



# Dataflow Programming Model For Reconfigurable Computing

L. Gantel<sup>\*†</sup> and A. Khiar<sup>\*</sup> and B. Miramond<sup>\*</sup> and A. Benkhelifa<sup>\*</sup> and F. Lemonnier<sup>†</sup> and L. Kessal<sup>\*</sup>

<sup>\*</sup> ETIS Laboratory – UMR CNRS 8051  
University of Cergy-Pontoise / ENSEA  
6, avenue du Ponceau  
95014 Cergy-Pontoise, FRANCE  
Email {firstname.name}@ensea.fr

<sup>†</sup> Embedded System Lab  
Thales Research and Technology  
1, avenue Augustin Fresnel  
91767 Palaiseau, FRANCE  
Email {firstname.name}@thalesgroup.com

**Abstract**—This paper addresses the problem of image processing algorithms implementation onto dynamically and reconfigurable architectures. Today, these Systems-on-Chip (SoC), offer the possibility to implement several heterogeneous processing elements in a single chip. It means several processors, few hardware accelerators as well as communication mediums between all these components. Applications for this kind of platform are described with software threads, running on processors, and specific hardware accelerators, running on hardware partitions. This paper focuses on the complex problem of communication management between software and hardware actors for dataflow oriented processing, and proposes solutions to leverage this issue.

**Index Terms**—reconfigurable computing, dataflow programming, hardware actors, real-time operating systems, multiprocessor architectures, FPGA, image processing

## I. INTRODUCTION

Heterogeneous Systems-on-Chip (HSoC) platforms provide good performances but are often difficult to program. For programming efficiency in an industrial context, it would be better if the developer can describe its application in a single language, independently from its future mapping. In this way, we need a homogeneous API (*Application Programmable Interface*) and specific operating system services to manage hardware accelerators with so much flexibility as what is done with software threads. Using Dynamically and Partial Reconfiguration (DPR) capabilities of the latest Xilinx<sup>©</sup> FPGAs, and the future Altera<sup>©</sup> FPGAs, we pose the problem in the context of dynamic hardware threads.

The needed homogeneous API is defined in the existing literature as middleware. The most popular definition of a middleware is the following: "The middleware is the software that assists an application to interact or communicate with other applications, networks, hardware, and/or operating systems. This software assists programmers by relieving them of complex connections needed in a distributed system. It provides tools for improving quality of service (QoS), security, message passing, directory services, etc. that can be visible to the user" [1]. Middleware provides a higher-level abstraction layer for programmers than API provided by the operating

system, such as sockets.

It reduces significantly the work of application programmers by relieving them of tedious and error-prone programming. Middleware is designed to mask some of the kinds of heterogeneity that programmers of distributed systems must deal with [2]. It always masks heterogeneity of networks and hardware processing units.

We are interested in the FOSFOR project, by providing communications on the platform, whose components are heterogeneous with dynamic hardware. This is why we set up a communication protocol between these components. It is important to note that, to have a transparent, flexible and scalable platform, its global communication medium must be defined and take into account this heterogeneity.

In this article, we propose to follow up the properties of a data-flow model of computation dedicated to the heterogeneous Systems-on-Chip from the actor-level down to the thread implementation. An overview of state of the art is given in section II. Section III details the design flow used to mitigate the complexity of the platform. Section IV describes the hardware actor model specified to be implemented at a lower level. The basis of the communication framework are explained in Section V. Section VI is about first experiments and results obtained with a real application.

## II. RELATED WORK

Embedded systems interact with the physical world. Most of these systems include heterogeneous designs that are specific to an application. Conceptually the distinction between the software and hardware implementations, in a computation point of view, is relative to their degree of concurrency and their performance limitations. The issue lies in the manner to identify the appropriate abstraction for representing the global system.

Among the dataflow design approaches, the Actor-oriented paradigm offers the abstraction to rise above the heterogeneity. This approach orthogonalizes component definition and component composition. A tool that refines heterogeneous actor-oriented models into software-based design and hardware-based program-level description is proposed in [3].

Several other data-flow oriented models of computation exist to design heterogeneous systems, but there is a lack in the partitioning of the application into a reconfigurable platform. We propose in this paper to reduce the gap between the high-level properties of the input specification and the final implementation onto a partially and dynamically reconfigurable architecture by defining the notion of reconfigurable software and hardware actors.

Verdoscia et. al. [4] already introduced a model of hardware actor, defined in a dataflow graph model, and implemented in hardware. Each actor was described as a Functional Unit (*FU*), executing simple operations such as additions, multiplications, loops and conditional instructions. As in the upper level of its model, the execution of this FU was conditioned by two input tokens and fired one output token. FUs are organized in clusters whose communication is based on Message Passing through shared memories. However, FUs inside the same cluster are connected together with a crossbar to improve latency.

