High-Level Architecture Exploration for MPEG4 Encoder with Custom Parameters

Marius Bonaciu, Aimen Bouchhima, Wassim Youssef, Xi Chen, Wander Cesario*, Ahmed Jerraya

TIMA Laboratory, Grenoble, FRANCE, +33(0) 4 76 57 43 34, {firstname.surname@imag.fr}
*MND, Paris, FRANCE, +33 (0) 1 30 57 61 90, wcesario@mnd.fr

Abstract - this paper proposes the use of a high-level architecture exploration method for different MPEG4 video encoders using different customization parameters. The targeted architecture is a heterogeneous MP-SoC which may include up 2 coarse grain SIMD (task level SIMD) subsystems to perform the computations. The customization parameters are related to video resolution, frame rate, Communication Network, level of parallelism and CPU types. These parameters are determined during the high-level architecture exploration, by estimating the architecture performances at early stages of the design flow. Experiments shows that the error factor of these high-level performances estimations are less than 10% compared to those obtained with final manually implemented RTL architecture. This method was used successfully for exploration of different MPEG4 architecture configurations with different customization parameters. We consider these experiments a breakthrough because they show how a complex design can be mastered through a set of pragmatic choices.

Keywords – Multiprocessors SOC architecture, Video encoder, MPEG4, Architecture exploration, Customization

I. Introduction

Video encoding is widely included in most of consumer, multimedia, mobile and telecommunication applications [1], and becomes a key technology for many future applications. These different applications impose different constraints on the encoding parameters (i.e. video resolution) and on the resulting design (cost, speed and power). Even if MPEG4 is an accepted common standard for most embedded systems domains, a plethora of MPEG4 architectures exist today to comply with different applications [2].

MPEG4 requires a huge amount of computations, and thus needs parallel computations and hardware accelerations. Encoding for digital cinema system (HDTV 1920x1080 video resolution) using full motion search, it requires 32TIPS (Tera Instructions Per Second, 10^{12}) as computation power. This corresponds to a generic platform with 32000 RISC processors running at 1 GHz in parallel. Current technology doesn't allow such integration, and such design is difficult to program and debug [3]. This implies an expensive design process, and it's out of reach of many applications domains.

Implementing an MP-SoC architecture until the RTL level, starting from a wrong set of ad-hoc parameters (i.e. CPUs number or communication topology) might turn out to be very costly. Each modification of parameters, will lead to the need of expensive modifications (which might also lead to a deteriorated final result, because new bugs may result after these "forced" modifications). In the worst case, it might require a complete redesign of the architecture. Very few products may justify such design budget, and the only working solution to get video encoding for low cost products (such as consumers) is to reduce the design cost of the product. The key solution to reduce the design cost is to explore the architecture at high-level, before the low-level architecture is implemented.

This paper proposes an efficient high-level architecture exploration method for different MPEG4 video encoders, using different customization parameters. This work concentrates only on performance estimations in term of speed.

A. Solution space for MPEG4 encoder on MP-SoC

Implementations of MPEG4 video encoder on MP-SoC can be applied in multiple domains: video surveillance, camera recorders, mobile telecommunications, home entertainment, etc. Each of them requires specific architecture configurations, and imposes their own constraints in term of speed, power and chip surface. Finding the final implementation solution requires adjusting a large number of parameters. These parameters can be split into two categories:

1) *Standard MPEG4 Algorithm parameters* are related specifically to the algorithm functionality: video resolution, frame rate, bitrate, quantization range, quantization type, motion estimation precision, motion search area, progressive/interlaced encoding, key frame rate, scene change detection, etc [4]. As it will be shown later in this paper, these algorithm parameters are not sufficient for implementing the MPEG4 video encoder on MP-SoC. To be able to implement the MPEG4 video encoder on a parallel architecture, the algorithm should be able to be easily parallelized / pipelined, by adding parameters for parallelism/pipelining support.

2) *Architecture parameters* are related to the targeted MP-SoC architecture: number of CPUs to be used, type of CPUs, HW-SW partitioning, communication topology, blocking/non-blocking protocol, arbitration type, message sizes, data width, maximum allowed data transfer latency, transfer initialization latency, etc.

B. Classical exploration flow

In classical exploration flows [5][6][7][8] (Fig.1a) , the designer implements the *Algorithm Specifications* starting from a set of already chosen *Algorithm Configurations*. After that, the *Architecture Specifications* is implemented, which should match with the *Algorithm Specifications*. In the end, the *Algorithm Specifications* and *Architecture Specifications* are combined, to obtain an *Algorithm/Architecture Executable Model*. This model simulates the algorithm and architecture running together, and it's used for *Performance Estimations*. If these estimations are not satisfying the requirements, the designer has to modify/redesign the algorithm and/or architecture specifications. This flow has some weak points:

a) The exploration space is highly reduced. The reason is that when having to change the algorithm and/or architecture specifications, the only things which can be changed is related to the parallelism/pipelining functionality of the algorithm on the architecture, some mapping decisions [8], data organizations and communications [5][6]. Any change leads to the need of completely redesigning the specifications, which means "restarting" the project.

b) Building the algorithm/ architecture executable model has to be done manually, which is a fastidious work, requires long design time, and might induces many errors[7]. Also, this model has to be re-designed every time the algorithm and/or architecture specifications are changed. The simulation speed of this model depends on the used abstraction level. If the abstraction level is too low [7], the simulation speed becomes unacceptable long.

c) The performance estimation precision depends on the used abstraction level. No matter the level of abstraction, the estimation precision represents a key issue. In [5][6], the performance estimations are covering only the communication. In [8], the performance estimations are covering precisely the computations, but the communication performances are estimated using an "always available" shared memory. This is insufficient if other communication topology is required. In [7], the estimations are precise, but the lack of any abstractions makes the simulations very long.

