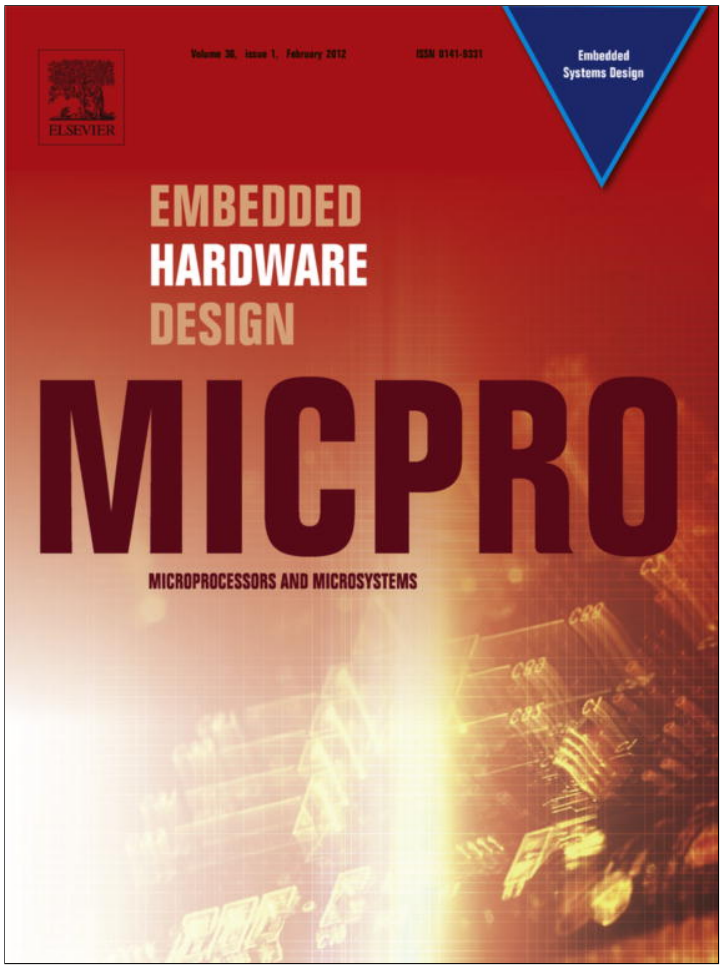

(This is a sample cover image for this issue. The actual cover is not yet available at this time.)

Microprocessors and Microsystems 36 (2012) 176–189

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A fast MPSoC virtual prototyping for intensive signal processing applications

Rabie Ben Atitallah [a,*], Éric Piel [b,1], Smail Niar [a,2], Philippe Marquet [c,3], Jean-Luc Dekeyser [c,4]

[a] Université de Valenciennes, LAMIH DIM-ISTV2, Le Mont Houy, 59313 Valenciennes Cedex 9, France
[b] Technische Universiteit Delft, Faculteit EWI, Mekelweg 4, HB 08.080, 2628 CD Delft, The Netherlands
[c] Université de Lille 1 – Batiment M3, 59655 Villeneuve d'Ascq Cedex, France

ARTICLE INFO

Article history:
Available online 26 July 2011

Keywords:
MPSoC
Model of computation
Model of execution
Simulation
Transaction level modeling
Virtual processor

ABSTRACT

Due to the growing computation rates of intensive signal processing applications, using Multiprocessor System on Chip (MPSoC) becomes an incontrovertible solution to meet the functional requirements. Today, Electronic System Level (ESL) design is considered a vital premise to overcome the design complexity intrinsic in the heterogeneity of these devices. However, the development of tools at the system level is in the face of extremely challenging requirements such as the rapid system prototyping, the accurate performance estimation, and the reliable design space exploration (DSE).

Focusing on the issue of ESL development tools, this paper describes an MPSoC environment design which targets the Multidimensional Intensive Signal Processing (MISP) application domain. Within this environment, we have defined first a generic execution model that supports any type of MPSoC. It can adapt to any parallel application and handle efficiently the scheduling and synchronizations at all the levels of granularity. Second, a new Virtual Processor (VP) based simulation technique is proposed for implementing the execution model. This proposal leverages the high-level specification of the system to provide a heterogeneous MPSoCs simulation without using an Instruction Set Simulator (ISS). VP-based simulation is implemented in SystemC at a timed transactional level allowing a good trade-off between high simulation speed and performance estimation accuracy. The usefulness and the effectiveness of our MPSoC environment is illustrated through two MISP applications executed on a typical MPSoC. Results show that our approach enables fast MPSoC virtual prototyping, data transfers and timing analysis, and reliable DSE for architectural optimizations.

1. Introduction

Nowadays, embedded signal processing applications such as the one used in networking, multimedia, and communications are becoming more and more complex. Two main aspects characterize these applications. The first involves the high complexity of the data structures, which generally represent multidimensional data arrays, while the second concerns the potential parallelism available in the application functionality. These two aspects lead to the Multidimensional Intensive Signal Processing (MISP) application domain. Hence, hardware designers are obliged to come up with new architecture definition for executing this field of applications. An inevitable solution to meet the performance goals consists in placing several processors in the same chip, thus creating MultiProcessor Systems-on-Chip (MPSoC). Today, MPSoCs are increasingly used to build complex integrated systems [1]. They must be designed with custom architectures to balance the implementation constraints between the application needs (i.e.: high computation rates and low power consumption) and the production cost.

MPSoCs have a huge architectural solution space which makes the Design Space Exploration (DSE) complex and most important challenge for designers. For instance, architectural parameters which can be explored include the processor type, the interconnection network topology, and the mapping of tasks (hardware or software) and data. In addition, a huge space of alternatives to implement and execute the systems is possible, which yields to a multitude of performance trade-offs in terms of execution time, power consumption, cost, etc. For MISP applications, this challenge is especially intensified by the high potential parallelism and the complex data distribution. The complexity lies mainly on the organization of the elementary tasks, which compose the application, and on the access patterns to their input and output data as parts of multidimensional arrays. These complex access patterns lead to difficulties to efficiently schedule the applications on MPSoCs. It is therefore important to define an appropriate MPSoC execution

* Corresponding author. Tel.: +33 (0)3 27 51 19 47; fax: +33 (0)3 27 51 19 40.
E-mail addresses: Rabie.Ben-Atitallah@lifl.fr (R. Ben Atitallah), e.a.b.piel@tudelft.nl (É. Piel), smail.niar@univ-valenciennes.fr (S. Niar), Philippe.Marquet@lifl.fr (P. Marquet), jean-luc.dekeyser@lifl.fr (J.-L. Dekeyser).
[1] Tel.: +31 15 278 6338.
[2] Tel.: +33 (0)3 27 51 19 48; fax: +33 (0)3 27 51 19 40.
[3] Tel.: +33 (0)3 59 57 78 05; fax: +33 (0)3 59 57 78 50.
[4] Tel.: +33 (0)3 59 57 78 04; fax: +33 (0)3 59 57 78 50.

0141-9331/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2011.06.001

model that makes profit from MISP application characteristics and also allows early system exploration.

The challenge of MPSoC DSE is tackled by several frameworks by means of the development of Electronic System Level (ESL) tools. The objective is to unify the hardware and software design and to offer a rapid system level prototyping. Based on the requirements like the timing accuracy and the simulation speed of the system, architects could select an appropriate level of abstraction to model the software executing on the processor. Fig. 1 shows the former used models [2] and also our proposed approach called the Virtual Processor (VP)-based simulation. In the past decade, commercial tools have succeeded to provide conventional RTL simulation environments for low level system prototype. This approach was very useful on one hand to the software developers for driver debug and integration. On other hand, it allows the hardware engineers to keep their traditional view of the system. However, the RTL tools cannot adequately support the complexity of future MPSoC since they are too slow for a meaningful execution of the software. In an attempt to reduce simulation time, a lot of research efforts have been put to evaluate the system using Cycle-Accurate (CA) simulators. They simulate the micro-architecture at the clock-cycle level and are by far the most common type of simulator used. At a higher abstraction level, an Instruction Set Simulator (ISS) sequentially executes the instructions and has no notion of concurrency of micro-architecture. The ISS description can be enhanced with timing annotations to approximate the execution time and obtaining a behavioural model. However, simulation speed with CA or behavioural ISS simulators are limited to few hundred thousands of simulated cycles per second. In addition to this challenge, as the architecture part must be closely adjusted to the application needs, frontiers between different domain experts (hardware, software, compilation, etc.) have to be broken. During the development process, the hardware/software interaction must be kept and the transition between the different design steps must be as smooth as possible.

In order to answer the design challenges of MPSoCs dedicated to MISP applications, a new approach is needed. In this paper, two contributions in the field of MPSoC ESL design and simulation tools are made. First, an efficient MPSoC execution model adapted for MISP applications is defined. It respects a repetitive Model of Computation (MoC) [3], which offers a very suitable way to express and manage the potential parallelism in the system. Second, a VP-based simulation technique is presented. It speeds up the time of the system verification with a good performance estimation accuracy. VP technique is implemented at a high-level simulation using the SystemC Transaction Level Modeling (TLM) 2.0 kit. It leverages the high-level system specification to provide a hardware/software co-simulation without using an ISS.

This paper is organized as follows. After Section 2 which presents the related works, Section 3 exposes an overview of the repetitive MoC on which our approach relies. In Section 4, an explicit execution model is defined in order to obtain a precise implementation of the MPSoC. Section 5 presents our technique for a high-level simulation. To evaluate our approach, experimental results are presented in Section 6.