A model of computation for the reconfigurable computing involves the definition of a reconfigurable dynamic hardware actor. By reconfigurable, it means a thread, implemented in hardware, which has a similar behaviour to a software thread.

To attack this issue, some operating systems which are able to manage both software threads and hardware accelerators have been developed. This is the case of BORPH [5], a Linux based operating system. It offers kernel drivers to control the execution of hardware accelerators. Santambrogio et. al [6] also present a Linux driver to control ICAP port of the FPGA. Thus they can dynamically reconfigure hardware accelerators by writing commands in well-known IOCTL registers.

Much complex accelerators have then been developed, such as Hybrid Thread [7]. In this article, the authors define a model of POSIX compliant hardware thread, capable of processing operating system calls through a shared memory, as software thread do. In the same way, ReconOS [8] authors present their own model, which achieves system calls via an API composed of VHDL procedures. In both cases, thread behaviour is described by a finite state machine for thread state execution, and the system calls are processed by a software proxy thread, running on a distant processor.

The drawback of this approach is that it leads to important time overheads. So Hybrid Thread authors realized a hardware implementation of the synchronization and scheduler services in order to lighten processor load [9]. This solution was also experimented in [10] where  $\mu\text{C}/\text{OS-II}$  was extended with hardware implementation of few services. At the expense of some hardware resources, this can lead to more efficient platforms and takes advantage of the FPGA parallelism capacities.

When programming heterogeneous platforms, differences between hardware and software threads have to be hidden, so a virtual layer has to be added. DNA Operating System [11] proposed a component-based system framework. Components are described to manage hardware dependent issues such as endianness, multiprocessor management, memory allocation, synchronizations and exceptions, or task context management.

The proposed interface has been developed in order to provide homogeneous interface for all applications. Maejima et al. [12] presented a hardware module which permits segregation between critical and non-critical services to make application safer. Petrot et. al [13] addressed the same issue and proposed a POSIX API running on top of the Mutek operating system. This layer called *Hardware dependent Software* offers a homogeneous API for all the processing elements in the platform.

### III. DATA FLOW PROGRAMMING MODEL FOR RECONFIGURABLE COMPUTING

#### A. Design flow

Today, embedded systems face increasingly higher performance requirements. Deeply pipelined processor architectures are being employed to meet desired system performances and system architects critically need modeling techniques to rapidly design the required platform. So, we propose to reduce the gap between the initial system specification and the final implementation, especially when targeting dynamically reconfigurable architectures. The main idea is to conserve the semantic of the components manipulated at high-level when refining the model till the hardware level. The concept, that already exists in the pure software domain with middleware approaches, has been extended to also consider hardware components.

Thus, the management of reconfigurable architectures is not limited to a specific technical solution anymore but is considered as a whole from the top to the bottom. A specific abstraction layer is developed (*a hardware and software middleware*) in relation to the semantic of the high-level components. This layer helps to map the high-level interactions between the actors of a specific application onto the abstracted platform. It also facilitates rapid exploration of candidate architectures under given design constraints such as area, clock frequency, power, and performance. Finally, it allows to foresee the future automatic generation of actor codes.

As illustrated in *Fig. 1*, we consider, from high level, a coherent and interoperable vision of communicating actors. This abstraction of communications facilitates the number of possible refinements into a subsequent abstraction layer (Programs) in the design flow.

To achieve this goal, the semantic of the managed components is based on the Synchronous DataFlow (*SDF*) model of computation (*MoC*) [14]. Actors description is made from the standard graphical language UML (*Unified Modeling Language*) and their interfaces are described using IDL3 (*Interface Language Description*).

At the higher level we represent each block of the application that performs a specific task by an actor which, receives and sends data through virtual channels. Those actors share the same API of the global virtual platform.

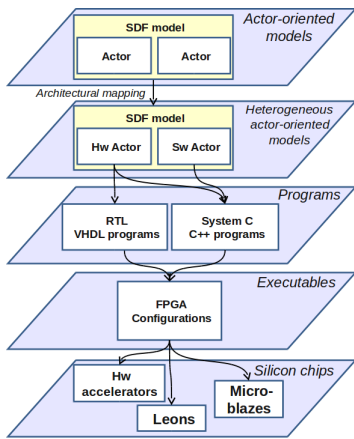


Fig. 1. Design flow for reconfigurable data flow applications

### B. Communication semantic

We want to virtualize the communications into a distributed heterogeneous embedded Multiprocessor System-on-Chip (MPSoC) platform containing reconfigurable hardware computing units. We propose a unified management of the communication, implemented both in software and in hardware, that provides the designer a single programming interface, accessible from the abstract actor layer. The global system is then controlled in a distributed manner by several communication managers implemented on each execution unit: processors and hardware accelerators (see section IV for more details).