C. Contribution

The key contribution of this paper is a working solution for architecture exploration, used for the implementation of the MPEG4 video encoder on MP-SoC. We use a flexible target architecture and a flexible modeling strategy, that allow for both algorithm/architecture exploration. Compared with the previously presented classical exploration flow, our proposed exploration flow (Fig.1b) is able to cover multiple requirements:

a) The need to explore a large solution space is solved by automatically generating the *Executable Algorithm / Architecture Model*. Different customized *Executable Algorithm / Architecture Models* can be obtained from a unique *Flexible Algorithm/Architecture Model for MPEG4*. This model provides the possibility of automatically customize the algorithm and build the abstract architecture, based on a set of *Algorithm/Architecture Configurations*.

b) The need of obtaining a fast simulation is covered by doing the architecture exploration at high-level. As result, by ignoring many low-level architecture details in *the Algorithm/Architecture Executable Model*, the simulation becomes fast.

c) The need of precise simulation results is solved by using a High-Level Architecture Exploration which provides estimation results with high precision. This is done by using precise estimations for the computations and communication times, by time annotating both computations and communications. Additionally, the exploration is capturing the computations and communications running together, to estimate the performances of the entire system.

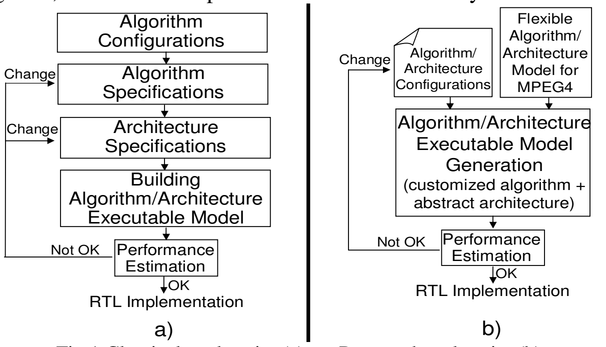
Algorithm Configurations
Change
Algorithm Specifications
Change
Architecture Specifications
Building Algorithm/Architecture Executable Model
Not OK
Performance Estimation
OK
RTL Implementation
a)
Change
Algorithm/ Architecture Configurations
Flexible Algorithm/ Architecture Model for MPEG4
Algorithm/Architecture Executable Model Generation (customized algorithm + abstract architecture)
Not OK
Performance Estimation
OK
RTL Implementation
b)

Fig.1 Classical exploration(a) vs. Proposed exploration(b)

By using such high-level architecture exploration, the time required to obtain efficient architectures of MPEG4 video encoder into MP-SoC is decreased drastically. This architecture exploration is achieved at High-Level using the *Flexible Algorithm / Architecture Model for MPEG4*. This decreases the time needed to obtain and test multiple architecture configurations. Thus, the time required to find acceptable architecture configurations is reduced. The proposed approach was applied successfully for the generation of several configurations of MPEG4 encoders.

The rest of the paper is organized as following. Section 2 presents the *Flexible Algorithm/Architecture Model for MPEG4*. Section 3 details the models used during the proposed high-level architecture exploration. Section 4 shows the architecture exploration flow for MPEG4 encoder with custom parameters. Section 5 presents the experiments and results, followed by conclusions in Section 6.

II. Flexible Algorithm/Architecture Model for MPEG4

This section presents the MPEG4 encoder algorithm and the *Flexible Algorithm/Architecture Model for MPEG4*, which will be used to generate different models required during the exploration.

A. MPEG4 video encoder algorithm

In this work we used the DivX specifications. The DivX is a popular algorithm implementation of the MPEG-4 video compression technology (ISO/IEC 14496-2). The idea of this technology is to compress and store only the *spatio-temporal* differences between consecutive frames. A block diagram of the DivX algorithm is shown in Fig.2. Describing each block is out of this paper's scope, and more details can be found in [4][9].

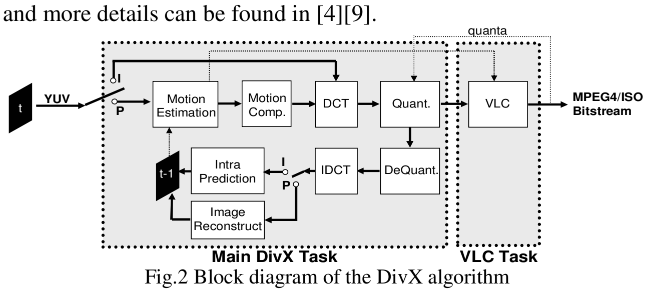

Fig.2 Block diagram of the DivX algorithm

The initial MPEG4 encoder algorithm is a sequential algorithm. The *Standard MPEG4 Algorithm parameters* don't provide any automatic parallelism/pipeline support, which makes difficult to implement the MPEG4 encoder on multi-processors. This drastically reduces the exploration space.

So, the need of inserting parallelism and pipeline support is required. The parallelism/pipeline shouldn't change the algorithm specifications, only the implementation will be different. Our goal is to build a *Flexible MPEG4 Encoder algorithm*, which supports the *Standard MPEG4 Algorithm parameters* plus parameters for the *level of Parallelism/Pipeline*.

For this, the MPEG4 algorithm was grouped into 2 pipelined tasks: *MainDivX task* and the *VLC (V̲ariable L̲ength C̲oding) task*. The *MainDivX* is processing the current image relative to the previous one, and its results are motion vectors, quantized DCT *Macro-Blocks* (image zones of 16x16 pixels). The *VLC* is compressing these results using a Zigzagging and Huffman compression. The final output respects the MPEG4/ISO standard.

