Abstract— H.264/AVC (Advanced Video Codec) is a new video coding standard developed by a joint effort of the ITU-TVCEG and ISO/IEC MPEG. This standard provides higher coding efficiency relative to former standards at the expense of higher computational requirements. Implementing the H.264 video encoder for an embedded System-on-Chip (SoC) is a big challenge. For an efficient implementation, we motivate the use of multiprocessor platforms for the execution of a parallel model of the encoder. In this paper, we propose a high-level independent target-architecture parallelization methodology for the development of an optimized parallel model of a H.264/AVC encoder. This methodology is used independently of the architectural issues of any target platform. It is based on the exploration of the task and data levels forms of parallelism simultaneously, and the use of the parallel Kahn Process Network (KPN) model of computation and the YAPI programming C++ runtime library. The encoding performances of the obtained parallel model have been evaluated by system-level simulations targeting multiple multiprocessors platforms.

I. INTRODUCTION

The H.264/AVC has been designed with the goal of enabling significantly improved compression performance relative to all existing video coding standards [1]. Such a standard uses advanced compression techniques that in turn, require high computational power [2]. For a H.264/AVC encoder using all the new coding features, more than 50% average bit saving with 1–2 dB PSNR video quality gain are achieved compared to previous video encoding standards [3]. However, this comes with a complexity increase of a factor 2 for the decoder and larger than one order of magnitude for the encoder [3].

Implementing a H.264/AVC video encoder represents a big challenge for resource-constrained multimedia systems such as wireless devices or high-volume consumer electronics since this requires very high computational power to achieve real-time encoding. For such a video encoder, it may be probably necessary to use some kind of multiprocessor approach to share the encoding application execution time between several processors.

In this paper, we propose a high-level independent target-architecture parallelization methodology of the H.264/AVC encoder based on the use of parallel programming models of computation. In this methodology, the two predominant concepts of parallelism; the data-level partitioning and the task-level splitting and merging are used. The objective is the simultaneous exploration of task and data levels parallelism, with the use of communication and computation workload analysis to get an optimal high-level parallel model of the H.264/AVC encoder.

Starting with the H.264 encoder block diagram and using the task-level decomposition, a first parallel model of the encoder will be proposed. This model is based on the Kahn Process Network (KPN) [4] model of computation implemented by the Y-chart Applications Programmers Interface (YAPI) C++ library [5]. Using the starting parallel model, communication and computation workload analysis shall be considered to identify the potential bottlenecks and thus to provide a global guidance when optimizing concurrency between processes. Based on the obtained results, task-merging and data-partitioning are then explored to get an optimized parallel YAPI/KPN model. The goal of this optimization is to get finally a parallel model with the best computation and communication workload balance.

The outline of the paper is as follow. Section II defines the adopted experimental environment. In section III, we present the starting parallel KPN model obtained by task-level decomposition, the main important issues for the YAPI implementation of this model, and finally the results of its system level functional validation. Section IV discusses the concurrency optimization strategy of the starting parallel YAPI/KPN model using the task-merging and the data-partitioning forms of parallelism. Section V presents the encoding performance simulation results of the proposed parallel model targeting multiprocessor platforms. Finally, section VI concludes the paper.

II. EXPERIMENTAL ENVIRONMENT

For the parallel specification of the H.264/AVC encoder, the JM10.2 [6] software reference version is used with main
profile @ level 4. The high system-level functional simulation of the obtained parallel models has been performed on a General-Purpose Processor (GPP) platform based on an INTEL Centrino 1.6 GHz running a Linux operating system. For the video streaming and video conferencing applications, we used popular video test sequences in the Quarter Common Intermediate Format (QCIF, 176×144 picture elements).