Communication can be respectively homogeneous or heterogeneous. The first type is related to communication between two actors implemented simultaneously in the hardware domain ( $Hw / Hw$ ) or in the software domain ( $Sw / Sw$ ). The second type concerns communication between two actors implemented separately in two different execution domains: ( $Hw / Sw$ ) and ( $Sw / Hw$ ).

To ensure synchronization and exchange between the elements of a platform, two types of transfer are needed (Fig. 2):

- i) data exchange : each time a calculation block finishes its processing on the input data, this data can be passed to the next block for another type of calculation according to the principle of synchronous dataflow transfer.
- ii) state coherency : control information should be exchanged between the communication managers (see section V) in order to maintain a permanent consistency of the actors location and states, even during reconfiguration.

## IV. RELOCATABLE HARDWARE ACTOR

### A. Target architecture

The FOSFOR project [2] (*Flexible Operating System FOR Reconfigurable platform*), aims to define a new kind of execution platform. This platform (Fig.3) targets heterogeneous applications in the sense that threads could be either software (running on one of the processors), or hardware (running as

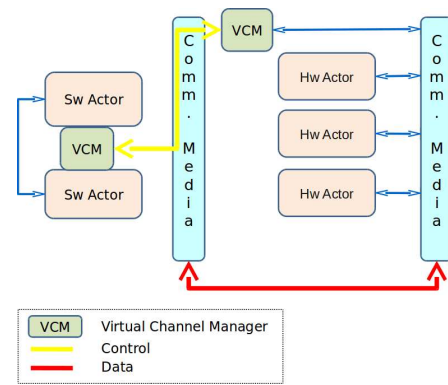


Fig. 2. Communications into the platform

an accelerator in a partition of the FPGA). At this level, the communication between the actor and the rest of the system is managed by using a system call paradigm. An actor can then be associated with the notion of Thread.

In the context of this project, hardware threads are managed by a hardware operating system (*Flexible HwOS*). It takes advantage of the DPR to schedule hardware threads on the available partitions, and implements similar services in hardware to those offered by the software operating system. Existing services include thread management, semaphore counters and mailboxes. The flexible HwOS allows hardware threads to process fast system calls without the need to pass through a software proxy thread [8].

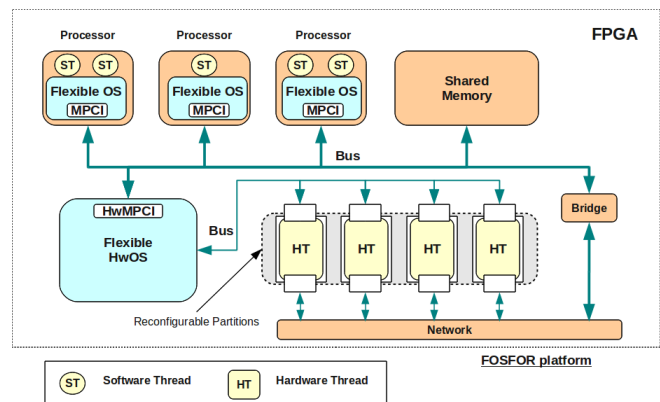


Fig. 3. FOSFOR Architecture

Software threads are managed by the RTEMS operating system (OS) [15]. This OS allows to abstract heterogeneity using its MultiProcessor Communication Interface (MPCI). Starting from this point, FOSFOR also implements this interface and permits hardware threads to access to any services in the system, both software and hardware, in a transparent way. Moreover, FOSFOR provides virtual channel services which abstract inter-thread communication and memory management (see Section V).

## B. The generic hardware actor

A hardware actor has been defined to abstract dynamic reconfiguration provided, for instance, in the Xilinx<sup>©</sup> FPGAs. Thus, it is composed of two main parts : a static part which contains all the interface with the rest of the system, and a dynamic part, specific to the actor which contains the Finite State Machine (*FSM*) controlling its execution, its accelerator and its private memory (*Fig. 4*).

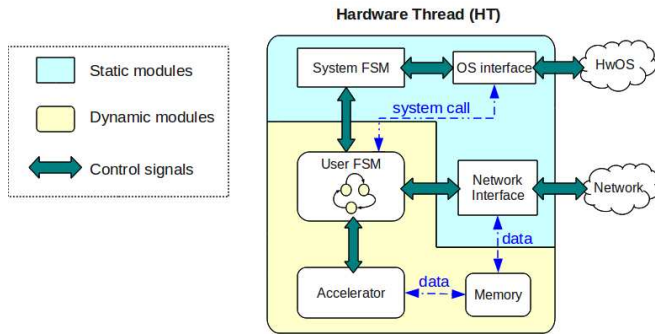


Fig. 4. Hardware Thread Architecture

The static part provides an interface to communicate with the operating system (*OS Interface*). It is composed of a standard double port memory in which the thread can write the identifier of a system call and its parameters. The call is read by the operating system via a dedicated bus connecting all the reconfigurable partitions (*Fig. 3*). Once the system call done, returns values are written back in the memory and are read by the thread. This static part also includes a Network Interface and allows the thread to communicate with the external world through a dedicated Network-on-Chip (*NoC*) [16].