Several approaches for parallelization can be found in [10][11]. In [10] and [11], the image is split into smaller areas to be able to achieve parallel computations. In their approach, the *MainDivXs* and *VLCs* were not separated, thus splitting the image into areas required an equal number of *MainDivXs* and *VLCs*. Also, a *VLC* has to wait for the corresponding *MainDivX* to finish its computations for all the *Macro-Blocks*, which means that the *VLC* is 90% into idle mode.

In our work the algorithm is divided into two pipeline stages: the first contains the *MainDivXs*, and the second contains the *VLCs*. We use SIMD architectures for each of the pipeline stage, to handle heavy computations. Since the two stages require different computation powers, the structure of both SIMD may be different in term of number and type of CPU. To adapt the algorithm to this architecture, the image is split into areas, and the computations are done in parallel for each of them.

We've exploited the fact that the *VLC task* doesn't have to wait for the *MainDivX task* to finish processing the entire image. By adapting the *VLC* task to work at *Macro-Block* level, once the *MainDivX task* finished processing a *Macro-Block*, the *VLC task* can start to compress it, while the *MainDivX task* continues to process the next *Macro-Block.* Also, the *VLC task* requires very few computations but large memory (because the need to store its standard Huffman tables). To reduce the memory for the *VLC tasks*, we use a number of *VLC tasks* much smaller than the number of *MainDivX tasks*, but just enough not to become a computational bottleneck. As results, the processing and compression are executed in parallel, and the application's memory is reduced.

Along with these tasks, the use of 4 other smaller tasks is required: *Video*, *Splitter* (for image splitting), *Combiner* (for final reordering) and *Storage* (Fig.3*)*.

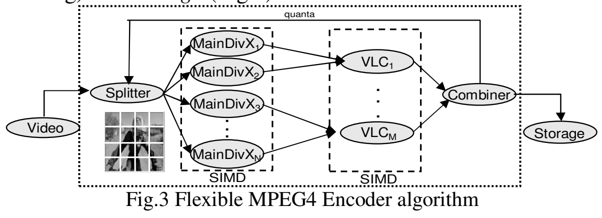

Fig.3 Flexible MPEG4 Encoder algorithm

The *Video* task doesn't belong in the final design. It's a test-bench task that simulates a video source. It sends the video under the form of a pixels stream, compatible with YUV420 standard to the *Splitter*. The *Splitter* divides the image and routes the pixels to the corresponding *MainDivX*, which processes the image. Once the *MainDivXs* processed a *Macro-Block*, its results are sent to a corresponding *VLC*, which will compress the *Macro-Blocks* one by one.

The compressed *Macro-Blocks* are then sent to the *Combiner*, which reorders all the *VLC* results, in order to obtain an MPEG4/ISO bitstream. Also it adjusts the quantization value to be used for the encoding of the future images. The bitstream is sent to the output *Storage* task, which is another test-bench task simulating a storage support. As result, the architecture's behavior is composed of 2 pipelines. One pipeline at frame level between the *Splitter* and the rest of the architecture using a lock step synchronization at frame level, and a second local pipeline at *MacroBlock* level between the *MainDivXs* , *VLCs* and *Combiner.*

Coarse grain parallelization was chosen instead of fine grain one (i.e. every basic function of the *MainDivX task* to be a different task) for simplicity and efficiency. Instead of achieving the parallelism by dividing the algorithm into multiple tasks, resulting in computation distribution on the architecture, the parallelism is done using data distribution. If one of the tasks becomes a bottleneck, the amount of data associated to that task is reduced [12]. Also, using fine grained partitioning for highly called tasks induces serious performance degradation, because of the required big number of context switches.

B. Configuration parameters

To explore the architecture for the MPEG4 video encoder, we use 2 categories of configurations parameters, shown in Table 1.

Algorithm parameters	**Architecture parameters**
Level of Parallelism/Pipeline	Number of CPUs
Video resolution	Type of CPUs
Frame rate	HW-SW partitioning
Bitrate	Communication topology
Key frame	Blocking/Non-blocking comm.
MotionEstimation precision	Arbitration type
MotionEstimation search area	Message size
Progressive/Interlaced mode	Data width
Scene change detection	Data transfer latency
Quantization range	Transfer initialization latency
Quantization type (H263,MPEG4)	Transfer close latency

Table 1. Algorithm and Architecture parameters to be explored

C. Flexible Algorithm/Architecture Model

The *Flexible Algorithm/Architecture Model for MPEG4* (Fig.4) is composed of *Modules* and an *Abstract interconnect execution model*. Each *Module* contains one task, which can be of 2 types:

a) *flexible tasks*, which has at least one of following characteristics:
- flexible computations – tasks which are belonging to a SIMD: *MainDivX* and *VLC*
- flexible input – tasks which are receiving data from a SIMD: *VLC* and *Combiner*
- flexible output – tasks which are sending data to a SIMD: *Splitter* and *MainDivX*

b) *fixed tasks* – none of the above: *Video* and *Storage*

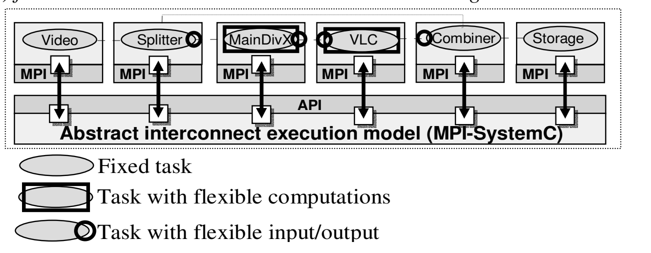
Video
Splitter
MainDivX
VLC
Combiner
Storage
MPI
API
Abstract interconnect execution model (MPI-SystemC)
Fixed task
Task with flexible computations
Task with flexible input/output

Fig.4 Flexible Algorithm/Architecture Model for MPEG4

Tasks are communicating via *API* calls. The interconnections are done through an *Abstract interconnect execution model*, which in [13] is called *High-Level Parallel Programming Model* (HLPPM). A HLPPM hides completely the low-level architecture details: Communication Network, HW/SW & HW/HW Interfaces.

