three different categories: (i) *Static* scheduling issues all tasks in an antidiagonal and then waits for all tasks to complete; (ii) *Dynamic* scheduling includes the selective dependence checking and issue across antidiagonals; (iii) *Memory* includes memory optimizations and dynamic scheduling. We use the blue_sky input with the high resolution, UMH motion estimation, and a 128×128 search range.

Dynamic scheduling improves execution time over static scheduling by about 30% and reduces synchronization time by 80% on six SPEs. Furthermore, the optimization decreases the SPE idle time compared to static scheduling. Dynamic scheduling issues tasks faster than the traditional 2D-wavefront algorithm and achieves better load balancing between SPEs.

TABLE I
ARCHITECTURE CHARACTERISTICS OF THE TWO PLATFORMS.

CPU	CMOS (nm)	LS Size (KB)	L1 (KB)	L2 (KB)	Total (KB)
Cell	90	6×256	32 I + 32 D	512	2112
Opteron	90	-	64 I + 64 D	2 × 1024	2176

TABLE II
c264 AND x264 ACHIEVED FPS USING THE BLUE_SKY VIDEO AND THE UMH ALGORITHM WITH 128×128 SEARCH REGION.

Resolution	c264	1 thread x86	2 threads x86
720x576	55.2	40.27	60.9
1920x1088	10.2	7.12	12.5

Memory optimizations improve execution time by 25% using six SPEs. We observe that memory optimizations reduce significantly communication time, up to 60% on the SPE, while they also reduce stall time on the PPE by at least 30.1% and at most 42%. Recycling application metadata and buffers has small impact on performance, but allows c264 to run larger input resolutions. The use of large pages has the biggest impact of all memory optimizations and reduces communication time by up to 50% compared to offloading. The use of large pages decreases significantly the number of TLB misses. A side effect of using large pages is that execution time of PPE code is reduced by about 5% in large resolutions. This happens because the SPE sends an interrupt to the PPE for handling each SPE TLB miss. Moreover, we observe a minor impact on the execution time of the serial application running on PPE using large pages (PPE HTLB).

D. Impact of task queue size

Differences in task execution time and long queue sizes can cause significant imbalance and decrease the performance of the encoder. We enforce load balancing, using one slot per SPE queue, although this increases issue and stall time. An entropy task may be delayed behind other tasks in the SPE task queues, which are served in FIFO. On the other hand, the PPE has a two-way SMT architecture. To compensate for priorities we spawn another thread for entropy encoding. The spawned thread is responsible only for entropy encoding.

Figure 4(c) shows the PPE execution time breakdown with six SPEs and varying queue sizes in three configurations: the optimized version with one queue slot on each SPE (left), using a PPE thread for entropy encoding (middle), and using a PPE thread for entropy encoding together with four task slots per SPE (right). The allocator time increases due to locking, as a result of concurrent accesses to data structures. Metadata handling time increases because of memory accesses from the two PPE threads on the shared PPE L2 cache. Note that the sum of synchronization, issue, and stall time is larger than when using one task slot per SPE queue. The serial part of entropy encoding is lower as expected, due to the additional thread that executes only this task.

E. Comparison with Other Platforms

To place our results in context, we also present x264 results on a dual-processor system, with two 64-bit Dual-Core