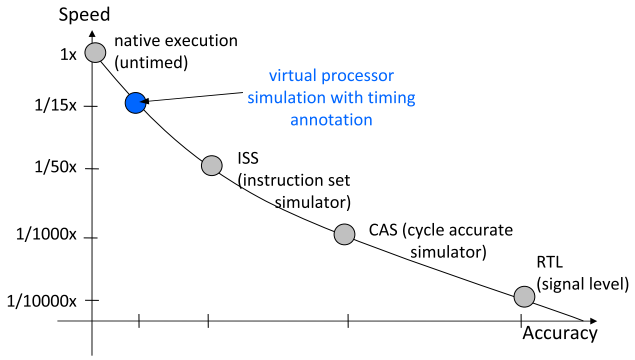

Fig. 1. Levels of the processing part modeling [2].

2. Related works

In an attempt to deal with the parallelism acquired from intensive signal processing applications, most approaches rely on a specific MoC. In general, such a MoC proposes a high-level formalism in order to exploit the parallelism at the task level in an easy and efficient way. Among widely used MoC, we can quote Khan Process Network (KPN) [4], Synchronous DataFlow (SDF) [5], multi-dimensional SDF (MDSDF) [6] and ArrayOL [3]. The main comparison criteria are the allowed data structures (mono-dimensional data flows or multidimensional arrays) and the expressiveness of the functions to access these data structures. In our work, the ArrayOL model is chosen. As a common point, all these languages permit a static scheduling in order to build efficient implementations. The expressiveness of the potential parallelism in signal processing applications is the vital aspect of ArrayOL model, and it hides most of the complexity of scheduling. Furthermore, its semantic is adopted in the standard MARTE UML profile [7].

For simulation, a MoC is not sufficient, it is necessary to define the precise order the tasks will be executed, the exact place the data will be stored, etc. This is defined by the *model of execution*. It specifies the behaviour with enough details so that the simulation at the various abstraction levels and the final implementation all correspond to the same execution order, bus access patterns, etc. Some works have been proposed to project ArrayOL to KPN [8] or to SDF [9], which indirectly leads to an execution model, via the works of Parks [10] (for KPN) or Buck [11] (for SDF). However these projections take into account only the application, and adapt more or less well to the underlying platform. Our work is inspired by these proposals, but ensures that this execution model follows the hardware/software mapping specified by the user. In addition, our execution model is especially adapted for MPSoC, focusing on the efficiency of the execution both in terms of execution time and memory usage.

For an efficient high performance SoC design, in addition to the specification of the application, the hardware architecture should also be taken into account. A huge space of alternatives to implement and execute the system is possible which yields to a multitude of performance trade-offs in terms of execution time, power consumption, cost, etc. It is therefore important to project the MoC onto an execution model where the hardware/software performance evaluation is possible. This challenge is set as the topic of several frameworks by means of a high-level simulation in order to reduce the evaluation time. SystemC [12] and SystemVerilog [13] are examples of hardware/software languages which aim to be used for system description at different abstraction levels. The last few years, Transaction Level Modeling (TLM) [14] has been embraced as a primary solution to make the system description easier and the simulation faster. This is enabled by the abstraction of inter-module communication into channel objects and the suppression of micro-architectural details. SystemC TLM 2.0 [12] offers different coding styles by providing a set of different interface types. The SoCLib [15] library provides an MPSoC simulation environment at timed TLM level using ISS. ReSP [16] and the Open Virtual Platform by Imperas Inc. [17] use the same level of simulation but also tackles the simulation speed problem by using advanced techniques for the software simulation such as OS simulation for the former and code morphing for the latter. However in our work, while the hardware is also simulated at the TLM level, we introduced the virtual processor concept to execute the application tasks in an abstract

way, allowing higher simulation speed-ups. In addition, heterogeneous architectures (i.e.: with several types of processing elements, such as an ARM9 and a DSP or a Microblaze and a hardware accelerator) become increasingly usual but the high level simulation tools tend to support only homogeneous architectures. In addition to the simulation of heterogeneous architectures, the advantage of our method is also to simplify the exploration of such architecture by abstracting a lot of micro-architectural details.

Often, hardware/software co-simulations use the lowest level of abstraction for the software: every machine instruction is decoded. Nevertheless, several approaches have been proposed to abstract the software description, in order to speed up the simulation. Some approaches rely on the notion of sampling [18,19]: the behaviour of the software (typically the memory accesses) is observed during one full execution, and then only a small representative number of observation samples are replayed later. Although this can highly reduce the simulation time while still giving accurate values for the non-functional properties (e.g.: execution time, energy consumption), the correct behaviour of the application cannot be verified, and no debugging can take place. In addition, on multiprocessor architecture, any modification on the architecture parallelism or re-organization of the software requires to repeat the consuming pre-processing phase to find the patterns. Another type of approach is to avoid the cost of the ISS by directly executing the software on the host processor. Benini et al. [20] proposed an implementation based on a debugger observing and controlling the software on the host. Gao et al. [21] extended this technique to allow accurate estimation of the timing by using *hybrid simulation*. It combines cache simulation and online trace-driven replay techniques to accurately predict performance of programmable elements in an MPSoC environment. To do so, the presented workflow puts together a specific ISS execution with native code execution into one simulation framework for achieving a better trade-off between high simulation speed and accuracy. Although it provides generally an error of less than 10% compared to an ISS, the speed-up is limited to 3–5 times. In addition to this, it requires the presence of an ISS, which might not be the case for early architectural exploration. In our approach, the application is simulated without using ISS. Software tasks, scheduler, and synchronization mechanisms are directly and explicitly executed as part of the simulation, instead of being a decoded sequence of instructions from the binary code of the software. In addition, a timing model is defined to approximate the execution time. However, the communication part is simulated at the transactional level. Our choices are influenced by the target application domain, which is mainly data-flow oriented. Honda et al. [22] demonstrated the usage of an additional software layer pretending to be the operating system to implement the connection between the hardware and the software. As this connection reduces the performance, in our contribution, we propose to avoid its usage by directly replacing the processor component by the software it would execute.

3. Repetitive MoC overview

The design of MPSoCs in our work specifically relies on the *repetitive* MoC of ArrayOL [3], which offers a very suitable way to express and manage the potential parallelism in the system. This MoC is accessed in our design environment via the MARTE (Modeling and Analysis of Real-time Embedded systems) standard profile [7]. MARTE allows to model both software and hardware of a system using UML. The hardware/software mapping can also be represented using the same repetitive formalism. With the repetitive MoC semantic applied to task and data parallelism, only the true data dependencies are expressed in order to specify the full parallelism of the application. Thus, any scheduling satisfying these dependencies will lead to the same result. Furthermore, this MoC has a single assignment formalism: no data element is ever written twice. Data can be read several times, though. The model can be hierarchical to allow descriptions at different granularity levels, permitting to handle the complexity of the applications.

For a better understanding of the main repetitive MoC concepts, we will consider the Downscaler algorithm, largely used in the TV signal processing field. It allows to scale high definition (HD) TV frames (1920×1080) down to a standard definition (720×480). The algorithm is composed successively of two main tasks: horizontal and vertical filters. The horizontal filter performs a scaling from the HD frame to the intermediate 720×1080 frame. This latter will be taken as an input for the vertical filter to generate a standard frame. In our MoC specification, the application is a set of tasks connected through ports as shown in Fig. 2. Tasks are considered as mathematical functions reading data from their input ports and writing data on their output ports.

In the Downscaler application, each of the two filters has a repetitive functionality. This repetition is characterized via a multidimensional repetition space called a shape which is noted as a column vector. For instance, for the horizontal filter task, we need to carry out the Hfilter sub-task in a repetitive way to cover all the HD frame blocks. The corresponding shape equals to {240, 1080, ∞} which signifies the Hfilter task is executed 240 times along the first dimension of the frame and 1080 times along the second dimension while the third represents the flow of frames (in time). All repetitions of this repeated task are independent: they can be scheduled in any order, even simultaneously.

Data exchanged between the tasks are arrays of elements. These arrays are multidimensional and are characterized by their shape which specifies the number of elements on each dimension. As an example in the Fig. 2, the Horizontal filter task has an output array of pixels with a corresponding shape equals to {720, 1080, ∞}. This implies that a flow of two dimensional arrays of size 720×1080 are exchanged between the Horizontal filter and the Vertical filter. The data parallelism of a task is specified in a repetition task. The hypothesis is that each instance of the repeated task operates with sub-arrays of the inputs and outputs of the repetition. For a given input or output, all the sub-array instances have the same shape, are composed of regularly spaced elements and are placed in the array in routine fashion. This hypothesis allows a compact representation of the repetition and is coherent with the MISP application domain which describes very regular algorithms. These sub-arrays are called *patterns*. The information needed to create these patterns is contained in a tiler, associated with each array. A tiler permits to build the patterns from an input array, or to store the patterns in an output array. To do so, different information (*origin*, *fitting* and *paving*) should be specified as shown in Fig. 2.

In MARTE, the concepts presented in this section have been extended to express the regularity and the parallelism in the whole embedded system. The MARTE profile allows to describe the hardware architecture in a structural way using the HW_Logical subpackage. Furthermore, this profile provides the Allocate concept as well as the Distribute concept specially crafted for repetitive structures. This latter concept gives a way to express regular distributions from an array of task (resp. data) to an array of processors or hardware accelerators (resp. memory units). This short overview of the repetitive MoC is given to help readers for a better understanding of the main choices in our proposal of the execution model. More details are presented in [23,24].