The targeted architecture model is also flexible as shown in Fig.5. It features 2 SIMD and an interconnect structure. The *Splitter* and *Combiner* may be HW or SW. For example, in case of Fig.5, it was freely chosen to map the *Splitter* and *Combiner* on HW, to avoid them to become I/O bottlenecks.

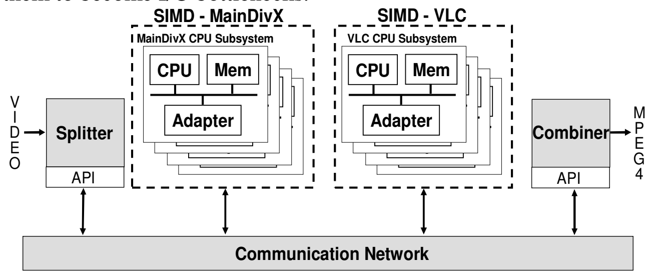

Fig.5 Abstract architecture model with 2 SIMD

III. Algorithm and architecture representation

During the architecture exploration, a unique representation will be used to combine both the architecture and the MPEG4 algorithm. This combined architecture/algorithm format will be used to model different customized algorithm/architecture instances at different levels of abstraction, starting from the pure algorithm model down to the architecture. All these models are executable SystemC models that can be simulated and used for performance estimations, debug, and as entry for design. This section describes in details all the models used during the high-level explorations.

A. Flexible Algorithm/Architecture Model for MPEG4

This model is a macro-code made of a set of generic SystemC modules containing each of them a single task written in C/C++. Tasks are communicating through message passing by calling a set of MPI primitives (see Fig.6)

```
MP_Init( *this,argc,argv);
MP_Finalize( *this);

MP_[I]Send( *this,buf,count,datatype,dest,tag,comm);
MP_[I]Recv( *this,buf,count,datatype,source,tag,comm,status);

MP_[I]BSend( *this,buf,count,datatype,dest,tag,comm);
MP_[I]BRecv( *this,buf,count,datatype,source,tag,comm,status);

MP_[I]SSend( *this,buf,count,datatype,dest,tag,comm);
MP_[I]SRecv( *this,buf,count,datatype,source,tag,comm,status);

MPI_Wait( *this,request,status);
MPI_Test( *this,request,flag,status);
```

Fig.6 MPI communication primitives subset

An example of a task using MPI primitives is illustrated in Fig.7.

```
//------------------ MainDivX"N" task ---------------------------------------------
void MainDivX"N"_MAIN(*image_memory"N",height"N", length"N", top_border"N",
                          left_border"N", bottom_border"N", right_border"N",&result)
{
  //initialization of computations
  MainDivX"N"_INIT (&image_memory"N", height"N", length"N");

  //data_receive_communication from the Splitter
  MPI_"PROTOCOL"Recv(this,&image_memory"N",sizeof(image_memory"N"),
                          "DATA_WIDTH",SPLITTER_ID,22,MPI_COMM_WORLD);

  //calls the function with flexible computations
  MainDivX"N"_COMPUTE (&image_memory"N",height"N", length"N",
                            top_border"N", left_border"N",
                            bottom_border"N", right_border"N",&result);

   //send_results_communication to the coresponding VLC
   MPI_"PROTOCOL"Send(this,&result,sizeof(result),"DATA_WIDTH",
                          VLC["target_vlc"]_ID,22, MPI_COMM_WORLD);
}
```

Fig.7 Example of task description using MPI primitives

*MPI_”**PROTOCOL**”Recv(this,&image_memory”**N**”,sizeof(image_memory”**N**”),”**DATA_WIDTH**”,SPLITTER_ID,22,MPI_COMM_WORLD)* receives data from other task. This primitive specifies the pointer were data will be stored, the amount of data, the communication data width and the unique ID assigned to the source task (*Splitter*). It can be noticed that most parameters are not yet fixed.

Flexible tasks with flexible computations are parameterized to be duplicated and work on different data. For example, the *MainDivX* task uses a set of parameters that specify the address where the image is stored, the image size, the border characteristics, the results storing address, and it uses them to call its computations.

Flexible I/Os are coded using Generate like loops. Fig.8 shows a part of the code of the *Splitter*.

```
void Splitter()
{
  for (target=0; target<"N"; target++)
  {
      MPI_"PROTOCOL"Send(&this,data[target],"BURST_SIZE",
                             "DATA_WIDTH",MainDivX[target]_ID,
                             22,MPI_COMM_WORLD);
  }
}
```

Fig.8 Describing the flexible I/O in the Splitter

The loop executes ***“N”*** MPI_Send to split the image among the *MainDivX* modules performing the encoding. Even the protocol can be parameterized. In Fig.8, the MPI_***“PROTOCOL”***Send can be expanded into MPI_**I**Send, MPI_**B**Send or MPI_**S**Send, according to the ***“PROTOCOL”*** parameter of communication. [13] details the differences between these protocols. The ***“BURST_SIZE”*** sets the communication message size, and the ***“DATA_WIDTH”*** defines the communication data width. This flexible model is used to generate an executable SystemC Model.