Once instantiated by the operating system, the hardware actor is able to execute four basic commands : start, suspend, resume and stop. As the actor is located into a dynamically reconfigurable partition, its state (from an OS point of view) has been specifically defined as described in the finite state machine of *Fig. 5*. As illustrated in this figure, a hardware actor is either preemptible or non-preemptible. In the latter case, the scheduler can not suspend or move the actor to another place. To come back to a preemptible state, the actor notifies it explicitly (*blocking system call or call to a specific primitive*).

The main advantage of a hardware component is its parallel nature. In order to be consistent with the SDF computation properties of the high-level specification (*see Section III*), we have added the architectural mechanisms that allow the actor to support software pipelining. According to a buffer management by the User FSM (*Fig. 4*), a hardware actor would be able to process three actions in parallel:

- i) Receive data from the network
- ii) Compute data stored in its private memory
- iii) Send data to the network

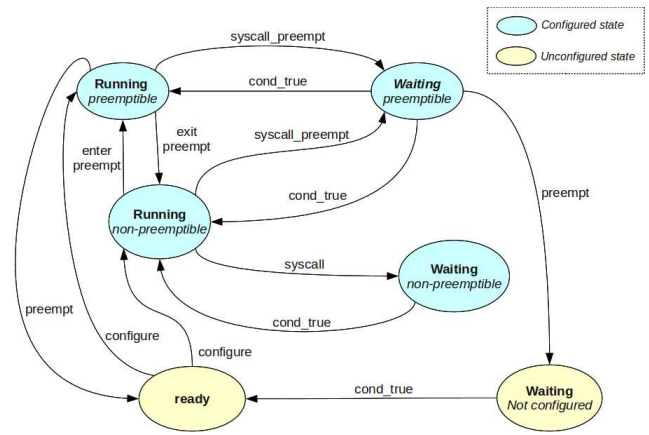


Fig. 5. Hardware Actor States

## V. COMMUNICATION FRAMEWORK

From the design point of view, this platform is divided into several interacting layers. The communication management part is named Middleware and is delimited by the red points in *Fig. 6*. It represents the solution for dataflow computation in the FOSFOR platform. The role of this layer will be to transparently execute the high-level actors by virtualizing the access to the needed operating system services. This transparency is reached by a standard API which homogenizes the communication process.

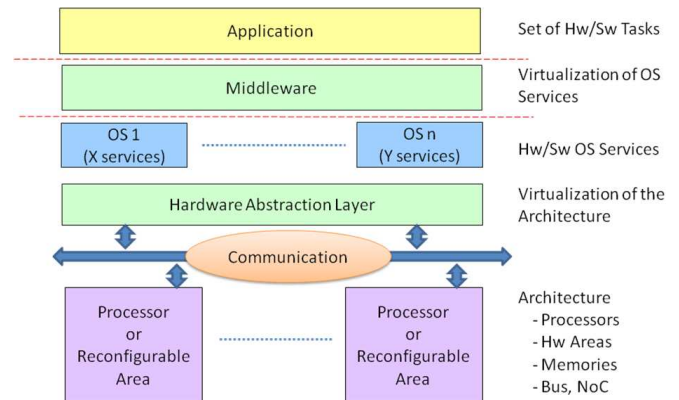


Fig. 6. FOSFOR layers

At low-level, the transfer of data between relocatable hardware threads is thus ensured by the management of Virtual Channels. Thus the middleware offers, both in hardware and software, additional services on top of the existing RTOS services. Threads can then use these services by subscribing to one or more channels. A subscription to one channel allows threads to be independent from the location of the others, thereby bringing closer the specification and its implementation.

### A. Middleware

The middleware must provide a set of system features available to the application to access to the execution resources, the

communication and the memory in a transparent way. System features virtualize the OS services and the application can then access those services regardless their physical location within the platform. This layer can request any service from the OS despite their location and the mechanics of invocation.

Its role is to enable communication between system resources ensuring synchronization of data exchange. The concept of control includes synchronization aspects, sending and receiving events, access to mutual exclusion semaphores, etc. We identify two types of communications: data transfer and control transfer.

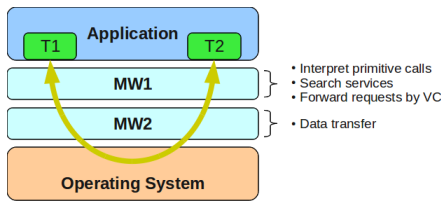


Fig. 7. Middleware layers

The middleware form the communication medium for transferring data between two thread, as shown in Fig. 7. This layer is described from two perspectives: the data transfer part (*MW2*), and the control part (*MW1*).