4. Execution model

The MoC we have presented specifies an accurate semantic for the application. However, for a given application and architecture, the system can still be implemented and executed in many different ways: whichever execution technique is used, the same

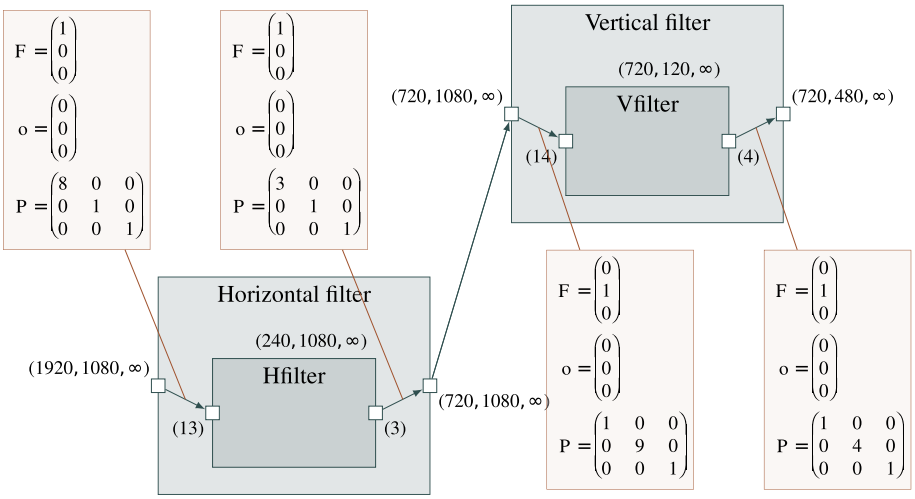

Fig. 2. The Downscaler example.

outputs will be generated, but with different non-functional properties such as the time to complete, the memory needed, or the power consumed. It is therefore important to define an efficient *execution model*. This model should be generic enough so that it can be used on any type of hardware architecture. Let us highlight that although the user does not choose, nor even parameterize the execution model, he/she can influence the implementation by controlling the models of the application, the architecture, and the related mapping. An execution model precisely defined is necessary for ensuring a realistic behaviour of the MPSoC even when simulating at high abstraction level where the software is not executed at a machine code level.

In the context of multiprocessor, the main aspects of the execution model concern the task scheduling, the synchronization of the tasks, and the way the data is laid in memory and accessed. In our work, we define an efficient execution model which is generic enough to support any type of MPSoC.

4.1. Task scheduling

The MoC, used by the user to describe the system, is very permissive in terms of ordering the task execution: whenever an instance of task has all its data dependencies cleared (i.e.: all its input arrays are available) it *can* be executed. The execution model has to define which task is being executed, for each processor, at every moment.

For performance reason, a critical aspect in MPSoCs, it is preferable to do static scheduling whenever all the conditions are known in advance. This avoids the little overhead that a dynamic scheduler brings. However, dynamic scheduling is necessary because not all the behavioural and timing details are present in the model of the system. The exact behaviour of the hardware cannot be predicted precisely enough to forecast the exact duration of each task (especially with respect to conflicts on the buses). Dynamic scheduling also permits to keep simple the code of a task which has to run on several processors. Otherwise, with static scheduling, a specialized version of the task would be needed for each processor. As a trade-off, we distinguish two levels in an application, as shown in the example of Fig. 3:

- The **higher level** corresponds to the tasks which have dependencies on tasks running on *other* processors. All the tasks higher in the hierarchy are also part of this level. This is represented by the green dotted set in the figure. They are scheduled dynamically for maximum efficiency, as synchronizations cannot be predicted with enough precision.
- The **lower level** corresponds to the hierarchies of task which have only dependencies with tasks on the same processor. This is represented by the plain purple set in the figure. As no interaction with other processors takes place at this level, it is sufficient to organize the tasks in an order which satisfies the data dependencies.

This separation between the level is done automatically, based on the model of the system, and especially the information on the mapping of the application on the hardware architecture. In our work, the dynamically scheduled entities are called *activity*. An activity corresponds to one leaf in the hierarchy of higher-level tasks (e.g.: tasks B and C in the figure). The higher nodes of the hierarchy, composite components, only influence these activities by adding synchronizations and repetitions. Moreover, as the application domain is the *regular* intensive signal processing, there is no low latency scheduling requirement. Therefore, a simple *cooperative* scheduling can be used, instead of the more complex and costly *preemptive* scheduling. After a task starts, it is never interrupted until the computation is finished. The scheduling policy is also kept simple: the first ready task is executed. Tasks are executed on the processors they are associated with. When a task is distributed, it is executed in parallel on each processor. Within a processor, repetitions of a task are sequentially executed (physically, there is only one execution thread anyway).

4.2. Synchronization mechanism

In order to enforce the data dependencies between the tasks, synchronization must take place. The MoC allows several readers of the same data. As a task can be distributed over several processors, there can also be several writers to the same data array (although each element is only written once). Therefore the synchronization mechanism must allow several tasks to wait for a data being produced, and several tasks to signal they have finished generating their part of the data.

The MoC does not enforce the level of granularity of synchronization: one repetition of a task could be executed as soon as all the input elements it needs are available, even if the whole data array is not yet entirely ready. However, synchronizing at such a fine granularity leads to a high overhead: every single read or write

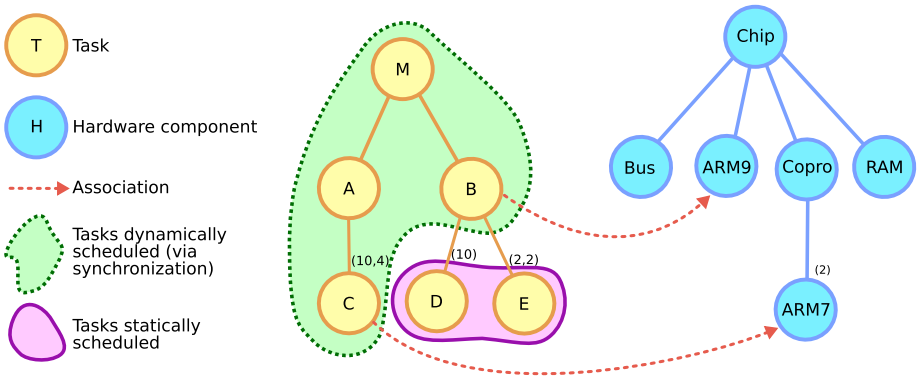

Fig. 3. Task scheduling depending on the hierarchical level and the mapping.

of an element corresponds to one synchronization. For the context of intensive signal processing, it is far more effective to synchronize only at the granularity of a data array (i.e.: only at the beginning and end of a task). The mechanism is defined as follows:

- Each consumer waits until all their input arrays are available.
- Each producer waits until every output array can be written before starting.

In practice, the synchronization scheme selected can be implemented via a FIFO mechanism with multiple readers and shared writers with a length of 1. For each generated array, one FIFO is created. Consumers must read one data from each FIFO corresponding to its arrays. Producers write data as soon as they start their iteration, so they are blocked if the data has not been read by all the consumers yet.

4.3. Data transfer and address computation

Access to the data is expressed in the MoC as reading or writing a small part (a pattern) of a bigger array according to the tiler. For every level of hierarchy, a pattern can itself be separated in smaller patterns by sub-tasks. This hierarchical access to the data can be implemented either by copying each pattern of each level into memory, or by accessing the original array via a series of address computation. Due to the general restriction of memory in MPSoCs, and the high access cost, the second alternative has been selected for the execution model. The only drawback of this scheme is when several tasks deeply buried in the hierarchy access the same data, data-copy could be more efficient. In practical terms, assembly connectors (between two instances) in the application model are equivalent to a FIFO in the execution model, and delegation connectors (between a component and an instance) are equivalent to address computation. As a series of tilers can be represented as just one larger tiler, the implementation is kept simple.

Moreover, address computation must have an additional level of indirection if the data is spread over several memories (specified in the association model). In this case, one additional tiler computation is necessary to determine which memory bank contains the data. Care has to be taken also when a data array is produced and consumed by tasks contained inside a task distributed over several processors. On each processor, the data in the array is different, as it corresponds to different repetitions of the bigger task. Therefore, the FIFO used to contain the array is a different one for each processor.

5. High level SystemC simulation

In this section, we propose an efficient simulation technique using the standard SystemC and its TLM 2.0 kit. The main objectives of this proposal are first to verify the functionality of MPSoC by the means of rapid system virtual prototyping. The second objective is to allow software engineers to perform a timing analysis and to monitor the traffic of patterns over the interconnect created by the execution of concurrent tasks. For the MISP application domain which is mainly data-flow oriented, it is very important to customize the software and the hardware according to the needs. The third objective is to offer a sufficient accuracy level for a fast and reliable DSE. The simulation is described at the timed TLM level. In this level, the simulated MPSoC reflects the high level modeled system (hardware and software) using the MARTE profile. It simulates the same pattern transfers as similar to the real system.

Fig. 4 highlights the differences between our approach (right) and the traditional hardware/software simulation approaches (left). In our approach, all the hardware architecture but the processing elements is simulated as in the traditional TLM simulation. While the processing elements (the application) are usually simulated with an ISS, in our approach the application is simulated without this level of indirection. Software tasks, scheduler, and synchronization mechanisms are directly and explicitly executed as part of the simulation, instead of being a decoded sequence of instructions from the binary code of the software. This allows improvements not only in the simulation speed but also in the ease of observation and debugging. A timing model is defined to approximate the execution time. These different aspects will be elaborated in the upcoming subsections.

5.1. Hardware description

In order to simulate most of MPSoC architectures easily with timed TLM, several generic hardware components have been developed for this level of abstraction. Our library includes parameterized data and instruction caches, a generic bus, a crossbar interconnection network, SRAM modules, several hardware accelerators, etc. These components are generic, allowing easy DSE. The description of the component interfaces is sufficiently flexible so that a variety of communication protocols can be implemented, such as the OCP [25] standard. In addition, our components are TLM 2.0 compliant to facilitate IP integration from other libraries and vice versa. More details on the hardware component design are presented in [26].

