14th European Signal Processing Conference (EUSIPCO 2006), Florence, Italy, September 4-8, 2006, copyright by EURASIP

USING RTOS IN THE AAA METHODOLOGY AUTOMATIC EXECUTIVE GENERATION

Ghislain Roquier, Mickaël Raulet, Jean-François Nezan, Olivier Déforges

IETR/Image group Lab
CNRS UMR 6164/INSA Rennes
20, avenue des Buttes de Coësmes
35043 RENNES Cedex, France
groquier@ens.insa-rennes.fr {mraulet, jnezan, odeforge}@insa-rennes.fr

ABSTRACT

Future generations of mobile phones, including advanced video and digital communication layers, represent a great challenge in terms of real-time embedded systems. Programmable multicomponent architectures may provide suitable target solutions combining flexibility and computation power. The aim of our work is to develop a fast and automatic prototyping methodology dedicated to signal processing application implementation on parallel heterogeneous architectures, two major features required by future systems. This paper aims to present the Algorithm Architecture Adequation methodology from both the description of application and multicomponent architecture to the dedicated real-time distributed executives. Then a comparison is done between executives which use or not a resident Real-Time Operating System. This work is finally illustrated by the execution of a multimedia application based on a scalable LAR video codec.

1. INTRODUCTION

New embedded multimedia systems require more and more computation power. They are increasingly complex to design and need a time-to-market always shorter. Computation limits of systems (i.e. video processing, telecommunication physical layer) are often overcome thanks to specific dedicated circuits. Nevertheless, this solution is not compatible with short time designs and future capacity adjustments. An alternative can be provided by software components (DSP, ARM) and hardware components (FPGA) since they are reusable and programmable. The parallel and heterogeneous aspects of multicomponent architectures raise new problems in terms of application distribution. A suitable design process solution consists in using a rapid prototyping methodology. The aim is then to go from a high level description of the application to its real-time implementation on a target architecture as automatically as possible. Moreover, Data Flow Graphs (DFG) have proven to be an efficient representation model to describe an algorithm applications. To this end, the Algorithm Architecture Adequation (AAA) methodology is presented. It is a rapid prototyping methodology, suitable for transformation-oriented systems and heterogeneous multicomponent architectures. AAA methodology aims to automatically generate dedicated real-time distributed executives from both application and target architecture description models. AAA methodology is well fitted for deterministic algorithms. Indeed, all the executives are scheduled off-line by this methodology and so they are suitable for deterministic algorithms such as image processing algorithms.

The use of a RTOS is not necessary thanks to the off-line scheduling. On-line scheduling implies more available data and program on components and is well-suited when the application behaviour can not be predicted [1]. The use of an RTOS is no longer essential when the application behaviour is known before the execution (for instance the image algorithm we develop). Off-line scheduling releases ressources which are often too limited in embedded systems. Nevertheless, the comparison between those two approaches on the same application has to be done in order to know the impact on the allocated memory, the effect on the inter-component communications and the real-time behaviour of the algorithm. The paper is organized as follows : section 2 introduces the AAA methodology. The executive specifications and the use of RTOS according to the AAA methodology are described in section 3. Application implementation of the LAR codec and comparison of the results are explained in section 4. Finally conclusions are given in section 5.

2. AAA FAST PROTOTYPING METHODOLOGY

The aim of the AAA methodology is to find the best matching between an algorithm specifying the application to perform and a multicomponent architecture. Besides, real-time and embedding constraints must be satisfied. AAA methodology is based on graph theory to model the software application and the hardware architecture. Both the software and the hardware are described by distinct graphs. AAA methodology transforms those two graphs with graph transformations in order to find an optimized implementation.

2.1 Algorithm & architecture models

The application algorithm is modelled by a data flow graph which is an oriented hyper-graph. Each vertex corresponds to an operation of the algorithm and each edge represents a data transfert between operations. DFG only sets a partially order on the execution of operations thus two operations which are not in data-dependence relation may be executed in any order, particularly simultanously by two processors. Thus, DFG shows the *potential parallelism* of an algorithm [2].