For an optimal balance between the encoding efficiency and the implementation cost, a proper use of the H.264/AVC tools has been proposed in a previous work [7] to maintain an acceptable performance while considerably reducing complexity. In comparison with the most complex configuration, a complexity reduction of more than 80% has been achieved with less than 10% average bit rate increase for all the CIF and QCIF used test sequences [17]. However, the associated sequential execution results in frames per second are of 2.06. Even with this configuration offering an optimal trade-off between coding efficiency and implementation complexity, we are still very far from a real time performance of 25 frames per second. Implementing this configuration of the encoder on embedded multiprocessor platforms represents thus a big challenge to achieve real-time encoding.

The obtained optimal encoding tools have been fixed as follows. An UMHexagonS fast motion estimation scheme, a search range of 8, 4 variable block sizes from 16x16 to 8x8, 3 reference frames, R-D Lagrangian optimization activated, Hadamard transform disabled, motion vector fractional pixel accuracy enabled, a QP value fixed to 28, and a CAVLC entropy encoding technique. In addition, the H.264/AVC standard uses different encoding structures, including the classical coding types and the advanced pyramid coding structures. The influence of these coding structures on performance and complexity is also analyzed in [7]. According to the obtained results, it has been shown that the bit rate output and the PSNR video quality are better using pyramid structures compared to the classical coding structures and that the 3Level-5B frames pyramid hierarchical structure offers the best performance/complexity values. Given this, the 3Level-5B has been adopted for our fixed optimal configuration.

III. TASK-LEVEL IMPLEMENTATION OF THE H.264/AVC ENCODER

The goal of this step is to extract the available task-parallelism by splitting compute nodes as far as possible to get the first starting valid parallel KPN model of the encoder. This model will be implemented using the YAPI multi-threading programming environment. The YAPI implemented parallel model is then validated using high level functional simulations. In this section, we will first present the selected communication granularity level and the starting parallel KPN model proposed using the task-level decomposition. After that, the adopted programming strategy along with the main issues for the YAPI implementation of the starting parallel model is presented. Finally, the YAPI system level functional validation results are discussed.

A. Communication granularity

For many previous task-level parallelization works [8, 9], the Group Of Pictures (GOP), slice or frame level communication granularity has been used. It has been shown that the GOP granularity level provides the best encoding performance. However, for embedded System-on-Chip implementation, the available memory is limited. Using these systems, the GOP or frame level communication granularity is not viable. For example, if the frame level granularity is selected, the associated FIFO communication channels should have at least one frame size. For low video resolutions (like the QCIF format), a minimum of 38 Kilo bytes FIFO size is needed. For higher HD resolutions, around 3 Mega bytes are necessary to ensure an inter task FIFO communication. This is not practical for limited resources embedded systems.

The optimal communication granularity is thus the fine grain level, i.e. at Macro-Block (MB) level since only the current and reference frames needs to be stored. Each frame is considered as the current workload, and the encoding process of each frame is divided between the processors.

B. Starting parallel KPN model

The Task Level Parallelism (TLP) is first considered [10]. This type of parallelism is achieved by decomposing the whole application into separate blocks. Each block defines one single task or process that runs a separate stage of an algorithm. For this case, the application blocks diagram [1] has served as a starting point for extracting the task-level parallelism. Given this, the sequential H.264/AVC encoding algorithm is first split into concurrent tasks that may be executed at the same time, and then the necessary inter-task communication is established using message passing KPN primitives [4].

Given the functional blocks diagram of the encoder and the sequential C-code specification, the starting proposed model is presented in the following figure 1.

Figure 1. Starting parallel H.264 KPN model

The “VidIn” process shown in figure 1 represents the input of the encoder. It is responsible for collecting the video data (YUV frames) from the input file (video sequence with YUV format), the frame width and height dimensions, the total frame number, and the frame rate information. Each frame is divided into “YUVMB” macro-blocks of 16x16 pixels. The
“Dmx” process forwards these macro-blocks to the “Sub”, “Mec”, and “Intra-Pred” processes.