5.2. The virtual processor

Processors are simulated in a special way. Instead of using an ISS, the processor corresponds only to the software it would execute, compiled specifically for the machine hosting the

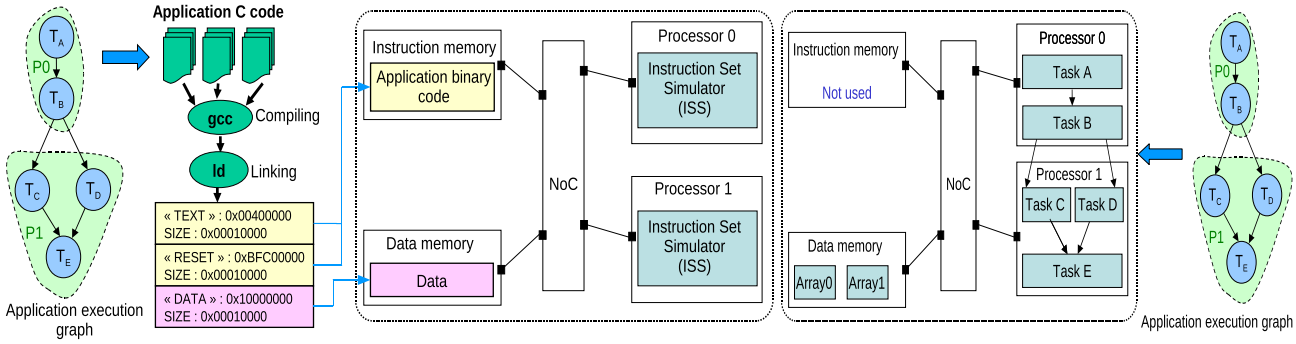

Fig. 4. Schematised simulation using the traditional approach (left) and the virtual processor approach (right).

simulation, plus a set of methods in charge of implementing the low-level functionalities such as read, write, or idle.

The main advantage of this approach is that the simulation can be highly speeded up, as the software is directly compiled for the host instead of being interpreted instruction by instruction. Besides, it provides other appealing advantages. In particular, it provides a high flexibility for simulating any type of processors, even those for which an ISS is not yet available. For example, when starting the exploration of the hardware architecture, it is useful to test different topologies and different types of processors, including mix of processor types. This can be done by simply defining the properties of each type of processor (e.g.: size of a word, frequency, cycles to execute an address computation), without having to create or acquire an ISS for each type tested. Another advantage concerns the tooling. For the software developer, it permits to debug the application with all the usual debugging tools for the host machine, which in general have a more complete feature set and to which the developer is more accustomed to. It also allows to keep the relationship between the data passed on the interconnects and the data patterns expressed in the high-level model.

This type of simulation can be applied to hardware accelerators as well. Typically, hardware accelerators provide very high performance for a specific task, part of the whole application. The elementary tasks which are mapped on such processing resource are written in a hardware description language such as VHDL or Verilog. Traditionally, the simulation has to be done at a low level such as RTL, or via a specially crafted software simulator. When using the Virtual Processor technique, the simulation component is generated in a similar way as for the processors: it is replaced by the tasks which are mapped on it, wrapped with the same small set of low-level functionalities.

Let us note that this approach is made possible by several specificities of the context, principally by the fact the system is modeled at a high level of abstraction, and by the fact we are targeting intensive signal processing systems. A simulation can be generated without manual modifications in the software source code or in the model of the system because:

- All the data transfers between the processor and the rest of the system are explicit: the reads and writes from/to the memory are expressed in the model via the tilers.
- All elementary components are available in a sufficiently high-level language (typically, C), so that they can be compiled for any type of processor architecture. This, nevertheless, does not prevent an optimized version of the components to also be available for the target processor. Both implementations can be provided, and the more fitting one for the target will be selected.

The simulation keeps a good accuracy because instruction and synchronization transfers via the interconnect are *largely smaller* than the data transfers. This condition is met in the MISP domain: the same computations are repeated on a very large number of data, so the instruction cache of the processor has a very low rate of misses. To keep a high simulation accuracy, elementary tasks should access amount of data not too large so that it always fit in the data cache. This is because the cache misses caused by the code of the elementary tasks are not simulated.

The implementation principle relies on the newly defined concept of *virtual processor*. A virtual processor provides a defined set of methods corresponding to the primitive functionalities of a processor (e.g.: read, write). At initialisation, the core of the virtual processor component calls the starting function of the software, which has been compiled for the machine hosting the simulation. The software has been adapted to use the processor primitive methods where necessary. In other words, the software is executed directly on the host processor, excepted for the instructions affecting the external behaviour of the component, which are simulated by calling the virtual processor special methods.

In our implementation, only three primitive functions are provided by the virtual processor: `read`, `write`, and `idle`. The first two functions, quite intuitively, allow to read a word from a given address or write a word to a given address (Listing 1, line 30). The `idle` function is used when the software has nothing to do (typically because it is blocked, waiting for some data to be generated). This function informs the rest of the SystemC components that for a short time this component will not generate any output and therefore does not need to be simulated. This allows to speed up the simulation by avoiding active polling during these moments. In a general case, there could be additional methods to allow configuring and receiving hardware interruptions, but this was not implemented as the execution model does not make use of this mechanism.

Listing 1 presents an example of virtual processor. The method `run` is used by SystemC to start the component's simulation. It calls the main function of the application `schedule`, which takes care of executing the tasks composing the application (as presented in the next subsection). A task (for example `task_C`) is always generated according to the execution model. It begins with a nested set of synchronizations and repetitions, corresponding to the high-level hierarchy. Then the input patterns are read, corresponding to the tilers. Then comes the static code corresponding to the low-level of hierarchy, which in its most simple form (like in this example) consists in solely calling the elementary function. Finally, the output patterns are written, and the synchronizations are used to inform other tasks that the data is available.

5.3. Scheduling and synchronization

As tasks are executed directly on the host, it is possible to optimize further the simulation by also executing the low-level software management directly on the host. More precisely, this is

```
1  void MIPSCore::run() {
2    schedule();
3  }
4  void MIPSCore::task_C (struct pa_task* a) {
5  // Pattern arrays definition of the task_C
6    float Tab_in[16], Tab_out[16];
7
8  // Synchronization with other processors.
9    sync_1.startReading();
10
11 // Reading input patterns of task_C
12   for (int i=0; i<=15; i++)
13     read(addr+i, &Tab_in[i]);
14
15 // Execution of elementary task C
16   elem_C(Tab_in, Tab_out);
17   Timer->add(Task_delayC);
18
19 // Writing output patterns of task_C
20   for (int i=0; i<=15; i++)
21     write(addr+i, Tab_out[i]);
22
23 // Synchronization with other processors
24   sync_1.finishWriting();
25   ...
26 }
27 void MIPSCore::task_D (struct pa_task* a)
28 {...}
29
30 void MIPSCore::read(const ADDR_TYPE &addr,
31                     unsigned int &data) {
32   int time = 0;
33   Timer->sync();
34   Port->read(address, data, time);
35   Timer->add_nosync(time);
36 }
```

Listing 1. Pseudo-code of a virtual processor.

the part of the software added by the execution model: the scheduling and synchronization mechanisms. This avoids to spend time simulating the synchronization (which is often based on shared memory accesses) and permits to more easily observe the application while debugging.

The implementation of a synchronization follows very much the implementation used for lower levels of abstraction: a tuple is shared in memory and accessed atomically in order to track the number of writers and readers of a data array. Instead of being contained in the memory of the simulated system, each tuple is actually one global variable shared between all the virtual processor instances.

As defined in Section 4, the task scheduling is composed of two parts. The lower tasks in the hierarchy are statically scheduled. Each set of statically scheduled tasks constitutes an *activity*. Activities are dynamically scheduled, according to the higher part of the task hierarchy. In the SystemC implementation, each activity corresponds simply to the concatenation of the code of all the sub-tasks. The simulation of the dynamic scheduler relies on the POSIX threads in order to schedule and unschedule the activities. Although the thread technology is usually used to run concurrently several tasks, here it is used only to manage efficiently the context swapping while being architecture-independent.[5] The implementation ensures that, for each processor, there is never more than one activity running at the same time (similarly to the execution on the actual processor). Each thread is by default blocked and unblocked one at a time by the scheduler, according to the scheduling policy. The scheduling policy algorithm is written as a separate function, and can therefore be independent of the abstraction level. When all the activities are blocked on a synchronization, the scheduler uses the special `idle` method of the virtual processor in order to simulate idle time (and skipping the unneeded simulation cycles). The scheduler is also in charge of detecting when all the activities have finished running. This permits to detect the end of the simulation: when the activities on every processor have terminated.

[5] Using SystemC threads here would not work, as these threads are handled by the SystemC scheduler, while we need to provide a specific scheduler per processor, with a specific scheduling policy.

5.4. Time modeling

To provide performance estimation for DSE, a timing model is integrated in our simulation. The performance estimation methodology must be able to adapt to different architecture topologies such as distributed memory and hierarchical systems. In addition, the solution must consider timing issues such as those associated with the processor's synchronization, or the contentions in the interconnect and communication protocol specifications. At the early development stage for which this abstraction level is used, it is neither useful nor possible to obtain an accuracy of one cycle or even one instruction. The accuracy needed is at the granularity of a data-access pattern: so that it is possible to know in which order are the requests on the bus and whether some overlaps.

On the hardware simulation side, the proposed timing model is compatible with the TLM 2.0 kit model. Every transaction is timed. To keep track of the time on the software side, several timing concepts are introduced, such as the *local timer* which is attributed to each processor or hardware accelerator. Its value is incremented after each task execution (Listing 1, line 17), and after external transaction from the component (such as a *read*, line 34 of Listing 1). The main advantage of using these local timers is to allow time decoupling between the different processors. Thus, the simulation can be faster by reducing overheads of switching often between the execution of each processor component.