The *Flexible Algorithm/Architecture Model for MPEG4* is made of a set of *Modules* interconnected through *MPI-SystemC HLPPM* (Fig.4). The *MPI_SystemC HLPPM* is a runtime execution environment for message passing communication using the subset of MPI primitives presented in Fig.6. It’s similar to MPICH [14] (supports the same MPI primitives) but with the possibility of including configurable timing annotations for the communication, using SystemC libraries. Fig.9 shows that the communication between 2 tasks are done using Communication Units (CU) (one CU for each task), which manages the MPI requests from the tasks, the communication with other CUs, and inserts the timing annotations. Since a CU can be connected to many other CUs, the MPI-SystemC HLPPM can support point to point and bus topologies.

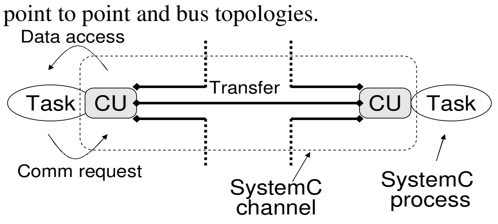

Fig.9 Task to Task communication using MPI-SystemC

Using the *Flexible Architecture/Algorithm Model for MPEG4*, many customized models can be macro-generated. This is done by expanding the flexible model with the desired algorithm / architecture configuration parameters, with the approach presented at [15].

Fig.10 shows an example of macro-generated SystemC Model with 2 SIMD subsystems (*MainDivX* and *VLC*), and the data dependencies between the tasks (the dotted arrows). This model is called *Executable SystemC Model of Combined Architecture/ Algorithm*. It is an un-timed model, and it captures both the architecture and the algorithm. Fig.11 (the WAITs will be explained later) shows the C/C++ code of the resulted *MainDivX1* task after the macro-expansion. It can be seen that all the algorithm/architecture parameters are now fixed. For different configuration parameters, different Executable SystemC Models are obtained. The key advantage of such model is its suitability for performances analysis, algorithm debug, syncronization debug, etc.

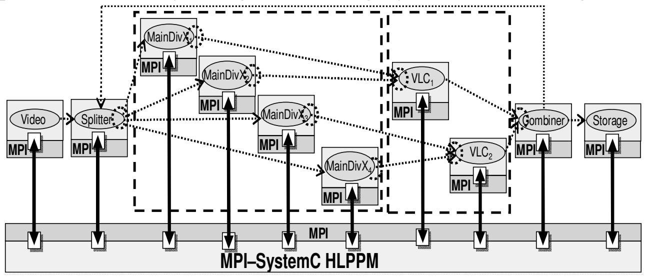

Fig.10 Executable SystemC Model of Combined Architecture/Algorithm

For performance estimations, a *Timed Executable SystemC Model* is used. This model is obtained by inserting time annotations for computations and communications, into the tasks code of the previously untimed Executable SystemC Model. The time annotations for computations are done by inserting WAIT calls to simulate the computations delays [14]. The time annotations for communications are embedded within the MPI-SystemC.

Fig.11 shows the code of the resulted Timed Model for the *MainDivX* task, which contains the time annotations for the computations, and the time annotations for the communications (integrated into MPI-SystemC HLPPM). The values for these delays are captured in tables and depend on the configurations chosen for the computations (i.e. CPU model, CPU clock frequencies) and communication primitives (i.e. data width, message sizes, latencies). This *Timed Model* allows performance estimations at High-Level for different algorithm/architecture configurations.

```
//----------------- MainDivX1 task ------------------------------------------
void MainDivX1_MAIN(*image_memory1,height1, length1, top_border1,
                        left_border1, bottom_border1, right_border1,&result)
{
  //initialization of computations
  MainDivX1_INIT (&image_memory1, height1, length1);
  WAIT(13.224);


  //data_receive_communication from the Spliter
  MPI_Recv(this,&image_memory1,sizeof(image_memory1),
             32,SPLITTER_ID,22,MPI_COMM_WORLD);

  //calls the function with flexible computations
 MainDivX1_COMPUTE (&image_memory1,height1, length1,
                         top_border1, left_border1,
                         bottom_border1, right_border1,&result);
  WAIT(2.312.564);

   //send_result_communication to the VLC
   MPI_BSend(this,&result,sizeof(result),32,
                 VLC[0]_ID,22, MPI_COMM_WORLD);
}
```

Fig.11 Timed Model for the MainDivX task

IV. High-level architecture exploration flow for MPEG4

This section describes in details the high-level architecture exploration flow (Fig.12) used for the MPEG4 video encoder.

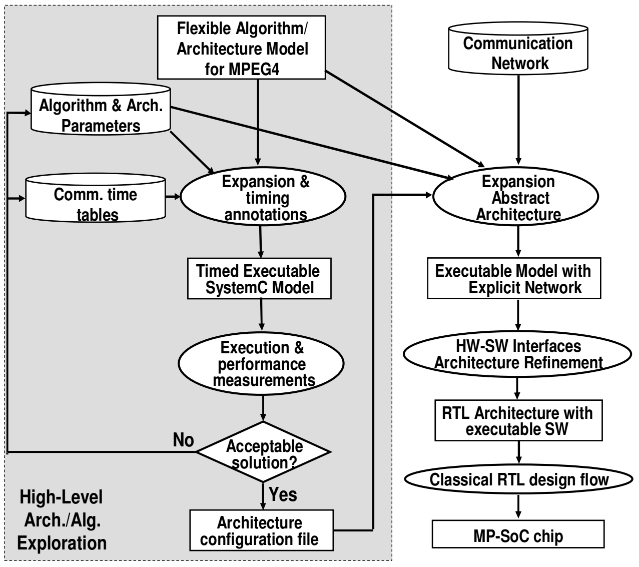
Flexible Algorithm/ Architecture Model for MPEG4
Communication Network
Algorithm & Arch. Parameters
Comm. time tables
Expansion & timing annotations
Expansion Abstract Architecture
Timed Executable SystemC Model
Executable Model with Explicit Network
Execution & performance measurements
HW-SW Interfaces Architecture Refinement
RTL Architecture with executable SW
No
Acceptable solution?
Yes
Classical RTL design flow
High-Level Arch./Alg. Exploration
Architecture configuration file
MP-SoC chip

Fig.12 Detailed representation of the design flow

This flow is composed of 3 major phases: (1) obtaining the *Timed Executable SystemC Model*; (2) performance estimation and reconfiguration; (3) building the final RTL architecture. Only the first 2 phases are part of the high-level architecture exploration and will be detailed. Presenting the phase of building the final RTL architecture is outside the scope of this paper [21].