We took as a starting point the existing middlewares, and in particular the principle of the ORB<sup>1</sup> which, in our case, will be represented by the notion of Virtual Channel (*VC*). A VC is a channel designation indicating a particular virtual circuit on a network that will use the local and the distant services of the couple (OS, Middleware).

Two approaches are possible for the communications of the homogeneous type:

- The first one corresponds to a synchronous communication when the sender and the receiver are present during the communication. In this case, the exchange or transmission of data is done by directly sending data packets from transmitter to receiver. In this case we define the notions of *VirtualChannelSend* / *VirtualChannelReceive* and the synchronization of the exchange and the reservation of resources necessary for communication.
- The second type of communication occurs when one thread is not present (preempted or relocated) in the system at the time of communication.

From the latter case, there are then two possible scenarios:

- The current thread waits for the other one. However when a timeout (proposed as a service parameter) is reached the control is handed back to the system through the middleware, so that it continues its normal execution.
- The middleware stores temporarily data in global memory and shares it in order to free resources from the first thread once the copy is complete. This

<sup>1</sup>Object Request Broker

last case corresponds to an asynchronous communication.

In addition to the historical services of the OS, the virtual channel services are currently viewed as additional services that are mapped on top of the existing OS services beyond those offered by the MPC I layer.

Currently the implementation of these services of Virtual Channel is ongoing. It includes the *OpenVirtualChannel* primitive, which establishes a point to point communication between actors and determines the type of communication (synchronous or asynchronous). The sending and receiving services of the middleware : *VirtualChannelSend* and *VirtualChannelReceive*, which allows sending and receiving messages on the VC, from memory when copying the contents of transfer, or directly from the thread's buffer.

The service *EndCommunication* is only used by hardware threads to indicate the end of data transfer to the middleware and make threads preemptible again (FSM of Fig. 5). Indeed, as depicted in Fig. 4, unlike software threads, the hardware threads manage their own local memory and must then inform the Flexible HwOS of the end of transactions.

The service *CloseVirtualChannel* allows the closure of the channel, thus the channel becomes free again for a new transaction.

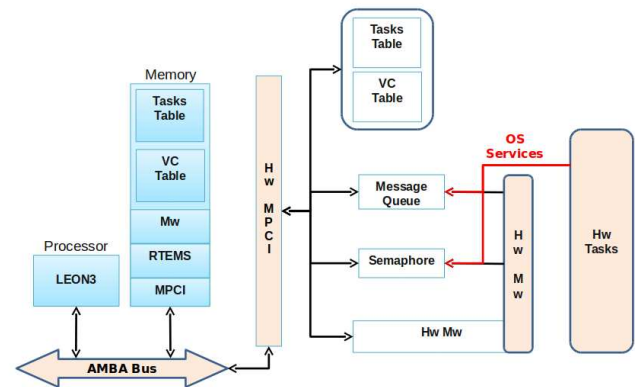


Fig. 8. Middleware communication

Each call to a service available on this platform requires to update the Virtual Channel tables, distributed both in hardware and software, onto the FOSFOR architecture (Fig. 8). This control flow is managed by the middleware on top of the message queue service of the underlying RTOS. Fig. 9 shows the attributes of the services and the return value of each of them.

### B. Network interface

The Network Interface (*NI*) is the static interface of the Hw Thread (Fig. 4). It is connected to a dedicated Network-on-Chip (*NoC*)[16] implemented, in order :

Services	Parameters	Return Value
OpenVirtualChannel	(id_VC, @Src, @Dest, Nb_Dest, R/W, State)	Error code
SendVirtualChannel	(id_VC, Buffer_Send, Size, Timeout)	Error code
ReceiveVirtualChannel	(id_VC, Buffer_Receive, size)	Error code
EndCommunication	None	None
CloseVirtualChannel	(id_VC)	Error code

Fig. 9. Virtual Channels services

- i) to offer a fast medium of communication between the hardware threads
- ii) to ensure them a way of communication with the software threads

This interface provides two low-level services, namely a procedure to send data from the thread's private memory to the NoC, and another to receive data from the NoC and write it into the same memory. Especially, this network has been designed to support bi-directional transactions, and so allows to implement efficient software pipelining mechanisms inside the hardware threads.

NI architecture is shown in Fig. 10. Two FIFOs allow the User FSM to stack Send and Receive requests. These requests are then respectively processed by a Packetizer and a Depacketizer. A DMA connected to the internal memory of the actor, is driven on one hand by the Packetizer to read data in memory and send it through the NoC. On the other hand, the Depacketizer receives data from the NoC and writes it into the internal memory.