Furthermore, in order for a processor to know the time elapsed during a transaction, a *time* parameter is used in the transaction payload. It is incremented by every component through which the transaction passes, from the request until the response. When the response reaches the processor, the information is used to increment the processor's local timer (Listing 1, line 35). For example, in the case of data cache hit, the elapsed time corresponds to the cache read time. In the case of a data cache miss, the cache initializes a new request that is transmitted via the interconnect to the corresponding target. The elapsed time equals the sum of the transmission time, the cache read (or write) time, and the memory access time. This technique allows to handle cases where the transmission time is not constant. For instance, network contentions may slow down the transaction. Before transmitting a request to a target, the interconnection network polls all input FIFOs to compare *time* parameters of the present transactions and determines which one will be selected for servicing. This allows to simulate contentions and therefore to correctly evaluate the request waiting time at the interconnect level.

This strategy requires the component's developer to identify for each component the pertinent activities concerning the time. In particular, this includes the execution of each elementary task for a given processor or hardware accelerator, data hits and misses for the caches, transmission/reception of a packet on the interconnect, read and write access for the shared memory modules, etc. Execution time estimation requires attributing an average delay value to each type of activity. In our approach, execution times for the hardware are either measured from a physical characterization of the hardware component or from an analytical model at a low abstraction level (using tools such

as CACTI [27]). The execution times for the software can be either measured by executing each elementary tasks once on an ISS or by estimation from the number of executed instructions. For the hardware accelerator, the usual simulation tools at the RTL level are used to estimate accurately the execution time of each task. Precise informations such as architecture pipelines, instruction latencies are all comprised in these average execution times. These values are stored as deployment information in the model, so that they are automatically used in the simulation.

6. Experimental results and evaluation

This section illustrates the usage of our approach to perform DSE. For this purpose, two case studies are used: the *Downscaler application*, which has already been briefly presented in Section 3, and the *H.263 encoder application*, which is also a typical MISP application. The possible architecture to execute these applications has several parameters which can vary: the number of processors (4–16), the number of memory banks (1, 2 and 4), and the cache size (2–64 kB). As the applications are based on several tasks, with potential parallelism present at several granularities, the mapping of the application on the hardware can be varied in many different ways as well, included asymmetric distributions. Therefore, both systems require a vast DSE to find the minimal architecture and the most efficient mapping.

After a more detailed description of these two case studies, and an overview on how the simulator is generated, the virtual processor approach will be compared against two other simulation approaches: a cycle accurate simulation (CA) which can be expected to be slow but to have a good precision, and the one provided by the OVPSim tool[6] which aims at a fast simulation. Primarily, the approaches are compared in terms of the two properties: simulation speed, and precision of the results. Later on, we will present via a case study how our approach allow precise observations of the various global properties of the system required to perform efficiently high ESL prototyping such as data accesses to the memory or network contentions over the time. A last case study will demonstrate the ability of our approach to simulate heterogeneous architectures, which mix both processor and hardware accelerator cores.

6.1. Example systems

To design the systems, the choice of a given mapping for our applications and the architecture on which they will be executed can not be done arbitrarily. In fact, this depends on the application requirements in terms of computation and communication resources. For instance, running the Downscaler with a rate of 15 f/s, 30 f/s or 50 f/s corresponds respectively to an execution time of less than 66 ms, 33 ms or 20 ms per frame. Depending on the frame rate required the computing power which must be provided by the architecture will be different. Profiling the application first on a simple architecture with only one 150 MHz MIPS processor showed that the total processing time per frame (400 ms) is widely greater than the application requirements. Thus, using a multiprocessor architecture is mandatory. For the communication requirements, based on the application task graph (Fig. 2), the quantity of data load (read and write) from memory can be approximated for each task based on input/output ports and the corresponding shape.

Fig. 5 represents a possible mapping of the Downscaler application onto the architecture which has 4–16 processors with private caches, one or several memory banks as storage resource, and a crossbar for the interconnections between these components. The workload (horizontal and vertical filters) is distributed on the processor by separating the data processing at the component level of the frame. The first processors execute the parts in charge of the luminance, while the rest of the processors handle the Cr and Cb components. For data mapping, a memory bank can be allocated for each frame component in order to reduce the conflicts between simultaneous accesses to the same bank. The complete specification of this case study, as well as the GASPARD2 environment used to generate the simulation are publicly available for download and testing [28]. Of course, these mapping and architecture sketched are solely one possible implementation.

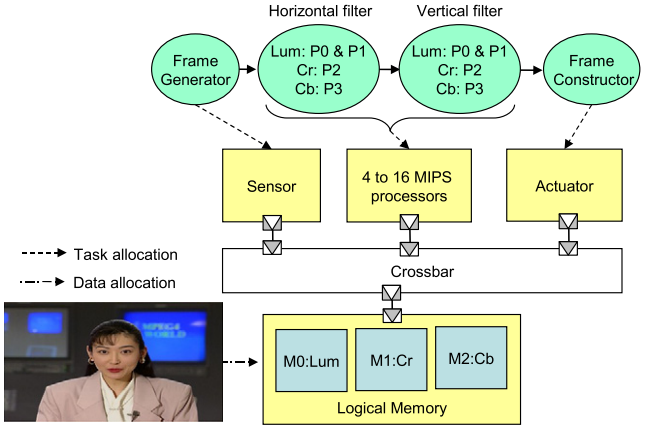

Fig. 5. Example of Downscaler mapping onto a homogeneous MPSoC.

The second example application is the intra part of an H.263 encoder. It receives a QCIF frame and encodes it. It is mapped on an architecture similar to the one used for the Downscaler. Tasks are distributed homogeneously in such a way that each processor handles a given set of frame blocks [23]. Further on, we will also present the H.263 encoder mapped and simulated on a heterogeneous architecture.

6.2. Design environment and simulation generation

As stated previously, the VP-based simulation is facilitated by the fact that the system is modeled at a high-level, via the MARTE profile. Our environment GASPARD2 [28] allows to design the high-level MPSoC specification and to generate automatically the VP-based simulation implemented in SystemC. This is particularly pertinent for DSE since this permits implementations to be automatically regenerated, after modifications in the high-level system specification. Our environment relies on an efficient design methodology called Model Driven Engineering (MDE) [29]. This methodology reduces the development and maintainability efforts of MPSoC design tools, and provides a high-level abstraction to the designer. Fig. 6 details how our tool generates the SystemC simulation (which is only one among several targets). The MPSoC model in UML is the input of a chain of transformations which go through several intermediary models until reaching the simulation source code. The simulation can then be compiled and executed to obtain the results. More details about the chain and the advantages of MDE for SoC design tools are presented in [30].

Concerning the generation of the simulation code, first, we have measured the elapsed time taken to generate the system simulator code starting from the high-level specification of the Downscaler system for various sizes of the MPSoC. Let us note that in all these studies, a computer equipped with a 2.33 GHz Xeon processor and 4 GB SDRAM is used. The code generation time was approximately 3 s, independently from the size of the system. This small duration associated with the ease to change the high-level model ensures a

[6] http://www.ovpworld.org/

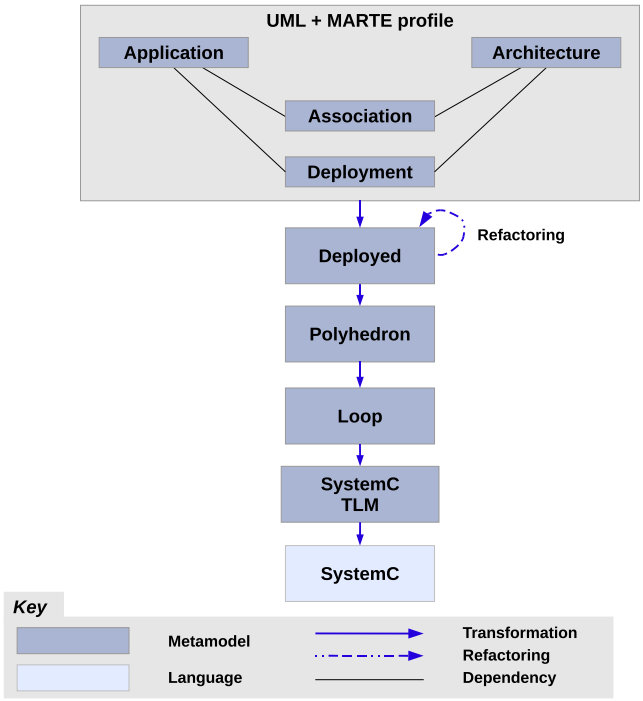

Fig. 6. Transformation chain towards SystemC.

fast DSE. For instance, changing the number of processors or the memory banks only requires to modify two numbers in the model and the Distribute link parameters. The generation time is constant because the system's representation is kept factorized all along the transformations. Independently of the number of instances, the code size remains the same. Approximately 1450 lines of code were generated.

The code is not intended to be modified by the user directly. Nevertheless, the generation was designed to create clear code which can be easily associated with the objects defined in the high-level model, so the debugging can be easily performed. Although it makes no doubts that manual coding could yield some improvements of the simulation speed, in most cases, these potential improvements would likely remain small. With the current implementation of the environment, the code is compact and benefits from some loop optimizations in the application applied during one step of the transformation chain. If required, additional transformations steps could be added to automatically optimize further the simulator. To give an insight on the appearance of the code to the reader, an extract of a generated code for the connection between repeated memory banks and the interconnect is presented in Listing 2.

```
for(reptitionSpaceI_Reshape_1b71[0]=0;
    reptitionSpaceI_Reshape_1b71[0]<
      repetitionSpace_Reshape_1b71[0];
    reptitionSpaceI_Reshape_1b71[0]++){
  for(int i = 0 ; i < 1 ; i++) {
    // Origin
    arrayInI_Reshape_1b71[i] =
      originMatrix_In_Reshape_1b71[i];

    // Paving
    for(int j = 0 ; j < 1 ; j++)
      arrayInI_Reshape_1b71[i] +=
        reptitionSpaceI_Reshape_1b71[j] *
        pavingMatrix_In_Reshape_1b71[i][j];
  }
  Bus_pointer->
    slave_port[arrayInI_Reshape_1b71[0]]->
    bind(*DataMemory_pointer->target_port);
}
```

Listing 2. SystemC generated code.