In AAA methodology, so as to be precise in the model description but not too much complex (i.e hardware level) for implementation, finite state machine (FSM) is defined like the atomic component of an architecture [2]. A processor or a specific circuit is then a composition of FSMs. In this way, multicomponent architecture may be represented by a network of FSMs interconnected with communication media (bus, shared memories . . .). The architecture may be modelled by non-oriented hypergraph where each vertex is a processor and each hyper-edge represents a communication media. In this model, a processor is composed by one operator and as many communicators as connected media. An operator executes operations which are a part of the algorithm and a communicator executes a communication operation when a data transfert is required. Operator and communicator are connected together by a shared memory inside a

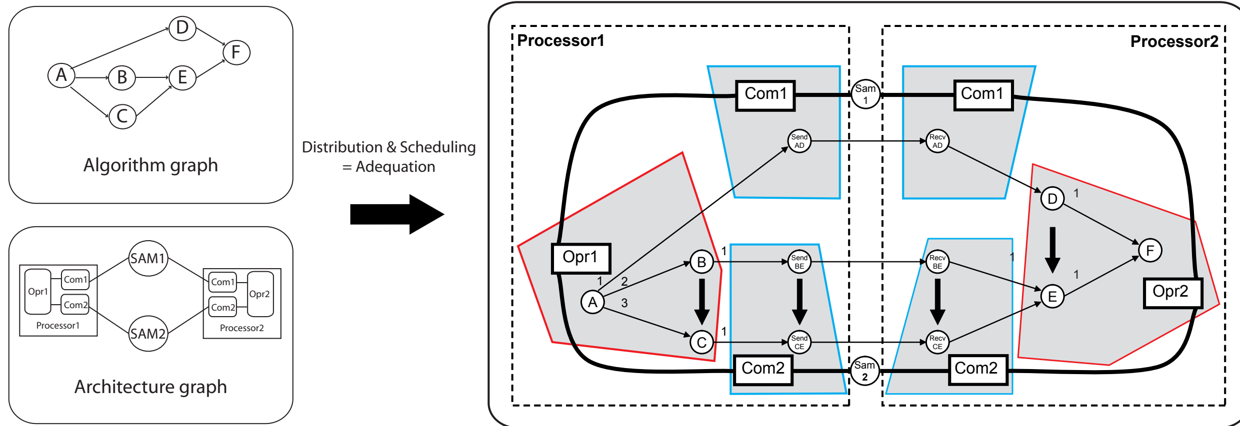

Fig. 1 – Distribution & scheduling of an algorithm onto an architecture

same processor.

Figure 1 represents an architecture graph which is composed of two processors and two media. Each processors are constituted by a single operator and two communicators. Architecture graph allows to exhibit the *available parallelism* useful during the implementation.

2.2 Graph transformation

The implementation graph is obtained by transforming the algorithm graph according to the architecture graph. This transformation corresponds to a distribution and a scheduling of the algorithm graph. Distribution, also called partitioning, allocate spatially parts of the algorithm onto components of the architecture. It consists in realizing a partition of the algorithm graph in different sub-graphs which describe the operations to execute for each operators. Scheduling allocates temporally operations onto components. It consists in executing sequentially all the operations allocated to an operator. The execution order has to take into account of the precedence required by the data-dependence between operations, otherwise total order of execution is automatically specified by adding precedence edge between operations without data-dependence relations during the implementation. These transformations are depicted in Figure 1 where the grayed areas represent operators and communicators. Operations of the algorithm are distributed on operators and communication operations are distributed on communicators. Scheduling when operations have not data-dependence is represented by the bold arrows in order to avoid deadlocks (e.g B must be computed before C). Otherwise, data-dependences must be respected (e.g A before B).

Efficient implementation graph is obtained by an optimization which realize simultanously distribution and scheduling. There is a large number of possible implementations. The optimization problem aims to select the most efficient one between them (real-time constraints, architecture ressources...). Moreover, the problem of distribution and scheduling in case of multicomponent is known to be NP-hard (an exhaustive research on all the possible fulfillments is inconceivable), this is why heuristics are used to match the best approximation of the optimal solution. This heuristic attempts to minimize the total execution time of the algorithm running on the multicomponent architecture. The definition of this greedy heuristic is described in [3].

