Efficient on-chip communications for data-flow IPs

Antoine Fraboulet, Tanguy Risset

Efficiently on-chip communications for data-flow IPs

Antoine Fraboulet
Citi, Insa-Lyon,
21 av. Jean Capelle
69621 Villeurbanne Cedex
France
antoine.fraboulet@insa-lyon.fr

Tanguy Risset
Inria, Lip, ENS-Lyon
46 Allée d'Italie
69363 Lyon cedex 07
France
Tanguy.Risset@ens-lyon.fr

Abstract

We explain a systematic way of interfacing data-flow hardware accelerators (IP) for their integration in a system on chip. We abstract the communication behaviour of the data flow IP so as to provide basis for an interface generator. We also explain which parameter this interface generator has to take into account. We validate our interface mechanism by a cycle accurate bit accurate simulation of a SoC integrating a data-flow IP.

Keywords: system on chip, SoC simulation, High level synthesis, interface generation

1. Introduction

Large scale systems on chip (SoC) design is faced with many problems among which efficient communication managing is one of the most important. In the past, communication efficiency has already been one of the major burden in parallel machine development but in addition, today's on-chip communications are limited by the power consumption problem. Network on chip based SoC architectures are not arriving as fast as envisaged because of the power consumption problem. In addition, shorter design time is required to meet the ever increasing time-to-market constraints. IP re-use and platform based design are pointed out as solution to faster design but they are still confronted with important limitations and are not really used in today's industrial design flows. Examples of these problems are: IP standardisation, system on chip cycle accurate simulation time [15, 14], and fast integration of special purpose hardware in a design environment.

We address the problem of providing a rapid and efficient integration of a special purpose hardware accelerator into a complex system on chip possibly integrating many processor cores and important software applications. More precisely, we will focus on the category of *stream processing* hardware accelerators. Stream processing performs on-the-fly costly computations on streams of data. The amount of control in these computations is reduced, but the complexity lies in the quantity of computations usually submitted to soft real time constraints (as for audio or video processing). Hardware accelerators containing parallel computation usually presented in the form of arrays of processors are mandatory to meet

these constraints. we refer to these accelerators as *data-flow* IPs. Systolic arrays are examples of such data-flow IPs but they are not the only ones. In this paper we only target linear array of processor which have a small number of input/output ports, the problem of interfacing 2D array of processors has not been studied yet.

The hardware accelerators used in SoC for portable communication systems (cell-phones, PDA,...) are originally designed by hardware designers in collaboration with signal processing engineers. For these IPs, the performance bottleneck lies in the communication of the data between memory and the IP. The paradigm used for specifying and designing these circuits is called the *data flow* model: designers manipulates streams of data which input and output the circuits without any information about the external storage of these data. The interface protocol is *data synchronised* which means that the behaviour is regulated by the consumption and production of data at the boundaries of the architecture. In this protocol, if the data is not present the computations are stopped, usually with a clock enable mechanism.

Until recently, these IPs where designed by hand with a precise methodology, knowing in advance the target SoC architecture. Hence the design of the interface was more or less performed together with the architecture. New trends in SoC design require the use of tools to accelerate the design phase of specific hardware accelerators. These tools are called *high level design*, *behavioural synthesis* tools or *hardware compilers* [6, 17, 16]. Most of the time, because of the huge design space they propose to the user a toolbox for tuning parameters of the resulting architecture. However, even if these tools greatly accelerate and secure the design of the IP itself, the time needed to write the interface may render the tool simply useless. Today, these tools produce some configuration information that helps in the design of the interface, but this part has not standardised. In this paper, based on the experience of MMAlpha [4, 5] and Gaut [17], we identify the common concepts that are used by data-flow IP designers to propose a generic interface mechanism. This generic interface mechanism will be parameterised by the configuration information output from data-flow IP generators. We also hope to bring a standard way of describing data-flow IP input/output behaviour so as to facilitate IP reuse.

We introduce in this paper some theoretical foundations for the automatic generation of data-flow IPs interface. As a reference to data-flow *model of computation* used to build IPs, we introduce the data flow *model of interface* which basically rely on a data synchronised protocol. We highlighted the various parameters that must be taken into account to adapt the data-flow IP to various SoC platform and we validate this high level interface design concept by a complete cycle accurate SoC simulation using the SocLib simulation environment and a high level designed hardware accelerator synthesised with the MMAlpha tool [6].