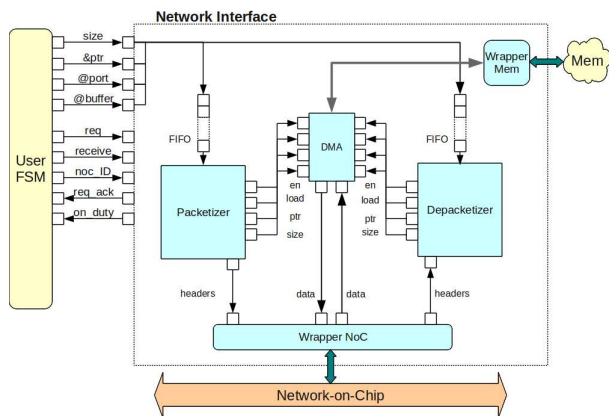


Fig. 10. Network Interface architecture

Elements connected to the NoC communicate through it by sending data packets over the network. A protocol has been specified and implemented to provide two main features to a thread.

The first one consists in writing data to an element connected to the NoC. This element can be another thread, a

memory or another kind of peripheral. The second is reading data from another element. In the case of a passive element, such as a memory, a special packet has been defined to allow read operation in two phases : one request from the thread and an answer from the memory.

The protocol adopted for the network has been thought to provide these two features in a transparent way for the developer. Moreover, additional data constitute the header of a packet, and make possible the routing of a packet between the hardware domain (*ie. the NoC*), and the software domain (*ie. the AMBA bus*), via a bridge able to understand this protocol (*See Fig. 3*).

## VI. EXPERIMENTS

### A. Image processing application

The application deployed on the FOSFOR platform is a target tracking application represented in Fig. 11. This application is responsible for detecting, tracking and recognizing targets existing in an infrared input video stream. The spatial resolution is 640x480 pixels by frame, and acquisition frequency at the output of the camera is 25 Hz.

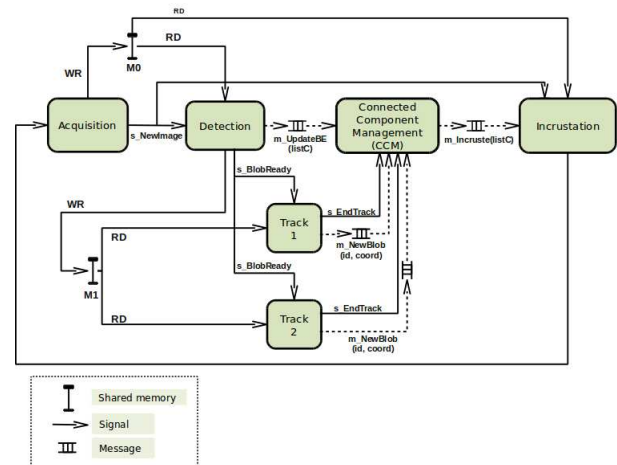


Fig. 11. Target Tracking Application

Application is divided into four static actors, and a dynamic one. The first actor of the static part corresponds to the acquisition of the data from the camera (*Acquisition*). It is followed by another one for the detection of the targets (*Detection*). The third actor gets the results from the tracking actors and ensures the coherency within a list of the current tracked targets (*CCM*). A last actor asks for this list and displays the encompassing square of each target into the input image (*Incrustation*).

The dynamic part of the application corresponds to the tracking actor, responsible for maintaining the coordinates of one of the target detected in the video (*Tracking*) by computing the Continuously Adaptive Mean shift (camshift) algorithm [17]. As a result, they provide to the CCM actor the bounding box coordinates of the target they are tracking.

## B. Application partitioning

Application has been ported first on a monoprocessor platform based on the Leon/SparcV8, running at 80 MHz on a V5SX50 FPGA. Table I shows the timing results for the detection actor which is the most time consuming.

Function	without FPU @ 80 MHz (in ms)	with FPU @ 80 MHz (in ms)
ghv_filter	18412.062	623.187
Roberts	13559	354.16
Local Maximums	3234.562	367.25
Tresholding	91	91
Closing	14297.562	14298.375
Labelling	515.937	513.5
LimiteLabels	98.312	96.687
LocateRegions	143.812	138.125
Total	58736.43	16720.43

TABLE I  
SOFTWARE IMPLEMENTATION TIMING

Some parts of the application are well suited to be implemented in hardware, such as *ghv\_filter* and *closing* operator. However, in the delay of the project, we chose to not focus on the timing performance, and privilege the demonstration of the dynamic management provided by the platform.

Finally, the detection parts are deported on a standard PC and connected to the FPGA through an Ethernet link. Thus, the hardware part of the application consists in the different tracking actors which will be created dynamically depending of the number of detected targets.