2.3 Executive generation

As soon as an optimized implementation is determined, an executive may be automatically generated for each operator. First of all, repetitions and synchronizations must be added into the optimized implementation graph. Indeed, reactive applications we want to implement are iterative by nature whereas a DFG does not exhibit any their repetitive feature. In this way, repetitions have to be inserted to each sequence of operations and communications. Likewise scheduling the algorithm graph does not show synchronizations between different sequencers of a processor. Since operators and communicators are independent, synchronizations between sequencers are necessary when a communication is required. To this end, semaphores are inserted in the implementation graph for each synchronization in order to guarantee precedence between computation and communication operations on the same processor [4]. Indeed, inter-processor synchronizations (more precisely, inter-communicators) are necessary for sending and receiving data. A more detailed description is done in [2]. The following Petri network (Fig. 2) shows the intra-processor synchronizations of the different sequencers for the figure 1 first processor (it is the same for the second processor except *send* are replaced by *receive*) where P and V [1] respectively waits for a semaphore until a resource is available and releases semaphore after the process has finished to use it. Semaphore parameters define respectively the operation to synchronize and its path to the communicator (e.g A_1), if the buffer is full (1) or not (0) and the medium which transfer the data.

For instance, operation A is computing and writing in the memory buffer AD, a semaphore $\{A_1, 1, SAM1\}$ is sent to the communicator *Com1* which is waiting for it (assume *Opr1* execution slower than *Com1*) and has now access to AD, communication operation *send(AD)* sends its contents to *processor2* then it sends a semaphore $\{A_1, 0, SAM1\}$ to *OPR1* which have now access to the buffer. Since communication sequence *Com1* is faster, *Com1* must wait for the buffer AD is full to access it. It should be noted that synchronizations are managed automatically in AAA.

Then, this executive graph is transformed into macro-executives which are as many as processors of the architecture. The generic executive is composed of a list of macro-instruction which specifies memory allocation, communication sequence(s) and computation sequences. These macro-executives are generics i.e. they are independents from a specific programming language. It authorises to transform later in the appropriate language specified by the different target operators (C or assembler for DSP, VHDL for FPGA...). This is the purpose of section 3.

[1] ***P****robeer* and ***V****erhoog* mean *decrease* and *increase* in Dutch

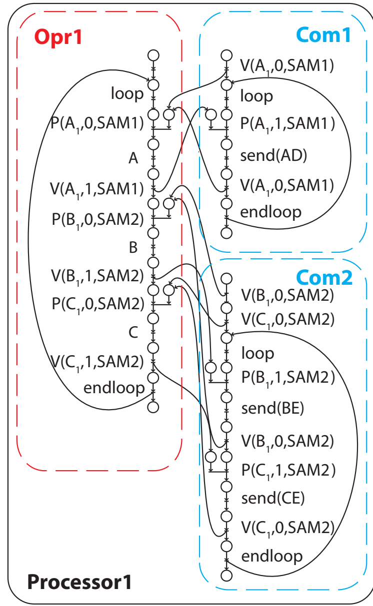

Fig. 2 – Petri net of the execution graph

2.4 SynDEx

SynDEx[2] is a system level CAD software principally designed at INRIA Rocquencourt Research Unit (France) in development with our image processing lab. This tool supports the AAA methodology for rapid protyping and for the implementation optimization of distributed real-time embedded applications onto "multicomponent" architectures. SynDEx allows to generate as many executives as processors of the multicomponent architecture respecting the methodology aforementioned.

3. GENERIC EXECUTIVE SPECIFICATION

Macro-executives are generated following the AAA methodology. We need to transform these different generic executives in order to have compilable executives which may be downloaded in the target processor.

3.1 Translation

Each generic executive is translated in a compilable executive by a macro-processor. The macro-processor transforms this list of macro-instructions into compilable code for a specific processor target. It replaces macro-instructions by their definition given in the corresponding library (also called kernel) which is dependent on a processor target and/or a communication medium. The translation is done thanks to these libraries. One of them is generic and do not depend on the algorithm. It supports the architecture specification such as memory allocations, sequence synchronizations and also inter-operator transfers. The other kind of libraries permits to describe the algorithm specification such as function definitions or function-calls. Since the macro-executive is generic, there is a large number of possible downloadable source code, thus users have to specify the most appropriate translations as possible. The free software Gnu-m4 is the macro-processor we use to transform the macro-executive into compilable source. Notice, in both cases, real-time distributed executives are statics, the deterministic behaviour of the real-time execution is guaranteed off-line by the distribution scheduling heuristic.