2. Hardware/Software interface of data-flow IPs

In this section, we precisely identify the context of the work, it implies precise definition of what we call data-flow IP, the schematic architecture of the system on chip we target and the overall principles of the hardware/software interface that we wish to generate.

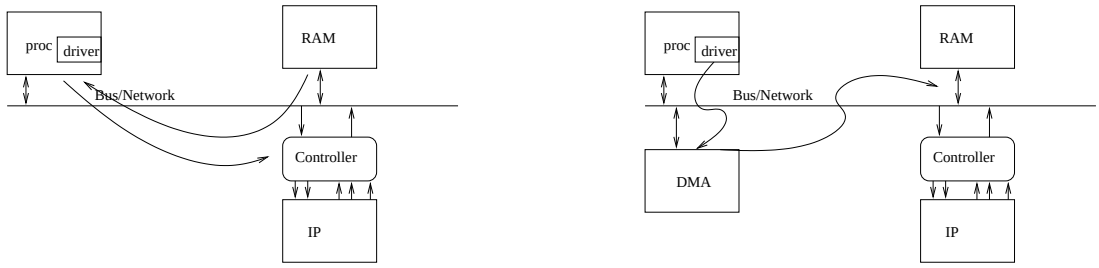


Figure 1. The two main interface modes: with or without use of a DMA module

2.1. Data flow IP and SoC environment

In recent SoC design, IP reuse is gaining much importance. Many processors IPs are to be connected on a communication medium which can be a bus, a hierarchical bus or a Network. We will refer to this communication medium as the *bus* even if this work can be applied to network on chip based SoC architecture. These processors are usually *initiators*, in the sense that they initiates communications. In addition, several other IPs are connected on the bus: memory, hardware accelerators, external communication systems, bridge to other buses and so on. In most recent SoC architectures the only hardware to be designed specifically will be some hardware accelerator dedicated to a particular algorithm to be executed on the SoC.

From the external communications point of view, a data flow IP is a black box controlled with a clock which receives and sends data possibly at each clock cycles. It has a number of input ports and output ports each of which having a certain bit-width. We assume that, in addition to the clock, the IP has a *clock enable* pin that can freeze the execution of the IP. Hence, if the clock enable is not set, everything behave in the IP as if the clock was not changing. This allow us to define the notion of *virtual clock*, the virtual clock is the one seen inside the IP, it does not take into account the clock changes that are not validated by the clock enable. Then, we assume that the IP is data synchronised, i.e. at each clock cycle, data are presented on the input port and at the raise of the clock (provided that the clock enable is set), the data is read by the IP and the virtual clock counter can be incremented. We do not assume that there is a hand shake protocol at this level of abstraction. In practice there is be a hand shake protocol at a lower level as in the VCI interface protocol for instance, but it can be abstracted thanks of the virtual clock mechanism.

2.2. Hardware/Software interface global scheme

In general we assume that the hardware accelerator will be controlled by a processor (that we call the *host* processor). Hence the interface of the IP is usually composed of a software part and a hardware part. The software part will command the data communication between memory and the IP, we call this part the *driver*. From our experience [4, 5], the communication scheme used to feed the hardware accelerator must respect important constraints so as to obtain acceptable performances: (i) it should try to use a Direct Memory Access module (DMA) to perform communication in burst mode, but only when it increases performances; (ii) the designer can set the burst size to balance between hardware cost,

communication latency and bus contention. Also, the architecture of the hardware/software interface must be re-usable for many different data-flow IP. It can be envisaged in two main modes depicted in Fig. 1: either the driver on the host processor directly executes all the communications or it simply controls a DMA which himself performs the communications between the memory and the IP. The use of this DMA can be made much more efficient if the DMA is designed specifically to this generic interface control mechanism.

The proposed interface mechanism must be used in various SoC architectures, hence we think that an *interface generator* tool must be used to generate various versions of the interface depending on parameters. The two modes for the interface is one example of such parameter, we review others in the next section. The generated interface must be parameterisable, in the sense that it should be able to handle various size of the streams sent to the IP. We propose a classification of all these parameters and identify which should be fixed and which should stay as a configurable parameters in the generated interface.