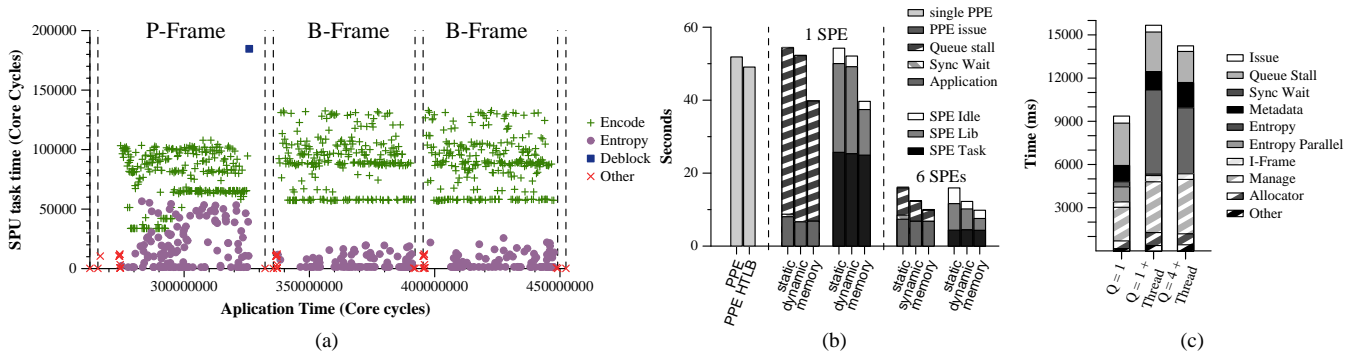


Fig. 4. Timeline with all executed tasks and their respective task execution time, collected from one SPE when all six SPEs are active (a). Impact of optimizations using the *blue_sky* sequence with UMH motion estimation and large resolution (b). PPE breakdowns with different queue slots per SPE (c).

AMD Opteron Processor 2216 running at 2.4GHz. Table I summarizes the characteristics of our architectures. We use six SPEs when we run *c264* on the Cell processor. Table II shows that the Cell outperforms the x86 processor by 37% and 43%, for small and large resolutions, respectively using one thread. However, *x264* outperforms *c264* when two threads are active, by 10% and 22% for small and large resolutions respectively. We observe similar results using different input video sequences. The performance of the AMD processor is similar to the Cell processor, when the implementations in both platforms are well optimized.

F. Programming effort

Table III shows the programming effort in line counts and estimated amount of time for developing and debugging. We mark column A if the feature or modification is forced by the Cell architecture. We also mark the possible solution to avoid the manual coding using a more complex runtime (R) or compiler (C). We categorize programming effort in five broad areas: (i) privatization of data structures, (ii) offloading code to SPEs, (iii) exploiting task parallelism, (iv) memory optimizations, and (v) SPE code optimizations.

Privatization of data structures is necessary in *x264* because the code modifies global variables and arrays in many steps of the encoding process. For the privatization of data structures, most of the time is devoted to restricting global memory accesses and understanding memory accesses in the context of H.264 video encoding.

Offloading code to SPEs requires major effort on the Cell. The most significant difficulty is the readjustment of pointers to point inside the address space of SPEs. Debugging can also be hard because the SPE does not have memory protection for boundaries checking. A compiler can assist offloading, by using static analysis and copying the offloaded code in different files for compilation.

We can divide parallelism exploitation into three smaller categories: preserving dependencies, application metadata handling, and task scheduling. Dependence analysis requires about one day effort because dependencies have been extensively analyzed in previous work [17]. Metadata handling requires about one week because of the difficulty in understanding global memory accesses in the video encoder context. Task scheduling also requires about one week due to the complexity

TABLE III
PROGRAMMING EFFORT EXPRESSED IN LINES OF CODE ADDED TO OR MODIFIED FROM THE ORIGINAL *x264* APPLICATION.

Category	Modification	LOC	Eff.	A	R	C
Privatize data structs	MB Analysis	2735	1 M			X
	MB Encoding	2170	1 M			X
	Entropy Encode	1047	1 M			X
	Deblocking Filter	430	1 W			X
	Frame mem copy	15	1 D			X
Offload to SPE	MB Analysis	2735	1 M	X		X
	MB Encoding	2170	1 M	X		X
	Entropy Encode	1047	1 W	X		X
	Deblocking Filter	430	1 W	X		X
	Frame mem copy	15	1 D	X		X
	Issue, entry code	1206	1 D			X
Task parallelism	Satisfy deps	40	1 D		X	
	Metadata handle	250	1 W		X	
	Task schedule	400	1 W		X	
Mem optimization	Tune strided args	5	1 D	X		X
	Reduce TLB misses	135	1 W	X	X	X
	Recycle data structs	244	1 W		X	X
SPE optimization	Code vectorization	357	1 W	X		X
	Branch elimination	~30	1 W	X		X
	Loop unrolling	~100	1 D			X
	Expand variables	~30	1 D			X

of dynamic scheduling. We can avoid the aforementioned modifications by using a runtime that supports dependence analysis, such as Cell Superscalar [18].