[2] **Syn**chronized **D**istributed **Ex**ecutive

3.2 Real-Time Operating Systems in AAA context

Our approach in this article is to compare two different translations. One the one hand, we have realized a "classical" translation in the sense of AAA [5]. The "classical" approach consist in straightly tranlating the optimized generic executive in the suitable programming language. On the other hand, we have also realized a translation which allows to manage a RTOS. This approach consist in translating the generic executive in order to configure a resident RTOS which is able to manage the different sequences as well as synchronizations between them. To this end, we developed new libraries to specify the translation of macro-intructions. Figure 3 describe the two approaches from software description to hardware implementation.

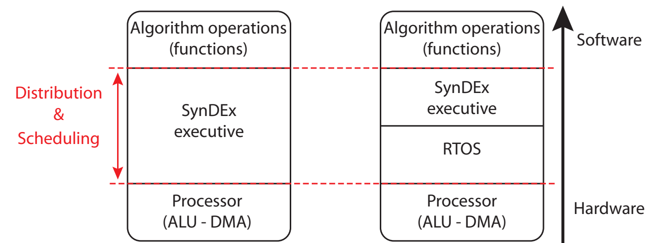

Fig. 3 – Software to hardware : two approaches

A RTOS is an operating system that is developed for real-time applications. It allows to manage multi-threaded processes. In this approach, computation and the communication AAA sequences are several threads synchronized by the use of semaphores. Synchronizations are now managed by the RTOS i.e we do not put "manually" a value (0 or 1) for each semaphore to lock or unlock a shared memory unlike it is done in the other approach. Notice that we use the RTOS only for allowing the creation and the scheduling of threads and the management of synchronizations in order to have a multi-threaded process. Yet we do not use it for managing software or hardware interrupts. Unfortunately, integrating the RTOS on the multicomponent target must have a cost. Indeed the use of a resident RTOS necessarily increase the overheads of the executives.

4. RELATED WORKS

4.1 RTOS overview

A great deal of RTOS exists for processors. Their primitives are often specific to particular kind of processors. This is why more generic RTOS primitives may be well-suited for a faster implementation on different processor types. The embedded Linux seems to become the RTOS more used in embedded systems. Indeed, embedded Linux has the advantage of being modelled with POSIX standard so different embedded Linux are generic to program contrary to the RTOS developped by processor manufacturers. Moreover embedded Linux may be embedded in most of existing processors. For TI DSP, some embedded Linux RTOS exists such as Media-Linux OS [6], Lightweight OS [6]. For the moment, the Texas Instrument RTOS called DSP-BIOS is used for embedded applications we develop. DSP-BIOS manages each process (one process per DSP). It has been configured by the compilable executives only for managing synchronizations between different threads and for scheduling them. Notice, DSP-BIOS is usable only with TI DSP and so its primitives are not generic.

4.2 Platform targets

Several hardware providers such as Pentek, Sundance or Vitec MultiMedia develop multicomponent hardware architectures and some of them have been validated in our lab to support the AAA methodology [5]. A Sundance hardware platform is used until now to implement executives generated with a RTOS. This platform is made up of a host PC with motherboard which supports several TIMs (Texas Instrument Module) interconnected by different kinds of media. Our platform is constituted by a SMT361 TIM and a SMT319 TIM. The first one is a module with a TI C6416 DSP which is well-suited for image processing and the second one is a framegrabber, which is an image digital-analogic converter. The TIM includes a C6414 connected via two FIFO to non programmable devices (the BT829 encode PAL stream to YUV stream and the BT864 decode YUV stream to PAL stream). The architecture platform is shown in the figure 6 with two SMT361 TIMs and a SMT319 TIM.

4.3 Preliminary results

In order to know the difference between on-line and off-line sheduling, we made tests on a simple application (sendings and receptions of datas) implemented onto two DSPs in order to have many communications. The results are shown in figures 4 and 5. The first graph shows the time necessary to carry out totally the execution compared to the data size. A comparison is done of the result between the "classical" approaches and the one which manage the RTOS in the figure 5. Processes with the RTOS are always slower than without it. However, the larger the data size is, the more the difference is reduced. So, the execution time do not seem to be excessively modified by the use of a RTOS especially with large data like in image processing. In addition to that compilable executives are definitely shorter and understandable compared to the other method. Synchronizations are made easier by using the RTOS. Thus the debugging during the algorithm functional checking is facilitated.