## C. Hardware implementation

For the demonstrator, we choose the ML506 board [18]. The software OS is RTEMS, which allows the communication between the two LEON3 processors [19].

The Camshift algorithm is divided into four steps. In the first step, it is waiting for a new image to process. Then it receives the frame of the image. In the third step, it processes the centroid of the target. Finally, when receiving the last data of the frame, it indicates if the target has been detected. The developed Camshift IP (Fig. 12) is a dataflow IP encapsulated into the hardware actor container of Fig. 4. A start signal indicates the beginning of a new frame. After starting, it consumes pixels corresponding to the frame and so, tracks a target into the frame calculating the centroid of the target. Once it receives the end signal, it delivers a result. This result is the coordinates of the centroid and indicates the processing convergence.

The control of the IP is decoupled into two channels. The first one is used to start and stop the processing. The second one to get data to process from the private memory. After processing, results are sent in an internal buffer which will be transferred on the NoC afterwards.

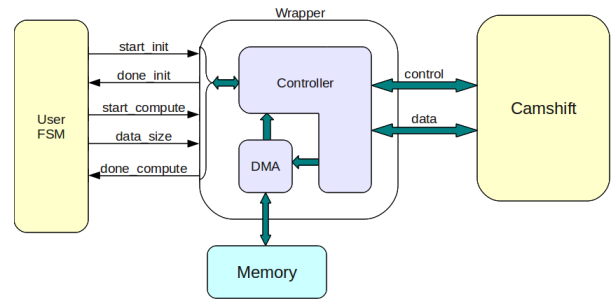


Fig. 12. Wrapper Camshift

## D. Application deployment

In order to map the dataflow model onto this target architecture, hardware actors have to support software pipelining. One solution to achieve this goal is to make the synchronization explicit between the NI Packetizer component, the NI Depacketizer component and the accelerator, into the request. A request will be composed of the information needed by the component to process the request, plus additional information about its synchronization with the other components (Fig. 13). Such a view fits into the dataflow model in which input tokens are generated and exchanged between the synchronization modules, regarding the requests instructions.

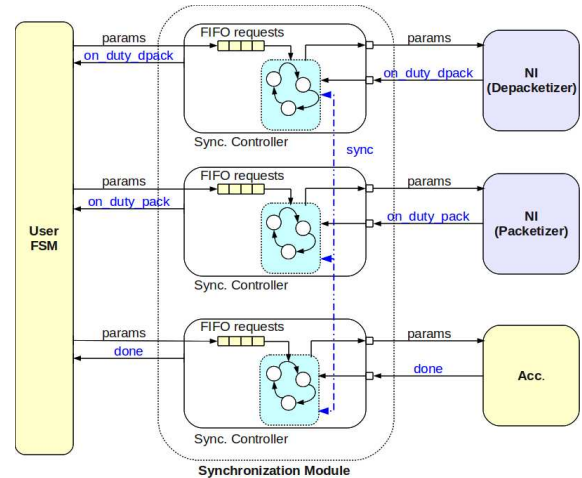


Fig. 13. Synchronization modules

Currently, the hardware thread model has been developed and integrated to the NoC and to the HwOS. Table II shows the FPGA resources used by the components needed to encapsulate the dataflow IP. Synchronization module is under development and will be integrated soon.

Adding the hardware reconfigurable container to the IP is not free, it represents a 13% overhead of the LUTs, and a 150% overhead of the registers for the small camshift accelerator. However the major part (*static*) of this container will be reused by the other reconfigurable actors and remains a poor percentage of the resources available in the current FPGA. Yet

Component	Registers	LUTs	BRAMs	DSPs
OS Interface	38	41	1	0
System FSM	4	6	0	0
Network If.	509	602	0	0
Packetizer	186	88	0	0
Depacketizer	67	103	0	0
DMA	132	184	0	0
Memory	0	0	1	0
Wrapper	129	75	0	0
Camshift IP	454	5387	0	5

TABLE II  
HARDWARE THREAD RESOURCES OVERHEAD

it can be a limitation for the number of reconfigurable slots into the FPGA.

Timing measurements for the Network Interface are shown in *Table III*. Measurements are done on the time to push and pop a requests in the FIFOs, and the time taken by the NI to decode the request and process it. Write and Read measures give the latency to write and read one 32-bit-width word in an internal memory (*ie. a BRAM*), connected on the top of the NoC.

Operation	Time (in cycles)
Push wr. req. by User FSM	6
Pop wr. req. by Packetizer	7
Process wr. req. by Packetizer and DMA	8
Push rd. req. by User FSM	8
Pop rd. req. by Packetizer	9
Process rd. req. by Packetizer and DMA	10
Write (transfer on NoC + process BRAM)	16
Read (transfer on NoC + process BRAM)	21

TABLE III  
NETWORK INTERFACE COMMUNICATION MEASUREMENTS

## VII. CONCLUSION AND FUTURE WORK

In this article, we presented a high-level model of computation dedicated to the heterogeneous Systems-on-Chip. We intend to propose a full flow in which the low-level hardware implementation matches with the concept brought by a SDF model of computation. In this way, a middleware layer has been added to an existing operating system in order to abstract the heterogeneity and the location of the different actors of the application.