3. Generic Interface generation principles

In this section we identify the concept that are useful to build an interface generator for data-flow IPs and we explain the parameters that the interface generator must take into account.

3.1. Communication activity abstraction

The crucial point is to abstract the IP communication activity in such a way that it is (i) platform independent (ii) compact. To be platform independent, one has to identify the communication activity that depends only of the IP. To be compact one must use loop like formalism to express a large number of communication in a very small space. The use of parameters not known at compile time in these loop is mandatory. The theoretical background of our work is based on the polyhedral model which uses polyhedra to abstract loop iteration domains. However, interface synthesis for linear arrays does not need a important knowledge on polyhedra as the only polyhedra manipulated are 1-dimensional.

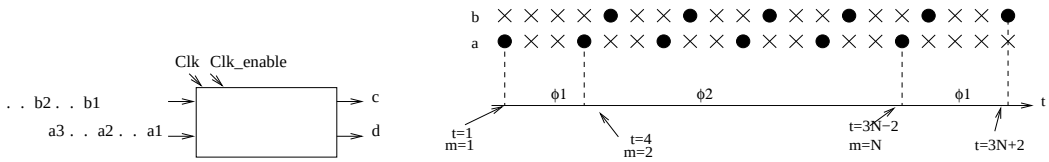


Figure 2. Graphical representation of input/output on a simple data-flow IP

Consider a very simple example: a hardware module that processes two streams $\mathtt{a}_1, \mathtt{a}_2, \ldots$ and $\mathtt{b}_1, \mathtt{b}_2, \ldots$ (only the input interface is detailed here). The `a` stream is 16 bit wide while the `b` stream is 8 bit wide. This IP inputs the first `a` sample at $t = 1$ and then one `a` sample every three clock cycle until N samples have been processed. The `b` stream is input at the same rate (one sample every 3 clock cycles) but starts at $t = 4$. The t counter corresponds to a counter on the virtual clock of the IP hence we are currently observing the behaviour

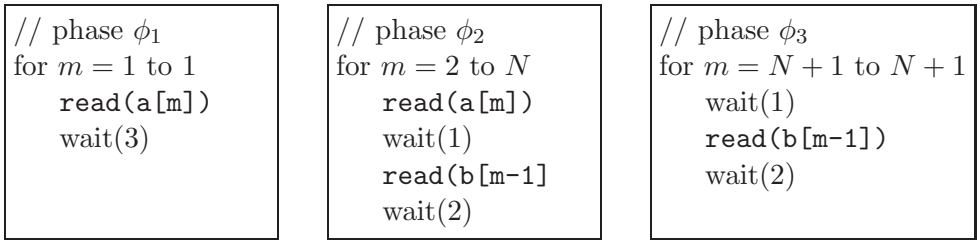


Figure 3. Data flow interface format used for specifying input/output behaviour of the IP of Fig. 2

from *inside* the module. The IP is represented on the left of figure 2, many signal processing filters have the same kind of input/output behaviour. The schedule of the inputs can be graphically represented by the right of Fig. 2, for $N = 6$.

The input interface behaviour has three *phases*. During the first phase, only `a` is input, then `a` and `b` are input, and finally there is a phase where only `b` is input. Because of the stream processing character of the application, we can divide each of these phases into the repetitive execution of a *motif* which correspond to the consumption of one *sample* (see right of Fig. 2). The index that can be used for specifying functional behaviour of the IP is the motif index `m` rather than the virtual clock t. Of course, as we are in the data flow model, there will always be a relation between the motif index `m` and the virtual clock counter t. For our example, this relation can be expressed as: in phase ϕ_2 the motif `m` start its execution at $t = 3\mathtt{m} - 2$. In addition we need a way to identify which data is concerned with a particular motif `m`. For that we assume that data input on a particular port are successive elements of a linear array. Any other indexing is possible, however, it should be as close as possible as the memory layout of the corresponding variable in a target SoC.

We gather the phase and motif information into the *data-flow interface format* presented in Fig. 3 which permits to represent the behaviour of the architecture of Fig. 2 inputs without assigning a value to N. In these loops, the `m` index is the motif index or equivalently the number of the sample treated. The virtual clock counter t can be reconstructed using the `wait` instructions. The instructions have a 0 cycle execution time except the `wait()` statement whose execution time is the argument. The phases and motifs completely define the communication behaviour of the IP. Any IP whose input/output behaviour cannot be expressed with these concepts cannot be interface by our mechanism, Note that it strictly imply data-independent communications, the communication scheme must be statically defined.