The “Sub” process reads the predicted “PredYUVMbToSub” macro-block, subtracts it from the current “YUVMbToSub” macro-block, and sends the residual data “YUVMbToDCT” to the “Det_Dec” process to perform associated transforms, respectively on the Y luminance and the UV chrominance macro-blocks (MBs). These MBs are first arranged into blocks of 4x4 pixels. Each 4x4 block is first transformed into DCT coefficients block using an appropriate integer transform, then Q quantized and sent as “QuanMb” to the “Vlc” process. The “Det_Dec” is also responsible for decoding “QuanMb” via a rescaling and an inverse transform and transmitting the “DecMb” to the “Add” process. The “Vlc” process receives the quantized DCT coefficients “QuanMb”, performs the CAVLC entropy coding and transmits the resulting “BitStreamFrm” compressed bit stream to the “VidOut” process. Finally, the “VidOut” process sends the H.264 compressed data bit stream to the output file (.h264).

The “Add” process uses the residual decoded “DecMb” MBs and the best inter or intra predicted MBs “PredYUVMbToAdd” to reconstruct the previously encoded “RecMbToIntra” (but un-filtered) macro-block. Using the current MB “YUVMbToIntraPred” output of the “Dmx” process and the reconstructed previously encoded MB “RecMbToIntra”, the “Intra_Pred” process maintains first the storage of this “RecMbToIntra” MB in the reconstructed frame declared as local variable in the “Intra_Pred” process code. Then, it performs an intra-prediction on each macro-block using 9 prediction modes for the 4x4 luma blocks, 4 prediction modes for the 16x16 luma blocks, and 4 modes for the 8x8 chroma blocks. The best intra-prediction mode cost obtained and the associated predicted MB “BestIntraPred” are sent to the “Mode_Dec” process.

Parallel to the intra-prediction process, each “Dmx” output “YUVMbToMotionEst” current macro-block is inter-predicted using one or more reference frames by the “Mec” process. This process is also responsible for maintaining the reference frames memory. The list of past frames is generated through the filtered MBs received from the “DB_Filter” process output. The “DB_Filter” process receives the reconstructed decoded “RecMbToFilter” macro-blocks (only the MBs used as reference) from the “Add” process and information about the references indexes and the motion vectors of this macro-block (already inter-predicted) from the “Mec” process. Then, filtering is applied on each reconstructed previously encoded MB to reduce blocking distortions. The best inter-prediction mode cost obtained along with the corresponding predicted MB are sent as the “BestInterPred” structure to the “Mode_Dec” process. Using the best intra-prediction and inter-prediction modes, the “Mode_Dec” process selects the best optimal “PredYUVMbToSub” predicted macro-block of them and transmits it to the “Sub” process.

C. YAPI programming strategy

For the implementation of the parallel model of figure 1 using the YAPI multi-threading runtime environment, we started with the sequential C reference code of the fixed configuration defined in section II. The sequential code is modified and structured by hand to describe the KPN in C++. Each Kahn process is described by a set of associated functions extracted from the original C code. The inter process communication is performed using solely the YAPI I/O FIFO primitives. Using global variables for this purpose is not allowed. Thus, to ensure inter process communication, all the global shared variables used in the sequential reference code are grouped into associated data structures for communication over FIFO channels.

For efficient task-level decomposition, a YAPI programming strategy is proposed. This strategy is first based on the analysis of the role played by each process in the proposed KPN model. Given this, all the process related functions and data structures are extracted from the sequential source code. The steps of our used strategy are as follows. (1) The definition of the code of each process including all related functions, local and global data structures. (2) Once the extracted code is compiled with no errors, we precede to its reimplementation using solely the YAPI C++ syntax. Global variables are converted into local variables transmitted over FIFO communication channels. (3) Finally, the behavior validation is performed. This step consists in checking for each process of the model that the associated separate code carries out the same computation with the same functionalities as the old sequential source reference code.