In terms of memory optimizations, tuning of strided memory accesses is straight-forward, because we only need to insert a few lines of code for stride selection during frame initialization. Allocating large pages, using the huge pages interface of the kernel, is more complicated than a library call and requires special handling. Recycling data structures requires about one week, because we need to understand the use of data structures in the context of the *x264*. Finally, SPE code optimization includes all code transformations in kernels: vectorization, branch elimination, loop unrolling and expanding kernel variables. Loop unrolling and expanding kernel variables are straightforward and require minimal effort. On the other hand, hand tuned vectorization is more complicated due to irregular data accesses in some kernels, whereas branch elimination also requires significant effort for only minor performance improvement.

V. RELATED WORK

Park and Ha [19] analyze the expected performance of *x264* parallelization at the macroblock-level for the Cell processor. They offload only the first phase of the encoder, the analysis to SPEs, whereas the rest of the encoder remains on the PPE. Di Wu et al. [6] present a real time encoder of H.264, however, their effort is focused on kernel optimization rather than parallelization. They ignore the entropy encoding and other auxiliary parts of the encoder, such as frame initiation and rate control. We present an end-to-end implementation that parallelizes or offloads every phase of the encoder and show that the task management can occur significant overheads. If we ignore the auxiliary parts of the encoder and decrease the search region we achieve 24 fps on six SPEs, which is close to the performance demonstrated in [6].

Xun et al. [5] use a decentralized pipelined parallel encoding algorithm to achieve real time encoding for high definition H.264 streams using eight SPEs. They manage to decrease task management overhead using decentralized task creation. For efficient communication they use multi-buffering and on-chip communication for data transfers between different modules of the encoder. We use a master-worker execution model that is more general and can be adapted for different architectures.

Previous research has also examined video decoding on Cell. Macroblock-level parallelism can be exploited in intra- and inter-frame and is extensively analyzed by Meenderinck et al. [7]. They use a master-worker programming model similar to ours. Meenderinck et al. also investigate different scheduling policies for macroblock-level parallelism, including a static scheduling approach to improve locality [8]. In contrast, we cannot exploit locality in many cases due to local store size limitations. Although this work is useful and shows the upper limits of performance that can be achieved with upcoming multi-core architectures in general and the Cell processor in particular, it has two main shortcomings: (a) It ignores important dynamic behavior which exists in the encoder and does not exist in the decoder. (b) It does not reveal the effort associated with providing a full working encoder or decoder application on such architectures. Furthermore, our evaluation uses the full H.264 encoding process.

VI. CONCLUSIONS

We presented the design, implementation, and evaluation of a fine-grain task-parallel version of the H.264 video encoder on the Cell processor, the required programming effort, and performance limitations. Starting from an existing, thread-based parallel encoder, *x264*, that uses frame-based parallelism, we redesigned the encoder to use master-worker, fine-grained task-based parallelism with appropriate scheduling. Our implementation of *c264* achieved up to 82 and 16 frames per second for 720×576 and 1920×1080 input resolutions respectively, while running the full encoding process with six SPEs. However, our implementation required significant programming effort, some of which could be mitigated with compiler and runtime support, whole our experimental analysis revealed several sources of performance bottlenecks: task management overhead, serial parts, and architecture limitations

such as local store size and TLB handling on the PPE. We find that the Cell processor is only slightly faster than an x86 processor of similar technology, when software is well optimized on both platforms.