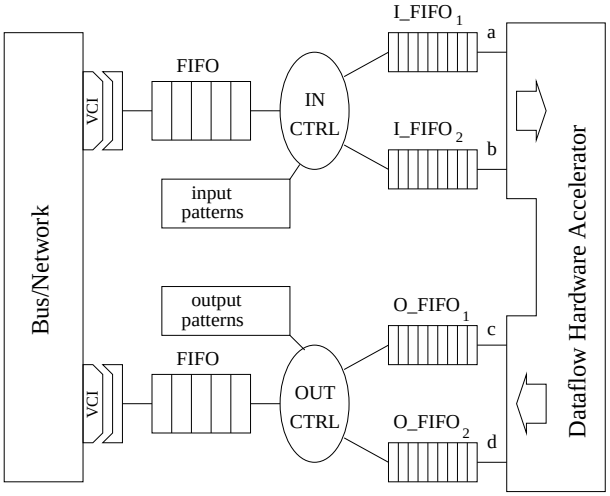
To be able to integrate this IP into a SoC, we need some more information, in particular the size of the bus and the size available to buffer data just before the IP. we define the communication scheme by a succession of *pattern* inside each phase: a *pattern* is the set of data that will be sent to the architecture to feed the architecture during a particular phase. A pattern can gather several motifs in order to take advantage of the bus burst mode. Input pattern and output pattern must be carefully merged taking into account buffering available to avoid deadlocks. On our example, during phase ϕ_2, if we have FIFOs containing 20 data of `a` and `b` before IP input ports, and assuming that the bus width is 32 bits, we can use the following pattern: send 10 bus-words of `a`, then send 5 bus-words of

data `b` (remember that `b` has a twice smaller bit width).

To sum up the important concepts that we have introduced in this section:

- The input/output behaviour of the IP is divided into a finite number of *phases*. A phase is a period during which communication occurs only on (possibly many) fixed ports.
- A phase repeats from one to a very large number of time (possibly fixed at run time) a *motif*. A motif has a fixed number of virtual clock cycle. It describes the repeated behaviour of the the input and output of the interface during a particular phase.
- A *pattern* (or communication pattern) is the set of data that will be sent to the architecture to feed the architecture during a particular phase.

3.2. Interface synthesis




```
// input phase 1
SEND(a[0],1)
for(m=1; m < N; m+=20)
  {
    // input phase 2
    SEND(a[m],10);
    SEND(b[m-1],5);
    // RECV instructions
    [...]
  }
// input phase 3
SEND(b[N],1);
//  RECV instructions
[...]
```

Figure 4. Hardware and Software interfaces for the Data-flow hardware accelerator of Fig. 3. Controllers must be configured according to communication patterns used in the software driver on the right hand side.

We are now able to introduce the basic architecture of our interface mechanism. It is represented on Fig. 4 with a maximum burst size of 10 bus words, assuming that 20 divides `N`. The hardware part is on the left. Data received from the host processor through the bus are de-multiplexed according to the pattern informations of the current phase. The hardware performing this control is a simple automaton parameterized by phase and pattern information, we call it the *controller*. The driver executed on the host processor is represented on the right. One can see how the patterns are used to write the driver (only input instructions are detailed). The compatibility between the controller configuration and the driver is essential, both should be generated by the same tool. This is the major argument for an interface generator, in the following we detail the generation process that we promote.

The interface generator reads input/output behaviour of the IP expressed in a particular format such as the data-flow interface format presented in Fig. 3 and produces the corresponding hardware controller and software driver. There are two difficulties, first the generator should be able to take into account many parameters coming from the characteristics of the target SoC architecture, second the interface should be efficient and in particular should use the bus burst mode but only if it really improve performances.

The first type of parameter are called *application dependent* parameters. This includes the IP dependent parameters which do not influence the hardware of the interface, this is precisely what has been presented in previous section: phases and motifs for the application considered on the IP. It also contains parameters that are dependent on the particular implementation of the application as maximum size of burst or performance expected (throughput, power consumption) that might influence the choice made by the generator. From these application dependent parameters the interface generator will generate the communication patterns taking into account the other parameters presenter hereafter.