D. Important YAPI implementation issues

The main important issues that should be discussed for an effective YAPI implementation of the proposed KPN parallel model of a H.264/AVC reference encoder are as follows:

1) Reference frames memory management:

For the parallel model of figure 1, there are some processes exchanging MB data streams, but there are others like the motion estimation compensation “Mec” process accessing data from reference frames. For this “Mec” process, a full reference frame has to be transmitted over a dedicated FIFO Channel. First, we thought for transmitting full reference frames from the “DB_Filter” process to the “Mec” process. In this case, for low resolution QCIF video format, about 38 Kilo bytes of data are needed for each reference frame. For the encoding configuration used, the reference frame number is fixed to 3. This results in a minimum of 115 Kilo bytes of FIFO size memory needed for transmitting the reference frames between “Mec” and “DB_Filter” processes. Typically, this is not practical, particularly for higher resolution video frames. In addition, transmitting these reference data over the associated communication channels is performed for each encoded MB. This results in a lot of redundant copies that will considerably increase the communication workload overhead and hurt thus the final encoding performance.

Given this and as using global shared variables is not allowed, we opted for the “Mec” process to handle reference frames and to maintain the memory of past frames. The “Mec” process is thus called for receiving from the “DB_Filter” process the intra and inters filtered MBs. After having received all the filtered intra MBs, the “Mec” starts the inter
prediction of the P/B-type MBs one by one starting from the
left.

2) Specialized functions Redundancy:
In the original C source code [6], there are particular
specialized functions that are associated to several processes.
For example, the “GetNeighbour()” function is used to gather
positions information on the neighboring Luminance and
Chrominance blocks. This function and all its children (like
“get_mb_pos()”) are needed by all the “Intra-Pred”, the
“Mec”, and the “Vlc” processes. To use such a specialized
function, the first option consists in their implementation in
only one process. For the others, a dedicated FIFO inter task
communication may be used to compute this function when
necessary. However, implementing this option leads to a
maximum communication overhead and an important data
dependency between processes. To minimize this overhead,
we opted for a redundant implementation of all the specialized
functions at the cost of more computing burden of the
associated processes.

Particularly, this is the case of the Rate-Distortion
Optimization (RDO) technique that has been activated in our
used encoding tools configuration. Setting this RDO option,
some specialized functions in the “Vlc” entropy coding
process are also used by the “Intra-Pred” and the “Mec”
processes. Effectively, to select the best intra and inter
prediction modes using the RDO optimization criterion, some
VLC functions (like “writeMBLayer()”) for computing the
rate (number of bits consumed), thus the cost of every possible
coding mode are required. Also for this case, we opted for a
redundant implementation of all the associated VLC functions
in the “Vlc”, “Mec” and “Intra-Pred” processes to minimize
the communication overhead at the cost of a maximum
computing burden of these processes.

3) Large local variables management:
With the YAPI run-time environment, a separate private
stack space is allocated for each process of the network. This
stack is used to store the intermediate results, the local
variables, and all functions calls from the main member
functions. For the H.264/AVC video encoding, there are a lot
of large local video data structures that are allocated on the
stack. As the total amount of stack space of each process is
fixed to 64 Kilo bytes, this may be insufficient and may result
in a “stack overflow” [11]. Such a stack overflow will lead to
an access violation that causes the program to be killed and a
core dump to be generated [11]. To cope with this, all the
stack large local video data structures have been allocated
dynamically on the heap using the “malloc” and “new”
functions. For the H.264/AVC video encoding, there are a lot
of large local video data structures that are allocated on the
stack. As the total amount of stack space of each process is
fixed to 64 Kilo bytes, this may be insufficient and may result
in a “stack overflow” [11]. To cope with this, all the
stack large local video data structures have been allocated
dynamically on the heap using the “malloc” and “new”
dynamic allocation services.

E. High-level functional simulation of the starting parallel
KPN/YAPI model

Based on the proposed YAPI programming strategy, the
parallel KPN model of figure 1 is implemented using the
YAPI multi-threading programming environment. The YAPI
system level functional validation results of the implemented
model are presented and discussed in this paragraph.