6.3. Accuracy and performance of the simulation

In order to evaluate the Virtual Processor-based simulation, we compare it to two other simulation techniques:

- A CA technique written manually in SystemC (for the hardware) and C (for the software). Such technique is currently in use industrially and is expected to provide good accuracy of the results but takes a long time to run.
- A high-level technique using the OVPSim tool [31] which aims, as our technique, at rapid system level virtual prototyping. The architecture was developed based on the code provided with the tool. Unfortunately, within this version only the binary code of the platform is available which prevented us to accurately match some timing parameters such as the buses and memory latencies with the ones modeled in the other simulations. The

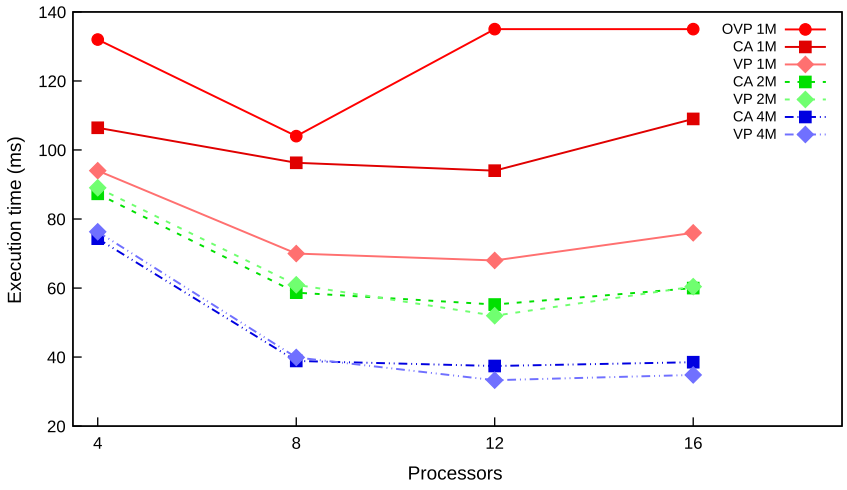

Fig. 7. Estimated execution time of the Downscaler with the VP, CA and OVPSim simulations, for different numbers of processors and memory banks.

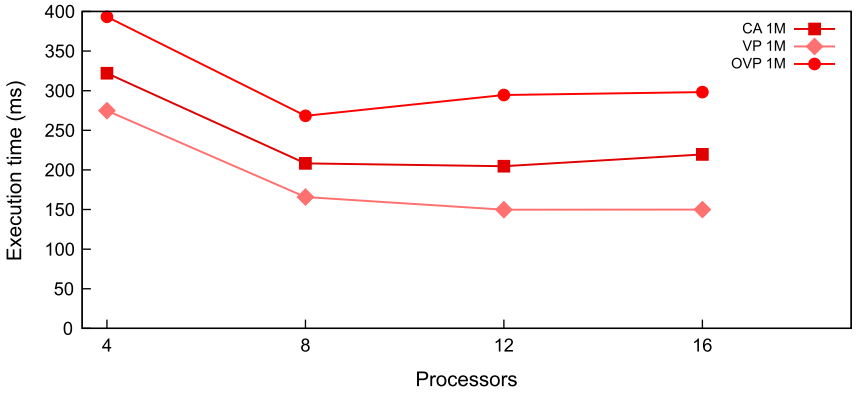

Fig. 8. Estimated execution time of the H.263 with the VP, CA and OVPSim simulations, for different numbers of processors with 1 memory bank.

software was an adapted version of the C manual implementation. OVPSim is a very fast simulator since processors are not ISS but use code morphing and just-in-time (JIT) compilation [17].

In order to evaluate the simulations, two main characteristics were measured: the simulation time, to compare the simulation speed, and the estimated execution time for a given set of input, to compare the precision of the techniques.

First, with respect to the precision, Fig. 7 summarizes the estimated execution time of the Downscaler system for the three types of simulations, with the number of processors varying from 4 to 16. For the CA- and VP-based simulations, the architecture varied between 1, 2, and 4 memory banks (noted respectively 1 M, 2 M, and 4 M). This represents a basic exploration of the architecture space for the system. Similarly, Fig. 8 summarizes the estimated execution time of the H.263 system for the three types of simulations with one memory bank and a number of processors varying from 4 to 16.

From Fig. 7, several remarks can be drawn. First, between 0.7% and 30% of under-estimation can be noted on the VP-based simulation compared to the CA simulation. Investigation using more detailed information (as presented in the next subsection) showed that the maximum error is obtained when a communication bottleneck occurs in the interconnection network for a long period of time. This is the case of configurations using only 1 memory bank (noted 1 M). On the other hand, the OVPSim simulation over-estimated the execution time between 8% and 44%. In the case of the H.263, between 8% and 32% of under-estimation is found with the VP-based simulation as shown in Fig. 8, while the OVPSim simulation over-estimates the execution time between 22% and 43%.

As a side note, we can validate the assumption mentioned in Section 5, that MISP applications have much less bus transactions for the instructions than for the data, by observing the ratio in the CA simulation. This ratio was respectively of 0.17% and 0.7% for the Downscaler and the H.263 encoder.

One important observation to make is that, although VP tends to underestimate the execution time compared to CA, the relative order between the configurations is respected. In particular, both simulations show that a 12-processor architecture gives the best execution time for the Downscaler application. This is explained by that the fact that due to the conflicts for accessing the data memory, increasing the number of processors is inefficient. This confirms the correctness of the proposed execution model for performance estimation and hence the reliability of exploration. For the H.263 encoder results, we have captured the same behaviour. This is not the case for the OVPSim simulation, which showed a much different profile, estimating that 12 and 16 processors lead to the worse configurations, and 8 processors provided the fastest execution. For the architecture exploration, OVPSim did not have a sufficient accuracy.

The plots allow to find the appropriate architecture with minimum resources. For instance, an 8-processor architecture with two memory banks seems suitable to run the Downscaler at a 15 f/s rate. The H.263 encoder requires 12 processors.

The second part of the study consists in comparing the simulation speeds. Fig. 9 represents on a logarithm plot the acceleration factor of the simulations compared to CA. All the architectures have one memory bank, and a number of processors varying from 4 to 16. Here, OVPSim shows a large advantage with an acceleration factor between 350 and 1000 over CA, compared to our approach which brought an acceleration with a factor of 12–37. We

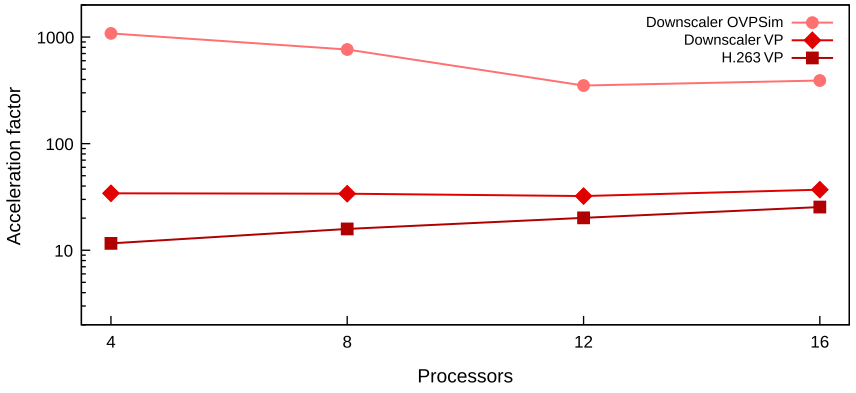

Fig. 9. Acceleration factor of the VP simulation and OVPSim over the CA one for H.263 and Downscaler applications, for different numbers of processors.

can note that OVPSim's acceleration tended to be lower with more processors. In absolute values, this simulation was three times slower for 16 processors than for 4 processors. On the contrary, the VP-based simulation had a higher speed-up for a higher number of processors. In absolute values, this simulation took almost always the same time to run, independently of the number of processors. For instance, the H.263 system was simulated in 17 s per frame with 4 processors, and in 15 s per frame with 16 processors.

To evaluate more specifically the VP technique, we have manually modified the 16-processors VP simulation to use the ISS component of the CA simulation. While with the ISS the H.263 simulation took 47 s per frame of which 6.7% of the time was directly related to processor simulation, with VP the simulation took 15 s of which only 0.9% was spent on the processor components. This technique permits to reduce the simulation of the processor to a negligible part of the whole simulation. Indirectly, more time is saved because the VP-based simulation also reduces the overhead of the SystemC kernel by avoiding a synchronization after each simulated instruction.

To summarize, our prototype implementation of the Virtual Processor approach allowed to increase the speed of the simulation by a factor of 12–37 compared to a Cycle Accurate simulation. This is far less than the highly optimized OVPSim tool which showed speed-up factors of up to 1000. Nevertheless our approach showed better precision in the results from the simulation, important to locate the most adequate configuration of a solution space. Moreover, the speed-up in the VP approach comes directly from the highly increased speed of the processor simulators, and indirectly from the reduced number of context switches between the components of the simulator. Consequently, the simulation technique adapts very well to the increasingly common parallel architectures. Another advantage of our technique is that the components simulating the other parts of the architecture can be kept as is from the lower abstraction level simulations in SystemC. This interoperability is vitally needed for an efficient ESL design.