The second type of parameter are called *structural parameters*, they reflect the different architectural choices made in the target SoC that may influence the form of the interface. It includes:

- bus interface parameters, in our implementation it correspond to VCI parameters (cell size, address size, ...);
- IP hardware dependent parameters: number of port, bit width of each port, reset mechanism;
- software parameters: target assembly langage characteristics, interruption handling;
- interface type, with or without DMA as represented on Fig. 1;
- controller hardware parameters: size of the FIFOs and size available to store pattern information (see Fig. 4);
- memory layout parameters: data storage organisation mode.

The first four of these structural parameters can easily be taken into account by changing the syntax of the code generated. The difficult part is to make the phase/motif information and the memory layout organisation compatible. In our version, we have assumed that successive data that enters a particular input port of the IP are stored contiguously in the memory. If this is not the case, as for instance for interleaved data in stereo signal processing, and if we still want to use burst access to memory, the interface must merge this information with the pattern information to program the controller. We think that this is very difficult to do in a generic way, hence we propose to generate an interface for a fixed memory layout organisation of each stream, the most common case being the in-order data layout organisation.

4. Experiments: cycle accurate SoC simulation

A simple version of this interface mechanism has been prototyped in the SocLib simulation environment combined with a data-flow IP generated by MMAlpha. The simulation

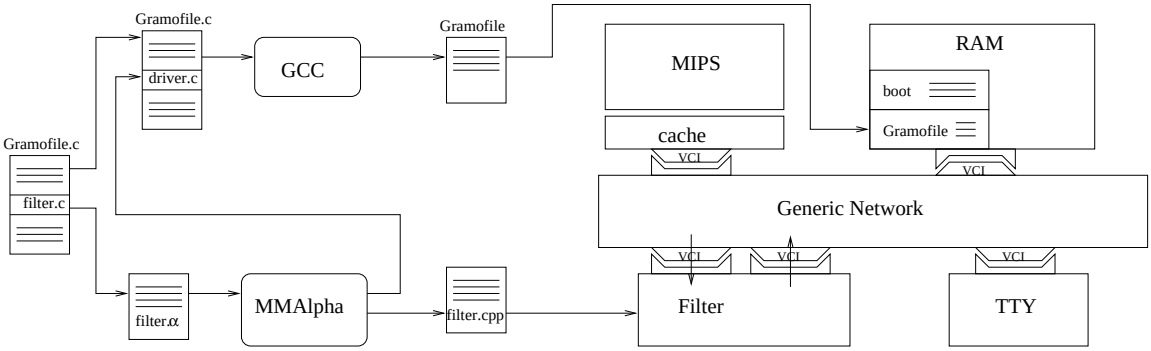


Figure 5. The SoC simulated and the global design methodology. Blank square box represent C++ simulation models of IPs.

is performed in SystemC at cycle accurate, bit accurate level. We briefly detail the implemented SoC and we give some performance results.

4.1. Target platform and design methodology

The main component of SocLib (`http://soclib.lip6.fr/`) is currently a set of SystemC simulation models for common IPs (MIPS processor, RAM, NoC, busses, DMA). These simulation models are publicly available and there exists synthesizable RTL versions of the IPs that can be used for the final design. MMAlpha [6] is a toolbox for designing regular parallel architectures (systolic like) from recurrence equation specifications expressed in the Alpha language. It is one of the only existing tools that really automates the refinement of a software specification downto RTL description within the same language: Alpha. We have developed a translator from Alphard (hardware description language, subset of Alpha) to SystemC [5]. We wrote by hand a simple version of the generic interface presented in this paper and we generated from MMAlpha the configuration of the controller and the software driver for any architecture generated by MMAlpha.

We have chosen a classical linux audio signal processing application called *Gramofile*. Gramofile processes audio files (.wav format) and proposes various simple filters like LP's tick removal. We have extracted a simple filter (referred as `filter.c`), and translated it in Alpha by hand. The translation was validated by replacing the original `filter.c` file by the C code generated from the `filter.alpha` by MMAlpha. Then MMAlpha generated a systolic version of the filter from which we generated a SystemC simulation model (`filter.cpp`), a SystemC file for configuring the hardware interface and the software driver. The software driver replaced the original `filter.c` in the gramofile program before its compilation to the MIPS processor. This driver explicitly performed all the communications between the memory and the hardware accelerator as seen on the right of Fig. 4.