REFERENCES

- [1] H. P. Hofstee, "Power Efficient Processor Architecture and The Cell Processor," in *11th International Conference on High-Performance Computer Architecture (HPCA '05)*, 12-16 February 2005, San Francisco, CA, USA, pp. 258–262, 2005.
- [2] NVIDIA Corporation, "Fermi Architecture White Paper," 2009.
- [3] A. Rodriguez, A. Gonzalez, and M. P. Malumbres, "Hierarchical Parallelization of an H.264/AVC Video Encoder," in *Proceedings of the international symposium on Parallel Computing in Electrical Engineering (PARELEC '06)*, (Washington, DC, USA), pp. 363–368, 2006.
- [4] Yen-Kuang Chen and Xinmin Tian and Steven Ge and Milind Girkar, "Towards Efficient Multi-Level Threading of H.264 Encoder on Intel Hyper-Threading Architectures," in *18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, 26-30 April, Santa Fe, New Mexico, USA, p. 63, 2004.
- [5] Xun He, Xiangzhong Fang, Ci Wang, and Satoshi Goto, "Parallel HD encoding on cell," in *International Symposium on Circuits and Systems (ISCAS '09)*, pp. 1065–1068, 2009.
- [6] Di Wu, Boonshyang Lim, Johan Eilert and Dake Liu, "Parallelization of High-Performance Video Encoding on a Single-Chip Multiprocessor," in *IEEE International Conference on Signal Processing and Communications*, 2008.
- [7] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez, and A. Ramirez, "Parallel Scalability of Video Decoders.," *Signal Processing Systems*, vol. 57, no. 2, pp. 173–194, 2009.
- [8] Chi Ching Chi, Ben Juurlink, Cor Meenderinck, "Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine," in *Proceedings International Conference on Supercomputing (ICS '10)*, June 2010.
- [9] "ITU-T H.264: Advanced video coding for generic audiovisual services," November 2009.
- [10] Videolan. *x264: A free H.264/AVC encoder*, <http://www.videolan.org/developers/x264.html>.
- [11] G. Tzenakis, K. Kapelonis, M. Alvanos, K. Koukos, D. S. Nikolopoulos, and A. Bilas, "Tagged Procedure Calls (TPC): Efficient runtime support for task-based parallelism on the Cell Processor," in *The 2010 International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC '10)*, Jan. 2010.
- [12] S.M. Akramullah, I. Ahmad, M.L. Liou, "Performance of a software-based MPEG-2 video encoder on parallel and distributed systems," in *IEEE Transactions on Circuits and Systems for Video Technology (TCSV '97)*, vol. 7, pp. 687–695, 1997.
- [13] S. M. Akramullah, I. Ahmad, and M. L. Liou, "A data-parallel approach for real-time MPEG-2 video encoding," *Journal of Parallel and Distributed Computing*, vol. 30, no. 2, pp. 129–146, 1995.
- [14] Erik B. Van Der Tol, Egbert G. T. Jaspers, Rob H. Gelderblom E.B. van der Tol, E.G.T. Jaspers and R.H. Gelderblom, "Mapping of H.264 decoding on a multiprocessor architecture," in *Image and Video Communications and Processing*, pp. 707–718, 2003.
- [15] Redbooks, IBM, *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*, ch. 4, p. 325. 2008.
- [16] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "HD-VideoBench. A Benchmark for Evaluating High Definition Digital Video Applications," in *Proceedings of the 10th International Symposium on Workload Characterization (IISWC '07)*, (Washington, DC, USA), pp. 120–125, 2007.
- [17] Zhuo Zhao and Ping Liang, "A Highly Efficient Parallel Algorithm for H.264 Video Encoder," in *International Conference on Acoustics, Speech and Signal Processing*, 2006, vol. 5 of *Acoustics, Speech and Signal Processing*, pp. 489–492, May 2006.
- [18] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta, "Memory - Cells: a programming model for the cell BE architecture," in *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, 2006, Tampa, FL, USA, 2006.
- [19] J. Park and S. Ha, "Performance Analysis of Parallel Execution of H.264 Encoder on the Cell Processor," in *Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '07)*, pp. 27–32, 2007.