6.4. Architectural optimizations

As one of the final objective of the VP technique is to provide values accurate enough to guide the architectural optimization. Here, two case studies are presented to demonstrate how our approach can permit such optimization.

Firstly, we present the observation of contentions over the interconnection network to select the optimal number of memory banks. This experiment is done with the Downscaler application on a 12-processor architecture. During the simulation, pertinent information about the system execution can be recorded into files for later analysis. Fig. 10 shows the number of contentions in the interconnection every 1 ms of execution time when using different number of memory banks, with the VP simulation. It shows that with one memory bank, the interconnection was saturated all along the execution. All configurations using only one memory bank can therefore be eliminated from the solution space without simulating the system with other parameters modified. With a higher number of memory banks, similar analyses permit to pick the best data distribution, and the optimal channel size.

In a second case study, we determine the minimal processor cache size of a 4-processor MPSoC. Fig. 11 shows the data memory

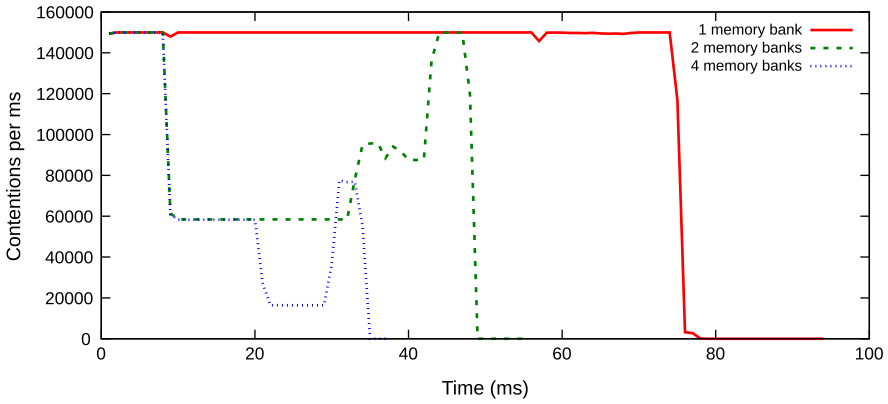

Fig. 10. Contentions in the network on a 12-processor MPSoC.

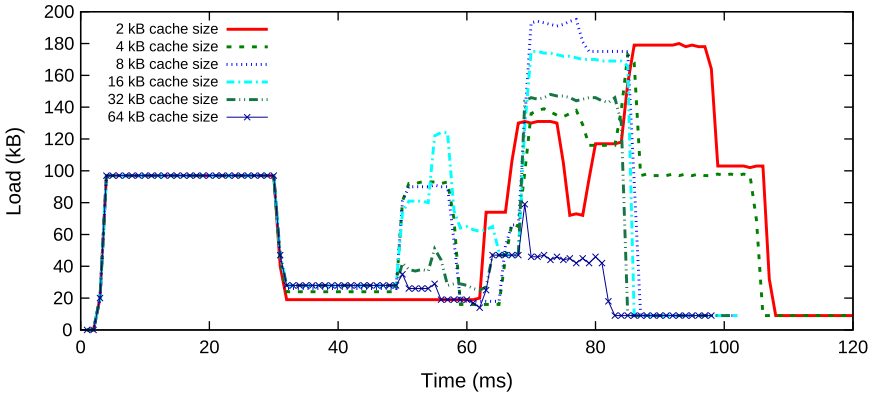

Fig. 11. Data reads on a 4-processor MPSoC.

load (reads in kB), sampled every 1 ms of execution time, for different processor cache size ranging from 2 kB to 64 kB. In general, caches with high size reduce the main memory accesses and thus improve the overall system performances. As here all processors execute the same tasks, we explore only symmetrical cache sizes. However, for configurations where the data structures are not the same on all processors, asymmetrical cache sizes could be tested as well. In our example, the move from 2 kB to 8 kB improves the execution time by 17% (from 123 ms to 102 ms). Over 16 kB, the performance almost did not improve, so increasing the cache size further than 8 kB would be useless. This is due to the limited number of communicating tasks of the Downscaler application (mainly two). Still, every cache size lead to a different pattern of load, showing that our simulation technique permits to observe transactions at a granularity fine enough to perform architectural optimization.

6.5. Heterogeneous MPSoC case study

So far, the case studies which have been described all involved an homogeneous architecture, with identical processors. In this part, we present how VP-based simulation can benefit also to systems with a heterogeneous architecture. The OVPSim tool can simulate heterogeneous multiprocessor platforms such as a 2-processor architecture with a MIPS and an ARM cores. In our approach, this aspect is extended to include processor and hardware accelerator cores. In addition, the system (application and architecture) description is generated automatically from the high-level specification allowing a rapid system level prototyping. With OVPSim, the architecture and the application are developed manually. Hand-coding is not suitable for an efficient development of large embedded systems because it is tedious, error-prone and expensive.

Previously, simulation results have shown that the H.263 encoder requires 12 processors with 4 memory banks to be run at a 15 f/s rate. To reduce hardware resources, design space is further explored by introducing an hardware accelerator to support the DCT task, in addition to the MIPS processors.

The choice of the hardware accelerator was driven principally by the comparison of processor usage for each task. Gaspard2 can report for each software component the time spent to execute it on the processors, and the time spent in synchronisations between all processors (i.e.: waiting for data, synchronization barriers). In the VP simulation, as the tasks are still explicitly separated, such measurements are straightforward to perform. Fig. 12 illustrates the processor usage percentage of the main tasks of the H.263 case study and of the parallel overhead, depending on the number of processors in the architecture. The DCT task is the most time consuming. This observation is compatible with other simulation

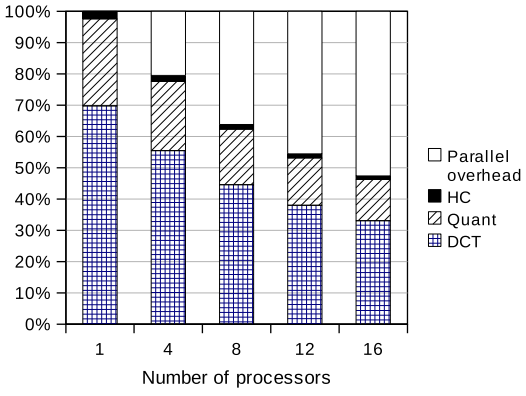

Fig. 12. Processor usage per frame for the intra part of the H.263 encoder.

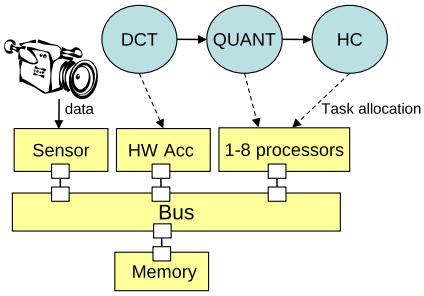

Fig. 13. Example of H.263 mapping onto a heterogeneous MPSoC.

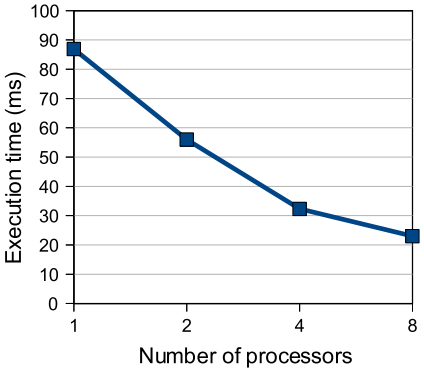

Fig. 14. Virtual processor simulation with heterogeneous MPSoC architecture.

results found in the literature [32,33] about the H.263 encoder. The DCT task is known to be efficiently executed by hardware accelerators.

In Gaspard2, a special transformation chain which targets VHDL [34] is used to generate the DCT hardware accelerator. The accurate timing information extracted from an RTL simulation of this generated component is used to annotate at the higher system level for the DCT Virtual Processor. Using this DCT hardware accelerator, a new architecture is modeled and the application mapped on it, as schematised in Fig. 13. The VP-based simulation is then used to observe four variations of this system: with 1 to 8 MIPS processors. The DCT hardware accelerator is simulated in a similar way than the general-purpose processors, excepted that the timing information for the DCT task contains the values measured by the low-level simulation. The simulation results are shown in Fig. 14. The selection of the appropriate implementation is driven by the application requirements and/or constraints on the hardware resources. For instance, the results of the simulations show that a heterogeneous 2-processor architecture with the generated DCT accelerator should be suitable to run the H.263 encoder at a 15 f/s rate. While this curve was obtained in approximately 10 min, the same result with usual synthesis tools from a corresponding VHDL code would require several hours. Hence, performing short estimations of the RTL models of each component followed by VP-based simulations of potential system configurations permits to accelerate the design space exploration.

7. Conclusions

Targeting the MISP application domain, we have presented a new ESL simulation approach, adapted to the MPSoC design. Our approach speeds up the simulation by leveraging the high-level modeling provided by novel contributions such as MARTE. It introduces the notion of Virtual Processor which, in its essence, consists in replacing the processors by the software tasks which are mapped

on it and adding a wrapper to translate external behaviour of the processor. Performance can be further optimized by executing also on the host the scheduling and the synchronizations. In order to ensure the same behaviour between the high-level simulation and the final implementation, an execution model adapted to MISP applications was introduced. In addition to the performance advantages, the VP approach also simplifies the debugging, facilitates timing analysis and transfer patterns observation, handles heterogeneous architectures, and allows to deal with processors which do not yet have a simulator component available.