1) Communication workload analysis:

The proposed parallel model of figure 1 has been validated
at YAPI system level. At this level, when this model is
executed, the YAPI “read”, “write”, and “execute” functions
generate information on computation and communication
workload of the application. For a QCIF “Bridge close”
sequence of 13 YUV frames, the communication workload
analysis is obtained, and shown in the figure 2.

This figure describes the total number of Write tokens
(Wtokens) and Read tokens (Rtokens) exchanged over all the
used data channels of the network. The number of tokens per
call is equal to 1 for all the “reading” and “writing” operations
(T/W and T/R). The “Tsize” for one token represents the
average amount of data communicated per call between two
processes. For the QCIF “Bridge close” 13 frames sequence,
each frame consists of 99 macro-blocks of 16x16 pixels. One
macro-block is constituted with two 8x8 blocks of chrominance,
and one 16x16 block of luminance. For example, given the implemented YAPI model, we have 1287
(99*13frames) intra and inter MBs communicated over the
“YUVMB” FIFO channel from the “VidIn” process to the
“Dmx” process. Given the “Tsize” of one token (40752 of 1
byte size), the total bytes number communicated over this
“YUVMB” channel is 1287*40752*1 bytes. For the used 13
frames sequence, and given the adopted 3Level-5B pyramid
coding structure, only 7 are used as reference frames (1 L-
frame, 2 P-frames, and 4 B-frames). These frames are
maintained by the “Mec” process after receiving the filtered
MBs from the “DB.Filter” process via the “FilteredRefMb”
FIFO. These references frames are reconstructed from 693
previously encoded macro-bocks communicated between the
“DB.Filter” and the “Mec” processes.

<table>
<thead>
<tr>
<th>size</th>
<th>Tsize</th>
<th>Wtokens</th>
<th>Rcalls</th>
<th>T/W</th>
<th>Rtokens</th>
<th>Rcalls</th>
<th>T/R</th>
</tr>
</thead>
<tbody>
<tr>
<td>h264.YUVMB</td>
<td>1182</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.YUVMBToSub</td>
<td>2</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.PredYUVMBToSub</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.PredYUVMBToAdd</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.YUVMBToICT</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.GuardMb</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.DecMb</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.RecodeToIntra</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.YUVMBToIntraPred</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.BestIntraPred</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.RecodeToFilter</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.RecodeToFilter</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.YUVMBToMotionEst</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.BestInterPred</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>h264.BitstreamFrom</td>
<td>1</td>
<td>40752</td>
<td>1287</td>
<td>1</td>
<td>1287</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Communication workload of the starting parallel model

The “Tsize” column of one token represents the size of the
data structure communicated per call between two
communicating processes. Given the “Tsize” column values of
figure 2, it is clear that the communication workload is
somewhat unbalanced for this starting computational network.
The “VidIn” and “VidOut” processes are used for
communication with the external environment. The “VidIn”
process is responsible for getting the input video from the
input file. The “VidOut” process stores the H.264/AVC
compressed data to the output file. These are platform
dependent tasks and thus, are not considered in the
communication workload analysis. The very large exchanged
data structures are outputs of the “Dmx”, “Sub”, “Dct_Dec”, and “Add” processes. The remaining tokens exchanged between the others tasks are all balanced.

Typically, to get better communication behavior, the data level parallelism and the task level splitting or merging should be used. Data level parallelism consists in splitting the data communicated over selected channels thus duplicating the associated tasks of the model. Task level merging consists in combining pipelined tasks that are exchanging large data structures. The task level splitting extracts the available task-parallelism by further splitting the compute nodes. The decision on data splitting and task merging or splitting will depend on the computational workload analysis of the network.

2) Computation workload analysis:

Typically, tasks will not need the same amount of processing time. Thus, a computational workload analysis is considered. For this purpose, a parallel computational “Gprof” GNU [12] profiling is performed. The obtained results are reported in figure 3 in terms of the CPU time percentage spent in the process execution.