A. Obtaining the Timed Executable SystemC Model

The *Timed Executable Model* is obtained in two steps. First the *Flexible Algorithm/Architecture Model for MPEG4* is macro-expanded to obtain the *Executable SystemC Model of Combined Algorithm/Architecture*. This initial model is used to compute the delays. Afterwards, the delays are inserted in the executable model.

Delays are obtained using a classical approach consisting of executing the code on an Instruction Set Simulator (ISS) of the targeted CPU. This gives approximate number of clock cycles required by the different tasks, independently of the communications. The obtained times aren’t 100% accurate, because the scheduling effect isn’t captured with this approach. However, the experiments show that the precision is enough for our architecture exploration, as will be shown later in this paper.

Communication times are given for different communication configurations (message size, data width, protocol, transfer latencies). In this work, these times are given as parameterized delay functions associated to each MPI primitive. The execution of each primitive is broken into 3 steps: initialization (initial synchronizations), transfer (for each data) and close (communication release). An execution time is associated to each of these steps, allowing a detailed viewing/analyzing of the communication behavior.

B. Performance estimations and architecture exploration

By compiling and executing the *Timed Executable SystemC Model*, performances can be measured using the function *sc_simulation_time()* after encoding every frame. The execution of this timed model gives an estimation of performances. The obtained performances can be represented using performances diagrams (graphic tables), and they include the time annotated computations and communications running together. Also, in the same graphic can be displayed for comparison the performances measured for multiple different algorithm/architecture configurations, to help the designer to take the next decisions. Fig.13 gives the estimated performance for the execution of MPEG4 for 25 frames of QCIF (176x144) video resolution movie, using 1,2,4,8,16 and 32 CPUs ARM7[17] at 60MHz for the MainDivX tasks and 1 CPU for VLC.

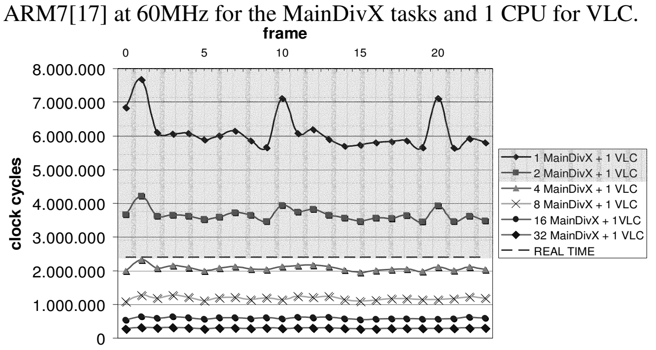

Fig.13 Performance estimated for QCIF, using ARM7, 60MHz

As benchmark movie we used one second (25 frames) of „snow-show" movie (similar to what the TV receivers show when there is no signal on the antenna). This represents the worst case scenario for the MPEG4 application. Consequently, is assured the real-time encoding for any other input cases. Also, the used search area for the Motion Estimation is 16x16. The reason is that previous research experiments showed that for QCIF (176x144) and CIF (352x288) resolutions, the full area search can be discarded, because the compression gain doesn't pay for the performance loss. However, this isn't true for higher video resolutions.

Fig.14 shows the estimated performances using ARM946E-S, 4kI$, 4kD$ CPUs at 60MHhz [17]. In order to achieve real-time, maximum 2.400.000 cycles are allowed for compressing 1 frame. From this figures, we can determine that minimum 5 ARM7 or 2 ARM946E-S processors are required to achieve real-time.

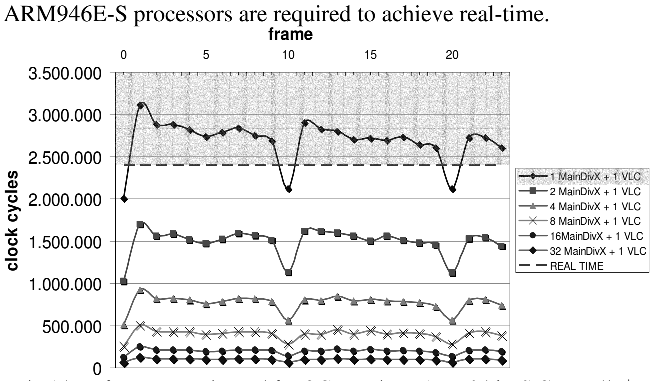

Fig.14 Performance estimated for QCIF, using ARM946E-S CPUs, 4kI$, 4kD$, 60 MHz

Different curves of these simulations are obtained doing a new macro-expansion of the initial model with different parameters. Besides number and types of CPU, several other parameters may be explored. For example, the communication may be explored via message size, data width, protocols and latencies.