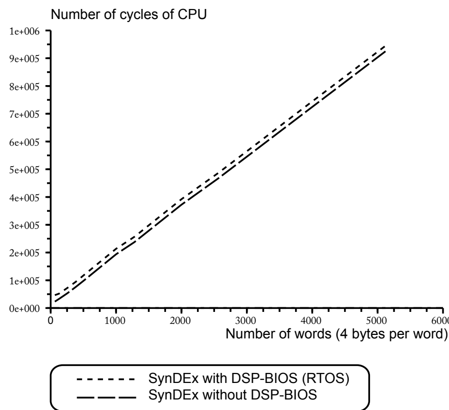

Fig. 4 – Comparison of execution times

4.4 LAR codec implementation

LAR[3] is a compression and a decompression image processing algorithm developed in our laboratory [7]. It is an efficient technique well-suited for image transmission. Its principle is to adapt the local resolution (pixel size) according to the luminance uniformity i.e typically a low resolution (block 8×8) when the luminance is uniform and a high resolution (block 2×2) when the activity is high. It is a scalable image

[3]**L**ocally **A**daptive **R**esolution

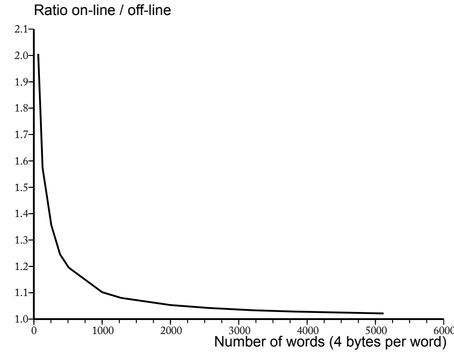

Fig. 5 – Ratio between execution times with and without RTOS

algorithm	no RTOS	RTOS
1	18.03 ms	18.05 ms
2	25.35 ms	25.45 ms
3	31.84 ms	32.07 ms

Tab. 1 – Execution times of LAR algorithms

and video codec from grayscale image compression to lossless colored video compression. Following the AAA methodology, an implementation is done of the LAR codec algorithm onto the architecture. Coder and decoder are respectively implemented on DSP1 and DSP2 of the architecture. The DFG of the LAR codec is described figure 7. Three different LAR scalable algorithms are implemented in order to compare the result when the complexity of the graph description increases [8].

Algorithm 1 : spatial (featured by block 2×2, 4×4 and 8×8) video codec for luminance.

Algorithm 2 : spatial video codec for chrominance adding to algorithm 1.

Algorithm 3 : spectral (featured by adding residual error) video codec only for block 2×2 luminance adding to algorithm 2.

The LAR codec results support those obtained previously subsection 4.3. Execution time is slightly slower with a RTOS process management. The coder is the slowest part of the algorithm (typically 4 times slower than the decoder). The real-time behaviour of LAR codec is thus given by the coder execution time since the total execution is pipelined by processors. Time obtained are shown in figure 1. The impact on the memory of DSP-BIOS is represented in figure 2. As envisaged, DSP-BIOS increase the memory used by processors (55 kBytes). However, the more the memory used is large, the more the impact is proportionally small. In algorithm 3, DSP-BIOS increases the memory used in coder and decoder DSPs respectively of 7% and 5%.

5. CONCLUSION AND PERSPECTIVE

This paper has presented the integration of a Real-Time Operating System in the AAA methodology executive gene-

algorithm	Coder DSP		Decoder DSP	
	no RTOS	RTOS	no RTOS	RTOS
1	874 kB	928 kB	899 kB	953 kB
2	747 kB	802 kB	658 kB	713 kB
3	642 kB	697 kB	528 kB	583 kB