![Figure 3. Parallel computational profiling of the first proposed model](image)

Given the profiling results of figure 3, it is clear that the computational workload of the model is too much unbalanced. Some processes have negligible complexity; others especially the “Mec” is still the most computational-expensive task with more than 50% of the total computing time complexity. This is because of the activated Rate-Distortion Optimization (RDO) option which maximizes significantly the coding gain at the cost of a very high computational complexity [13].

Finally it is clear, using the obtained communication and computation workload analysis results, that the starting model of figure 1 does not have good concurrency properties. This outlines the potential of using different steps of task level splitting or merging and data level splitting to derive in a structured way a parallel implementation of the encoder that has a balanced computational workload and good communication behavior.

IV. DATA-LEVEL SPLITTING OF THE MOTION ESTIMATION AND COMPENSATION PROCESS

This section presents the different steps that have been used to derive in a structured way a parallel implementation of the H.264/AVC encoder that has a balanced workload and good communication behavior. This optimized model has been obtained using the task level merging and data level splitting techniques. The task-merging has been used to merge the “Dct_Dec”, the “DB_Filter”, the “Sub”, the “Dmx”, and the “Add” processes into only one “Dct_Dec_Filter” process. In this case, the associate channels transmitting very large token structures are thus removed. For the most computational-expensive “Mec” task, data splitting is proposed for a better concurrency optimization. Given the profiling results of figure 3, the “Mec” process has been split into three “Mec1”, “Mec2” and “Mec3” processes. However before deciding for the partitioning of the communicated data to the motion estimation compensation processes, a data dependency analysis applied for these processes has been considered to minimize the dependencies and to maximize the parallelism rate between the triple decomposed tasks.

A. Data dependencies analysis in the motion estimation and compensation process

Several types of data dependencies are introduced in the H.264/AVC standard. In this section, we are concerned only with the data dependencies of the motion estimation and compensation module as the data-partitioning is performed solely for the “Mec” process.

For the inter prediction of a current MB, the “Mec” module requires the left, top, and top-right MBs as shown in figure 4 (a). In fact, the Predicted Motion Vector (PMV) is first determined using the motion vectors of the neighboring MBs and their corresponding reference indexes. Then, the difference between the final optimal motion vector and the PMV is encoded. On the other hand, the inter-prediction module needs the previously encoded reference frames. Before processing a current MB, the co-located MB and the minimum of its eight neighboring reference frame MBs should be available as shown in figure 4 (b).

![Figure 4. Data dependencies in the inter-prediction module](image)

B. Data partitioning strategy

As mentioned in section III.A, the communication granularity at the MB-level is selected. The encoding process of each frame is performed MB by MB beginning from the left side. The processing of each frame is thus divided by the different processes of the network. For this, when starting the inter-prediction of a current MB, we have the certitude that the reference data of the co-located MB and their eight neighboring MBs of the previously encoded frame are available in the DPB list of the past encoded frames.

However we are not sure that all the motion vectors of the neighboring MBs of the same frame under processing are already available since each “Mec1”, “Mec2”, or “Mec3”
process will compute separate associated MBs. To minimize the spatial data dependencies between these three inter-prediction “Mec" modules, we propose to split each frame into three MBs regions. Each region consists in a columns number of MBs, as shown in figure 5. For example as observed in figure 5, the total number of MBs communicated from the “VidIn” process to the “Mec1" and “Mec2” processes is equal to: 
\[
((\text{Width}/16)/3) \times (\text{Height}/16) \quad \text{(i.e. 27 MBs for one QCIF resolution low frame)}.
\]

Tripling the “Mec" process results in tripling the associated input and output FIFOs channels. For example, three “YUVMbToMotionEst” FIFOs are used and so around only the third of the total communication load is transmitted over each FIFO channel. This represents 27*40752 bytes per QCIF frame that are copied MB by MB to the “Mec1" and “Mec2” processes. The token structure has not been modified but only the number of communicated tokens is reduced. For the output FIFOs: “RefIdxMvToFiler", “RefIdxMvToVlc", and “BestInterPred", the same procedure has been used.