In the case study, the approach was put into the context of DSE with the usage of two typical examples: the Downscaler and the H.263 encoder. The simulation offers at least a 10-fold speed-up over a traditional CA simulation, while providing a relatively better precision than faster simulations such as in OVPSim. Coupled with our environment, our simulation is well adapted for rapid prototyping of MPSoC systems, permits a reliable hardware/software DSE, and is compatible with the TLM 2.0 kit to facilitate IP integration. Hence, our approach contributes to increase the productivity of MPSoC designers.

Future research will focus on several areas. First, we will enhance our VP-based simulation with power estimation tools for multi-objective DSE. Second, it could be interesting to target SystemC simulation at lower abstraction level to explore solutions selected with VP-based simulation. Third, we would like to evaluate the possibility to integrate the VP technique with simulations of hardware components at higher levels. Finally, we plan to automate the DSE phase so that simulation results can pilot modifications of the input model and control the target simulation level.

References

[1] W. Wolf, A.A. Jerraya, G. Martin, Multiprocessor System-on-Chip (MPSoC) technology, IEEE Transactions on CAD 27 (10) (2008) 1701–1713.
[2] T. Meyerowitz, Transaction Level Modeling Definitions and Approximations, 2005. <http://www.eecs.berkeley.edu/~alanmi/courses/2005_290A/reports/290a_modeling.pdf>.
[3] P. Boulet, Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing. Research Report RR-6467, INRIA (March 2008). URL <http://hal.inria.fr/inria-00261178/en/>.
[4] G. Kahn, The semantics of a simple language for parallel programming, in: J.L. Rosenfeld (Ed.), Information Processing 74: Proceedings of the IFIP Congress 74, IFIP, North-Holland, 1974, pp. 471–475.
[5] E.A. Lee, D.G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.
[6] M.J. Chen, E.A. Lee, Design and implementation of a multidimensional synchronous dataflow environment, in: Proceedings of the IEEE Asilomar Conference on Signal, Systems, and Computers, 1995.
[7] Object Management Group, A UML profile for MARTE, 2007. <http://www.omgmarte.org>.
[8] A. Amar, P. Boulet, J.-L. Dekeyser, F. Theeuwen, Distributed process networks using half FIFO queues in CORBA, in: ParCo'2003, Parallel Computing, Dresden, Germany, 2003.
[9] P. Dumont, Spécification multidimensionnelle pour le traitement du signal systématique, ThÃse de doctorat. PhD Thesis, Laboratoire d'informatique fondamentale de Lille, UniversitÃ© des sciences et technologies de Lille, December 2005.
[10] T.M. Parks, Bounded scheduling of process networks, PhD Thesis, EECS Department, University of California, Berkeley, CA, December 1995. URL <http://ptolemy.eecs.berkeley.edu/publications/papers/95/parksThesis/>.
[11] J.T. Buck, Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, University of California at Berkeley, 1993. URL <http://ptolemy.eecs.berkeley.edu/publications/papers/93/jbuckThesis/>.
[12] Open SystemC Initiative, SystemC, World Wide Web document, 2008. URL <http://www.systemc.org/>.
[13] IEEE, System Verilog, 2005. <http://www.systemverilog.org>.
[14] L. Cai, D. Gajski, Transaction level modeling: an overview, in: Hardware/Software Codesign and System Synthesis, 2003, pp. 19–24.
[15] The SoCLib project: an open modelling and simulation platform for system on chip design. <http://soclib.lip6.fr/>.
[16] G. Beltrame, L. Fossati, D. Sciuto, ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28 (12) (2009) 1857–1869.
[17] B. Bailey, System level virtual prototyping becomes a reality with OVP donation from imperas, Tech. rep., EDA, June 2008.
[18] R. Wunderlich, T. Wenisch, B. Falsafi, J. Hoe, SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling, in: 30th Annual International Symposium on Computer Architecture, San Diego, USA, 2003.
[19] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, B. Calder, Discovering and exploiting program phases, IEEE Micro 23 (6) (2003) 84–93. doi:http://doi.ieeecomputersociety.org/10.1109/MM.2003.1261391.
[20] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino, SystemC cosimulation and emulation of multiprocessor SoC designs, Computer 36 (4) (2003) 53–59.
[21] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, H. Meyr, Multiprocessor performance estimation using hybrid simulation, in: DAC '08: Proceedings of the 45th Annual Design Automation Conference, ACM, New York, USA, 2008, pp. 325–330.
[22] S. Honda, T. Wakabayashi, H. Tomiyama, H. Takada, RTOS-centric hardware/software cosimulator for embedded system design, in: Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04), Stockholm, Sweden, 2004.
[23] E. Piel, R. Ben Atitallah, P. Marquet, S. Meftali, S. Niar, A. Etien, J.-L. Dekeyser, P. Boulet, Gaspard2: from MARTE to SystemC simulation, in: Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML profile DATE'08 Workshop, 2008.
[24] P. Boulet, P. Marquet, E. Piel, J. Taillard, Repetitive Allocation Modeling with MARTE, in: Forum on specification and design languages (FDL'07), Barcelona, Spain, 2007, pp. 280–285, invited Paper.
[25] I.P. OCP, Open core protocol specification 2.0, 2003. <http://www.ocpip.org/>.
[26] R. Ben Atitallah, S. Niar, S. Meftali, J.-L. Dekeyser, An MPSoC performance estimation framework using transaction level modeling, in: The 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Daegu, Korea, 2007.
[27] N. Jouppi, CACTI home page. <http://www.hpl.hp.com/research/cacti/>.
[28] DaRT Team LIFL/INRIA, Lille, France, Graphical array specification for parallel and distributed computing (Gaspard2), 2008. <https://gforge.inria.fr/projects/gaspard2/>.
[29] D.C. Schmidt, Model-driven engineering, IEEE Computer 39 (2) (2006) 41–47.
[30] E. Piel, P. Marquet, J.-L. Dekeyser, Model transformations for the compilation of multi-processor Systems-on-Chip, Generative and Transformational Techniques in Software Engineering II 5235/2008 (2008) 459–473.
[31] Imperas Inc., OVP World home page. <http://www.ovpworld.org/>.
[32] A. Ben Atitallah, P. Kadionik, F. Ghozzi, P. Nouel, N. Masmoudi, H. Levi, HW/SW Codesign of the H. 263 Video Coder, in: Canadian Conference on Electrical and Computer Engineering (CCECE'06), 2006, pp. 783–787.
[33] S. Jang, S. Kim, J. Lee, G. Choi, J. Ra, Hardware-software co-implementation of a h.263 video codec, IEEE Transactions on Consumer Electronics 46 (1) (2000) 191–200.
[34] S. Le Beux, P. Marquet, J.-L. Dekeyser, Model driven engineering benefits for high level synthesis, Research Report 6615, inria (2008). URL <http://hal.inria.fr/inria-00311300/en/>.

Rabie Ben Atitallah is currently an Associate Professor in Computer Science at the University of Valenciennes and member of LAMIH laboratory within the DIM (Decision, Interaction, and Mobility) team. He is also an associated member of DaRT project at the INRIA Lille-Nord Europe research institute. He is an IEEE member and a member of High Performance and Embedded Architecture and Compilation (HiPEAC) European Network of Excellence. Previously, he received his PhD degree in Computer Science from the University of Lille1 in March 2008. Between March 2008 and August 2009, he had a post-doctoral position at INRIA Lille-Nord Europe and the University of Valenciennes. His research interests include Embedded system design, MultiProcessor System-on-Chip (MPSoC), Low power-aware design, Virtual prototyping, Simulation, and Dynamic reconfigurable computing.

É**ric Piel** born in France, is currently post-doc in Software Technology Department of the Delft University of Technology (The Netherlands) working in the Poseidon project, in partnership with Thales Nederland. The subject of the research is to ease and improve the integration of large-scale component-based systems. Previously, he received his PhD in 2007 at INRIA Lille (France) on the subject of embedded system specification and model transformations, and his engineer diploma in Computer Science at the University of Technology of Compiégne (France) in 2003. He has also worked in the industry in the R&D department of Bull on the subject of mixing Real-Time capabilities and parallel processing.

Smail Niar is a Professor in computer science at the Institut des Sciences et des Techniques de Valenciennes (Valenciennes - France). His research activities are done at LAMIH – "Information, Decision Making & Embedded Systems" Research group. He is also a member of INRIA-Lille DaRT Project and member of High Performance and Embedded Architecture and Compilation (HiPEAC), the European Network of Excellence FP7-ICT programme.

Philippe Marquet is currently an assistant professor at the University of Lille, France and a researcher within the INRIA, the French institute for research in computer science. Philippe MARQUET has received a PhD in Computer Science from the University of Lille in 1992. His research interests include the design of parallel, embedded and reconfigurable architectures, the definition of programming models, languages and compilers dedicated to parallel computing. He also worked on the definition and implementation of real-time operating systems for SMP architectures. Recently he has worked on the design of a massively parallel architecture on a chip. He (co-)advised 11 PhD thesis.

Jean-Luc Dekeyser received his PhD degree in computer science from the University of Lille 1 in 1986; afterwards, he was a fellowship at CERN Geneva. After a few years at the Supercomputing Computation Research Institute in Florida State University, where he worked on high performance computing for Monté-Carlo methods in High Energy Physics, he joined the University of Lille 1 in France as an assistant professor, in 1988. There he worked on data parallel paradigm and vector processing. He created a research group working on High Performance Computing in the CNRS lab in Lille. He is currently Professor in computer science at University of Lille 1 and is also heading the DaRT INRIA project at the INRIA Lille Nord Europe research center. His research interests include embedded systems, System on Chip co-design, synthesis and simulation, performance evaluation, High Performance Computing and Model Driven Engineering.