The target platform chosen is represented on figure 5, The hardware components of the platform are: a MIPS R3000 processor (with its associated data and instruction cache), a standard memory, a component used for displaying output (referred as TTY) and a specific hardware accelerator generated with MMAlpha. All these components are connected via VCI ports to a simple network (internal architecture of this network is not precisely simulated, only the latency and bandwidth can be parameterised). The software running on

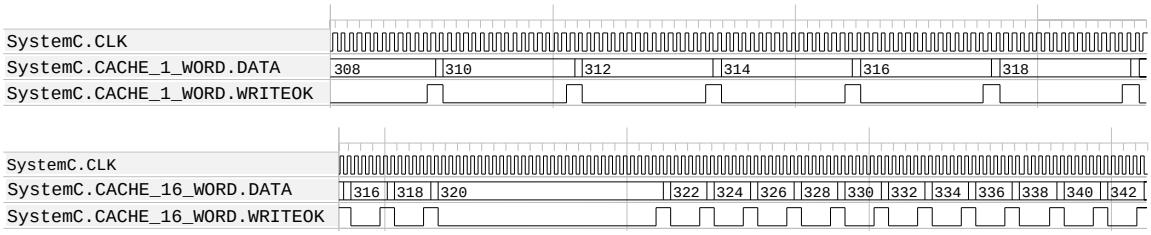


Figure 6. Effect of different size of cache lines (1 bus word and 16 bus word) on the arrival of data to the IP.

the MIPS, in addition to bootstrapping information, is composed of the Gramofile program cross-compiled with GCC to a MIPS target. In this experiment the interface does not use a DMA: communication are executed from the MIPS. Another restriction is that the hardware interface configuration is fixed, we did not implement the dynamic configuration of the interface yet.

4.2. Simulation and performance results

A previous work [4] had experimented a VHDL implementation of a simple hardware interface and shown that the hardware of the controller was not very important, most of the hardware complexity will be spent into the various FIFO used. We want now to validate the efficiency of the resulting communications occurring on chip. The SoC simulation results are presented in the table 1.

complete simulation time	29.21 s
complete simulation cycles	600000 cycles
simulation speed	20540.9 cycles/seconds
hardware pipeline throughput	20450 samples

Table 1. Performances obtained from the complete SoC simulation

In this experiment the hardware is not used as much as it could be. Indeed, one sample is treated every 30 clock cycles while the theoretical bound deduced from the bit width of input/output for one sample is one sample every 2 clock cycles. We have analysed the causes and it appears that the MIPS access to the memory was the main reason of this bottleneck. First, as we did not use a DMA to access memory we have doubled the amount of communication: memory to MIPS and then MIPS to IP. Second, the MIPS cannot access the memory in burst mode, this result in a data transfer approximately every 15 clock cycle. The combination of these factors ($2 \times 15 = 30$) explains the inefficiency.

The cycle accurate simulation permits to determine precisely which parameter influence the communication efficiency. For instance, in Fig. 6, we have shown the influence of cache line size on the frequency at which data could arrive to the IP. The use of cache line of 16 bus word increases the throughput and the effect of a cache miss can be clearly visualized. With cache line reduced to one bus word, the data arrive regularly but slower. On the other

hand it can be shown that cache size does not influence performances on this application as cache are direct-mapped. The results presented here clearly indicate that efficient handling of burst mode through a DMA module is a crucial issue to obtain an efficient use of the hardware accelerator.

5. Related work and Conclusion

Many recent works present SoC simulation environment [13, 2, 9]. These approaches do not have an Open Source policy as the one of SocLib. The main concern is the acceleration of the simulation at various level of precision: transaction level or cycle accurate level, these works do not propose generic interface mechanism for IPs. Some attempt have been made to abstract communication behaviour at high level [3] or at low level [19, 18, 1], but none of these approaches make the assumption that the IP are in the data-flow model. Many of the interesting results in this field are based on the work of the Ptolemy [12] project and especially the introduction of various computation models for data-flow IPs: process network [11] or SDF [20, 10]. An interesting theory has been developed for multi-rate IP-based systems [7], but the problem solved is the interconnection between several data-flow IP while we specifically target a single data-flow IP controlled by a host. Several high level design tools are now clearly identified either as research prototypes or industrial products [6, 17, 16, 8, 11], each of these tools have implemented ad-hoc communication protocol between the generated data-flow IP and the host. We expect our model to be usable at least by MMAlpha and Gaut.