Once one MB is filtered, it will be copied in one or two associated reference FIFOs. For example as observed in figure 6, the total number of filtered reference MBs, outputs of the “DB_Filter” process and received by the “Mec1" module, is equal to: 
\[
((\text{Width}/16)/3 + 1) \times (\text{Height}/16) \quad \text{(i.e. 36 filtered MBs for one QCIF resolution low frame)}.
\]

The main important problems that we encountered for the effective data-splitting of the three inter-prediction modules are discussed as follows:

1) Data dependencies between the “Mec" processes:
As shown in figure 5, when starting the inter-prediction of the MBs of the column “i" (last column of the associated data “Mec1” process region) from the second line of the frame under processing, the “Mec1" have to receive motion data of the top-right neighboring MBs from the “Mec2" process. On the other hand, to process the MBs of the column “j" (first column of the associated data “Mec3” process region), the “Mec3" process requires the motion data of the left neighboring MBs from the “Mec2" process.

However, before starting the inter-prediction of the MBs of the column “i+1" and these of the column “j-1" starting from the second line, the “Mec2" process should have read the motion data of respectively the left neighboring MBs from the “Mec1” process and the top-right neighboring MBs from the “Mec3” process.

2) Reference frames memory:
As it is applied for all the channels connected to the “Mec," processes, three “FilteredRefMb” FIFOs are used. Typically, it is not possible to split the reference frame between the three used FIFOs since the multi inter-prediction modules need always the same reference frames. However, given the performed data dependencies analysis, it is sufficient to use the co-located MB and its eight neighboring MBs of the corresponding reference frame to inter-predict a current MB. For this, we propose to partition the reference data and thus to send to each “Mec," process just the needed reference data. For this, we suggest to split each reference frame into three partitions and to transmit each reference data partition to only its corresponding FIFO, as shown in the figure 6.

Once one MB is filtered, it will be copied in one or two associated reference FIFOs. For example as observed in figure 6, the total number of filtered reference MBs, outputs of the “DB_Filter” process and received by the “Mec1” module, is equal to: 
\[
((\text{Width}/16)/3 + 1) \times (\text{Height}/16) \quad \text{(i.e. 36 filtered MBs for one QCIF resolution low frame)}.
\]

The optimized parallel model obtained is given in figure 7. This figure clearly shows the task-merging of the “Dct_Dec_Filter" process, and also the data-partitioning for the “Mec" process into the “Mec1", “Mec2", and “Mec3" processes with the appropriate connections between the “Mec," processes and their environment. This model has been implemented and validated at YAPI system level. The communication workload results are obtained and shown in figure 8 for the same QCIF “Bridge close” 13 frames sequence. A computational “Gprof" profiling is also performed and reported in figure 9.

It is clear from figure 8 that the total number of tokens communicated from/to the motion estimation and compensation processes has been reduced. Effectively, the number of tokens transmitted over the “YUVMBToMotionEst1" connecting the input of the “Mec1" process has been reduced to 324 MBs tokens (27 MBs per frame * 12 P&B frames). The total bytes number communicated over this channel is 40752*27*1 bytes. Among 7 reference frames, only 252 filtered reference MBs tokens of 1032 bytes token size (36MBs * 7) are copied in the “FilteredRefMb1" channel from the “Dct_Dec_Filter" process.
to the “Mec1” process. Given the obtained “Tsize” column values of figure 8, except the large token data structure transmitted into the “YUMBToDCT” FIFO, it is clear that the optimized proposed model has better communication behavior compared to the starting model.

In addition, as indicated in figure 9, the data partitioning of the motion estimation and compensation processes comes with a decrease in the computational burden of these processes, and thus a better computational workload balance of the model is observed. The final proposed model has obviously better communication and computational behavior compared to the first starting model. Anyway, one can further use the data parallelism for the motion estimation and compensation to more reduce the computational workload on the associated processes.

V. MULTIPROCESSOR SIMULATION RESULTS

The system-level simulation and modeling framework Sesame/Artemis [14] has been used to evaluate the encoding performances and the compression speedup obtained for the proposed parallel model of the H.264/AVC encoder targeting multiprocessor platforms.