C. Validation of the High-Level simulation results

The architecture exploration allows fixing a set of parameters that will define the number of required CPUs, models of CPUs, communication protocols, message sizes, maximum latencies, etc. These parameters will be followed during the architecture implementation. Fig.15 shows an example of obtained configurations.

```
CPUs Number  → 5 (4 MainDivX + 1 VLC)
IPs Number   → 2
Splitter     → IP
MainDivX1    → CPU (ARM7)
MainDivX1    → 60MHz
MainDivX1    → no cache
... the same for the other 3 MainDivX
VLC1         → CPU (ARM7)
VLC1         → 60MHz
VLC1         → no cache
Combiner     → IP
Splitter-MainDivX1 send protocol      → Blocking
MainDivX1-Splitter recv. protocol     → Non-Blocking
Splitter-MainDivX1 burst size         → 128 bytes
Splitter-MainDivX1 data_width         → 32 bits
Splitter-MainDivX1 init_latency       → 2 cycles
Splitter-MainDivX1 data_latency       → 3 cycles
MainDivX1-VLC1 send protocol          → Blocking (FIFO 810bytes)
VLC1-MainDivX1 recv. protocol         → Blocking
VLC1 recv. arbitration                → AnySource
MainDivX1-VLC1 burst size             → 810 bytes
MainDivX1-VLC1 data_width             → 32 bits
.... similar for the other modules
```

Fig.15 Example of architecture configuration file

V. Experiments and results analysis

This section presents the experiments results obtained for the architecture exploration of the MPEG4 application for QCIF (176x144) and CIF (352x288) video resolution at 25 frames/sec, using ARM7 and ARM946E-S processors running at 60 MHz.

For QCIF video resolution using only ARM7 processors running at 60MHz, the resulted architecture required 5 processors: 4 processors for 4 *MainDivX* tasks, and 1 processor for 1 *VLC* task. Simulation for 1, 2, 4, 8, 16 and 32 CPUs for *MainDivX* and 1 for VLC was shown in Fig.13. The obtained architecture configurations are shown in Fig.15.

For CIF (352x288) video resolution, the same experiments were conducted. In case of using only ARM7 processors, the architecture required 23 processors: 20 for *MainDivX* task and 3 for *VLC* task. Initially, 16 processors were sufficient for the *MainDivX* tasks, but the communication degradation made this impossible. So it was opted for more processors, instead of choosing a very "super" communication. Fig.16 shows the performance diagram using ARM7 CPUs at 60 MHz.

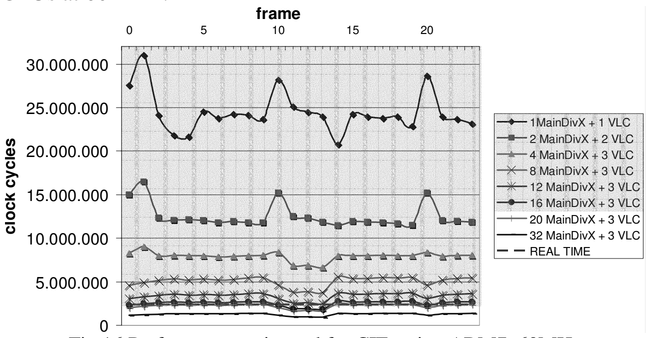

Fig.16 Performance estimated for CIF, using ARM7, 60MHz

When ARM946E-S processors were chosen, things got simpler because of the higher provided computation power. Fig.17 shows the obtained performance diagram for CIF video resolution, using ARM946E-S processors. Fig.17 shows that 10 ARM946E-S processors at 60 MHz were required to obtain a real-time functionality: 8 for *MainDivX* tasks and 2 for *VLC* tasks.

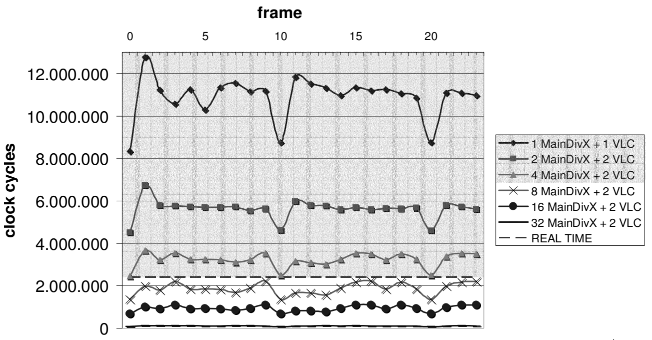

Fig.17 Performance estimated for CIF, using ARM946E-S CPUs, 4kI$, 4kD$ at 60 MHz

For higher resolutions of MPEG4, ARM7 and ARM9 are not enough. Additionally the amount of embedded memory gets higher than what the current technology may allow. For memory, an off chip memory may be required which might change the required interconnect. For computations, powerful DSP or VLIW processors are needed, or HW instructions [18][19].

To validate the precision of the high-level architecture exploration, an RTL architecture was built more or less manually, using one of the architecture configurations obtained during the high-level architecture exploration. Fig.18 shows that the precision of the performance estimations, obtained during the high-level architecture exploration, are very close with the one measured at RTL level. The communication infrastructure used in case of the RTL architecture, is a highly performant and customizable data transfer architecture [20] that can be easily configured with the communication parameters obtained by architecture explorations.

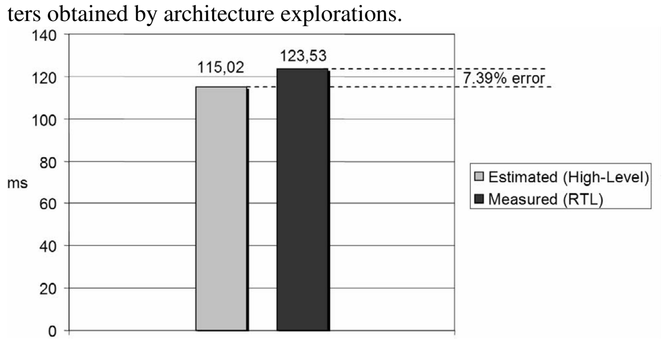

Fig.18 Estimated vs. Measured performance precision
[QCIF, 1 frame, 2 ARM7 CPUs (1 MainDivX + 1 VLC), 60MHz]