In this paper we have delimited a class of architecture that is subject to high throughput requirement and restricted enough to allow an interface mechanism which is generic and efficient. The notion of phase, motif and pattern can be defined for many data flow IP and are particularly useful for high level design tools that can use them to generate hardware and software interfaces together with the IP. The cycle accurate simulation environment that we propose ensures that future development will result in real, not just theoretical, improvements. These experiments also validate the feasibility of connecting a design compiled by the MMAlpha tool in a SoC environment.

References

[1] N. Banerjee, P. Vellanki, and K. Chatha. A power and performance model for network-on-chip architectures. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[2] M. Bolado, H. Posadas, J. Castillo1, P. Huerta1, P. Sánchez1, C. Sánchez, H. Fouren, and F. Blasco. Platform based on open-source cores for industrial applications. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[3] V. Chandras, A. Xu, H. Schmit, and L. Pileggi. An interconnect channel design methodology for high performance integrated circuits. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[4] S. Derrien, A. C. Guillou, P. Quinton, T. Risset, and C. Wagner. Automatic synthesis of efficient interfaces for compiled regular. In *Internationnal Samos Workshop on Systems, Architectures, Modeling and Simulation (Samos)*, Samos, Grece, July 2002.

[5] A. Fraboulet, T. Risset, and A. Scherrer. Cycle accurate simulation model generation for soc prototyping. Technical Report 2004-18, LIP, ENS-Lyon, May 2004.

[6] A.-C. Guillou, P. Quinton, T. Risset, C. Wagner, and D. Massicotte. High level design of digital filters in mobile communications. Technical Report 1405, Irisa, June 2001.

[7] J. Horstmannshoff, T. Grotker, and H. Meyr. Mapping multirate dataflow to complex RT level hardware models. In *Int. Conf. Application Specific Systems, Architectures (ASAP)*, pages pp. 283–293., 1997.

[8] B. Hounsell and R. Taylor. Co-processor synthesis a new methodology for embedded software acceleration. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'03 Designers' Forum)*, Paris, France, March 2004.

[9] H. Jang, M. Kang, M. Lee, K. Chae, K. Lee, and K. Shim. High-level system modeling and architecture exploration with systemc on a network SoC: S3C2510 case study. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[10] H. Jung, K. Lee, and S. Ha. Efficient hardware controller synthesis for synchronous dataflow graph in system level design. In *ISSS*, pages 79–84, 2000.

[11] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, 2000.

[12] E. Lee et al. Overview of the ptolemy project. Technical Report UCB/ERL No. M99/37, University of California, Berkeley, july 1999.

[13] M. Loghiy, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[14] D. G. Pérez, G. Mouchard, and O. Temam. A new optimized implementation of the systemC engine using acyclic scheduling. In *Design, Automation and Test in Europe Conference and Exhibition (DATE'04 Designers' Forum)*, Paris, France, March 2004.

[15] F. Pétrot, D. Hommais, and A. Greiner. A simulation environment for core based embedded systems. In *Annual Simulation Symposium*, pages 86–91, Atlanta, GA, U.S.A, April 1997.

[16] R. Schreiber et al. High-Level Synthesis of Non Programmable Hardware Accelerators. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, Boston, July 2000.

[17] O. Sentieys, J. Diguet, and J. Philippe. Gaut: a high level synthesis tool dedicated to real time signal processing application. In *EURO-DAC*, Sept. 2000. University booth stand.

[18] C. Shin, Y. Kim, E. Chung, K. Choi J. Kong, and S. Eo. Fast exploration of parameterized bus architecture for communication-centric SoC design. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[19] V. D. silva, S. Ramesh, and A. Sowmya. Synchronous protocol automata: A framework for modelling and verification of SoC communication architectures. In *Design, Automation and Test in Europe Conference (DATE'04)*, Paris, France, March 2004.

[20] M. Williamson. *Synthesis of parallel hardware implementations from synchronous dataflow graph specifications.* PhD thesis, University of California, Berkeley, CA, 1998.