Tab. 2 – Codec memory used

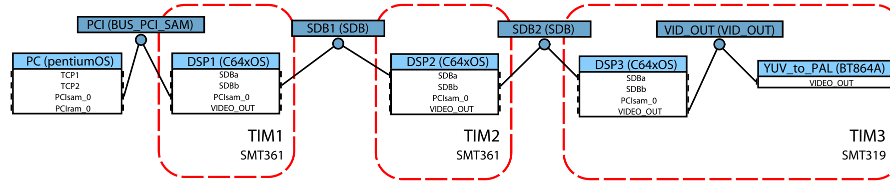

Fig. 6 – Platform target

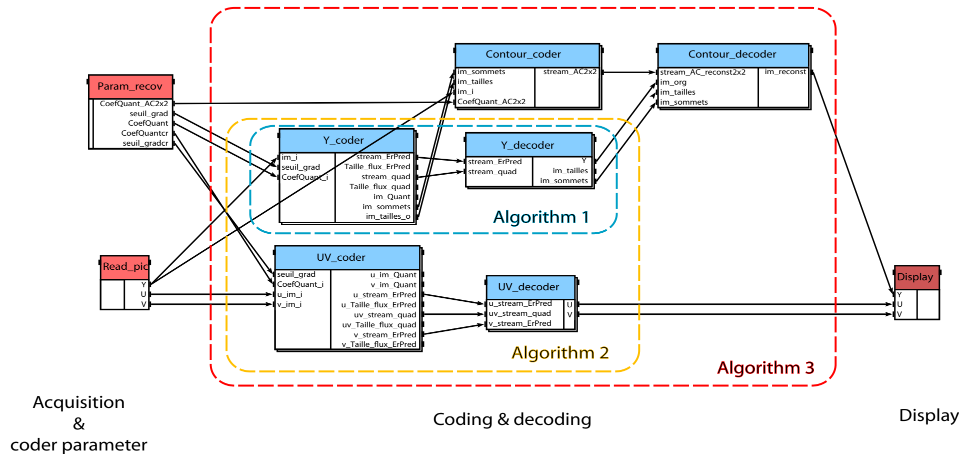

Fig. 7 – LAR DFG description

ration. The AAA methodology has been introduced in order to present the automatic generation of dedicated real-time distributed executives. The development of libraries during the execution specification has enabled to manage RTOS and to implement it in the PC-multi-DSP target. These libraries have been integrated in existing projects developed in our lab. These projects are free to download.

Though the RTOS has an impact on processor target such as execution time or allocated memory, we noted that the overcost is more slight when data size grown. So, using a RTOS seems to be almost as efficient as the "classical" approach for image processing algorithms where data are often large. Moreover, executives automatically generated including RTOS primitives are really simple leading to a better comprehension for users. One of the main objectives for RTOS developpers is to keep primitives compatible with new components. This point is important to keep our automatic code generation available in the future. The LAR codec implementation allowed us to verify the real-time behaviour of the codec in the case of the RTOS approach.

We are working on the integration of a more generic RTOS such as an embedded Linux in the AAA methodology executive generation. Embedded Linux will allow to develop faster multithreaded processes on different kind of processors and platform targets with the same RTOS executive specification. This work on embedded Linux according to the AAA methodology will allow the implemention of new image processing algorithms in development in our image-lab such as MPEG-4 AVC and MPEG-21 SVC standards.

REFERENCES

[1] F. Balarin and *al.* Scheduling for embedded real-time systems. *IEEE Design and Test of Computers*, 15(1) :71–82, January-March 1998.

[2] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, Mont Saint-Michel, France*, June 2003.

[3] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. In *proc. of IEEE CODES'99 7th Int. Workshop on Hardware/Software Co-Design, Rome, Italy*, May 1999.

[4] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages : NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[5] M. Raulet, F. Urban, J-F. Nezan, O. Déforges, and C. Moy. Syndex executive kernels for fast developments of applications over heterogeneous architectures. In *EUSIPCO'05, Antalaya, Turquey*, September 2005.

[6] J. Kretzschmar and R.Baumgartl. Lightweight rtai for dsps. In *1st Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Palma de Mallorca, Spain*, June 2005.

[7] O. Déforges and J. Ronsin. Region of interest coding for low bit-rate image transmission. *IEEE International Conference on Multimedia and Expos (ICME), New-York, USA*, August 2000.

[8] M. Raulet, F. Urban, M. Babel, O. Déforges, J-F. Nezan, and Y. Sorel. Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures. In *IEEE Workshop on Signal Processing Systems (SIPS'03), Seoul, Korea*, September 2003.