Using the Sesame system-level design methodology, three software model specifications are required: the application process network model, the target architecture model, and the mapping model of the application onto the architecture. For this, the optimized parallel model given in section IV is first ported to the Sesame framework. This has been performed by transforming the YAPI model into a C++ PNNRunner (Process Network Runner) application. The network model is then simulated with the PNNRunner simulator to generate a computational and communication events traces of the application execution, called trace-event queues [14].

Parallel to the application model specification, the target architecture is modeled with the Pearl object-based simulation language. For this, the Sesame environment provides a small library of architecture component models. These are consisting of black-box base models of a processing core, a generic bus, a generic memory, and several interfaces for connecting these base model building blocks. Once a target architecture model is validated, a trace-driven cosimulation of the application events traces queues mapped to the architectural components is carried out. Such a cosimulation requires an explicit mapping of the KPN processes and channels to the particular components of the target architecture. More than one KPN process can be mapped to a same processor. In this case, the system simulator automatically schedules the events from the different queues [15].

For our case, we used three platform models. The first platform is mono processor. The second is based on two processors and the third is with four processors. We used general purpose processors (assumed to be MIPS R4000). For the memory we selected a DRAM model along with a 64 bits wide bus. Communication between components is performed through buffers in shared memory. Mapping application processes to this platform has been decided given the obtained profiling results of figure 9. For the bi-processor platform, the total computational load has been distributed between the two processors. The “Mec1”, “Mec2”, and “Dct_Dec_Filter” processes are mapped to one, and all the others to the second processor. With the four processors target architecture, the “Mec1”, “Mec2”, “Mec3”, and “Intra-Pred” are mapped separately to the different cores to guarantee a competitive execution between them. The “Dct_Dec_Filter” process is added to the “Mec2” process to run on the same core. The “Vlc” is also added to the “Intra-Pred” process and mapped to the fourth processor. The encoding simulation results of 7 YUV frames of the QCIF “Bridge close” sequence obtained using this multi-core shared-memory-based architecture are presented in the following table I and compared to those previously obtained in references [16] and [17]. In fact, these two data-partitioning-based works showed that the computing performance and compression speed are better compared to the several others parallelization works in the literature [17].
It is clear from this table that the obtained encoding performance in frames per second is getting better linearly with the number of used processors. Compared to the data-level parallelization approaches proposed in [16 and 17], our solution, based simultaneously on the task and data levels parallelism, achieved a better execution speedup. In fact, for references [16] and [17] data splitting is performed respectively at the MBs row and MBs region granularity levels. However, for our case a more fine-grain Macro-Block communication granularity level is exploited. Thus, with a more fine grain data amount exchanged by the processors, our proposed methodology is more appropriate for embedded multiprocessor SoC implementations given their limited on-chip memory resources.

Finally, as shown in figure 9, the distribution of the computational workload between the four used cores is nearly balanced. In this case, the processors will operate most of time in parallel, and so for each of them a little idle time will be spent on synchronization with the others.

VI. CONCLUSION

For its cost-effective multiprocessor implementation, the H.264/AVC encoder has been parallelized at high system level using a parallel streaming programming model. Starting with the reference C code, a first parallel model of the encoder is proposed. This model, based on Kahn Process Network (KPN) model of computation, is implemented using the YAPI multithreading environment. Using communication and computation workload analysis of the proposed KPN/YAPI model, it is shown that the first proposed parallel model does not have good concurrency properties. Based on these results, different steps of task level merging and data level splitting are used to derive in a structured way a parallel implementation of the H.264/AVC encoder that has a balanced computation workload and good communication behavior.

To evaluate the encoding performances of the proposed optimal parallel model, the system-level simulation and modeling framework Sesame/Artemis has been used targeting multiple multiprocessors platforms. The obtained simulation results are promising in reference to other data-level parallelization approaches proposed in the literature.

REFERENCES