To compress 1 frame at QCIF resolution, using 2 ARM7 CPUs (1 *MainDivX* + 1 *VLC*) running at 60MHz, the high-level estimations predicted that 115.02 ms are required (in Fig.13, 6.82 million cycles is equivalent with 115.02 ms). The performance measured for the obtained RTL architecture proved that 123.53 ms were required to compress 1 frame. The 7.39% precision error comes from the impossibility to capture with our proposed high-level estimations, the performance degradations of the:

- OS (scheduling, service calls latencies induced by the API calls)
- Interconnect between the CPU buses and communication infrastructure (the conflicts for local bus grant between the CPUs and the Network Interfaces).
- HW/SW Wrappers

By using the proposed high-level architecture exploration, different and already validated architecture configurations were explored very quickly, even for a big number of CPUs and complex communications. This process dramatically shortened the time required to do the architecture exploration. As an example, in case of 25 frames of QCIF resolution video and using ARM7 processors running at 60MHz, approximately 15 minutes were required to generate the Timed Model. The simulation for 25 frames took approximately 2 minutes. Exploring one architecture solution takes less than one hour. This is the time required just to simulate one frame at RTL level. Approximately 25 hours were required to simulate 25 frames using the RTL model. So, the high-level performance estimations error of less than 10% compared with the low-level performance measurements is more then acceptable, considering the gain of design time. In these experiments we’ve used a Pentium4, 3GHz, 1Gbytes RAM, using Linux Mandrake 9.2.

Currently, in order to adapt this approach to other applications, the Initial Specifications Model must be adapted manually. Also, if the communication network is changed (uses different topologies with the ones already supported), the MPI-SystemC HLPPM model needs to be adapted to the new communication constrains. Automating these tasks or finding a method to reduce the effort needed for applying the proposed design paradigm to different applications are, in our opinion, open research subjects, and we have chosen to leave this point for future works.

VI. Conclusions

The architecture exploration of the MPEG4 video encoder into MP-SoC raises many challenges, because its complexity in term of computation, communications and memory requirements. Architecture exploration at low-level is a very long time consuming process. This paper proposed the use of a high-level architecture exploration method, since early stages of the design flow. The design effort gain is more important when lots of processors and complex communication networks are used, as showed by our case study. Because the architecture validation is done at High-Level, finding the optimal architecture configurations becomes possible in a much shorter time, compared with the validation at Low-Level. The proposed approach was successfully used during the architecture exploration of MPEG4 video encoders. The time required to explore different architectures was reduced from days to approximately 1 hour. This method can be extended for different other application, by adapting the Flexible Algorithm/Architecture Model for this new application.

References

[1]AV140 Video Recorder, http://www.archos.com/products/prw_500431.html

[2]eXpressDSP compliant MPEG4 Simple Profile Video Encoder for TI TMS320C64x DSPs, http://focus.ti.com/catalog/docs/thirdpartysoftwarefolder.tsp?softwareId=193

[3]M.W.Youssef et al, ”*Debugging HW/SW Interface for Multiprocessor SoC: Video Encoder System Design Case*”, 41st DAC, San Diego, CA, June 2004

[4]I. Richardson, “*H.264 and MPEG-4 Video compression*”, white paper

[5]Pieter van der Wolf et al, „*Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach*”, CODES-ISSS, Stockholm, Sweden, September 2004

[6]Satya Kiran M.N.V et al, “*A Complexity Effective Communication Model for Behavioral Modeling of Signal Processing Applications*”, 40th DAC, Anaheim, CA, June 2003

[7]L. Formaggio et al, “A *Timing-Accurate HW/SW Co-Simulation of an ISS with SystemC*”, CODES-ISSS 2004, Stockholm, Sweden, September 2004

[8] Xtensa LX Xplorer 1.0.1,www.tensilica.com

[9]V. Bhaskaran et al, “*Image and Video Compression Standards: Algorithms and Architecture*”, Norwell,MA:Kluwer 1995

[10]Y. He et al: “*A Software Based MPEG-4 Video Encoder Using Parallel Processing*”, IEEE Trans. on Circuits and Systems for Video, Nov. 1998

[11]K-K. Leung et al, “*Parallelization Methodology for Video Coding – An Implementation on the TMS320C80*”, IEEE Transaction on Circuits and Systems for Video Technology, Dec.2000

[12]M.Raulet et al, “*Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures*”, in proceedings of IEEE Workshop on Signal Processing Systems 2003, Seoul, Korea, Aug.2003

[13]C-C. Chiang, ”*High-Level Heterogenous Distributed Parallel Programming*”, Proceedings of the 2004 International Symposium on Information and Communication Tech., Las Vegas, Nevada, June 2004.

[14]MPICH – A Portable MPI Implementation, http://www-unix.mcs.anl.gov/mpi/mpich/

[15]M4 - a GNU implementation of the UNIX macro processor, http://www.seindal.dk/rene/gnu/whatis.htm

[16]Luciano Lavagno et al, “*Specification, Modeling and Design Tools for System-on-Chip*”, ASP-DAC’02/VLSI Design, Jan. 2002

[17]ARM7 & ARM946E-S, http://www.arm.com/products/CPUs/index.html

[18]Pierre Paulin et al , “*Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management*”, Best paper, ISSS/CODES 2004, September 2004, Stockholm, Sweden,

[19]Tensilica XTENSA-LX, http://www.tensilica.com

[20]S.-Il Han et al "*An Efficient Scalable and Flexible Data Transfer Architecture for Multiprocessor SoC with Massive Distributed Memory*”, DAC'04, San Diego, USA, June 2004

[21]A. A. Jerraya, W. Wolf, ”*Multiprocessor Systems-on-Chips*”, Morgan Kaufmann Publishers, ISBN 0-12-385251-X, September 2004