A model of hardware actor has been evaluated, which takes into account, on one hand the advantage provided by the partial dynamic reconfiguration, and on the other hand the need of abstraction regarding the model of computation. With the increase of the FPGA circuits size, intermediate results reinforce our choice of providing generic interfaces, both for hardware and software actors. Such an abstraction facilitates the partitioning and also the design exploration of more and more complex heterogeneous Systems-on-Chip.

In the future work, we will continue to implement the middleware and the synchronization mechanisms of the hardware actors.

## VIII. ACKNOWLEDGEMENT

FOSFOR is a research project funded by the French National Research Agency (*ANR*). The authors would like to thank all those who have helped in the realization of this article.

## REFERENCES

- [1] T. A. Bishop and R. K. Karne, "A survey of middleware," in *Computers and Their Applications*, 2003, pp. 254–258.
- [2] F. Muller, J. Le Rhun, F. Lemonnier, B. Miramond, and L. Devaux, "A flexible operating system for dynamic applications," *XCell Journal*, no. 73, pp. 30–34, Nov. 2010.
- [3] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-oriented design of embedded hardware and software systems," *Journal of Circuits, Systems, and Computers*, vol. 12, pp. 231–260, 2003.
- [4] L. Verdoscia and R. Vaccaro, "Actor hardware design for static dataflow model," in *Workshop on Massive Parallelism: Hardware, Software, and Applications*, 1994, pp. 421–430.
- [5] H. K.-H. So and R. Brodersen, "Improving usability of fpga-based reconfigurable computers through operating system support," in *Field Programmable Logic and Applications*, 2006. *FPL '06. International Conference on*, Aug. 2006, pp. 1–6.
- [6] V. Rana, M. Santambrogio, D. Sciuto, B. Kettelhoit, M. Koester, M. Porrmann, and U. Ruckert, "Partial dynamic reconfiguration in a multi-fpga clustered architecture based on linux," in *Parallel and Distributed Processing Symposium*, 2007. *IPDPS 2007. IEEE International*, Mar. 2007, pp. 1–8.
- [7] E. Anderson, W. Peck, J. Stevens, J. Agron, F. Bajot, S. Warn, and D. Andrews, "Supporting high level language semantics within hardware resident threads," in *Field Programmable Logic and Applications*, 2007. *FPL 2007. International Conference on*, 27-29 2007, pp. 98–103.
- [8] E. Lubbers and M. Platzner, "A portable abstraction layer for hardware threads," in *Field Programmable Logic and Applications*, 2008. *FPL 2008. International Conference on*, 8-10 2008, pp. 17–22.
- [9] J. Agron and D. Andrews, "Hardware microkernels for heterogeneous manycore systems," in *Parallel Processing Workshops*, 2009. *ICPPW '09. International Conference on*, Sep. 2009, pp. 19–26.
- [10] S. Nordstrom, L. Lindh, L. Johansson, and T. Skoglund, "Application specific real-time microkernel in hardware," in *Real Time Conference*, 2005. *14th IEEE-NPSS*, Jun. 2005, p. 4 pp.
- [11] X. Guerin and F. Petrot, "A system framework for the design of embedded software targeting heterogeneous multi-core socs," in *Application-specific Systems, Architectures and Processors*, 2009. *ASAP 2009. 20th IEEE International Conference on*, Jul. 2009, pp. 153–160.
- [12] T. Nojiri, Y. Kondo, N. Irie, M. Ito, H. Sasaki, and H. Maejima, "Domain partitioning technology for embedded multicore processors," *Micro, IEEE*, vol. 29, no. 6, pp. 7–17, Nov. 2009.
- [13] B. Senouci, A. Bouchhima, F. Rousseau, F. Petrot, and A. Jerraya, "Fast prototyping of posix based applications on a multiprocessor soc architecture: "hardware-dependent software oriented approach"," in *Rapid System Prototyping*, 2006. *Seventeenth IEEE International Workshop on*, Jun. 2006, pp. 69–75.
- [14] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, no. 9, September 1987, pp. 1235–1245.
- [15] (1988) Rtems. [Online]. Available: <http://www.rtems.org/>
- [16] L. Devaux, D. Chillet, S. Pillement, and D. Demigny, "Flexible communication support for dynamically reconfigurable fpgas," in *Programmable Logic*, 2009. *SPL. 5th Southern Conference on*, 1-3 2009, pp. 65–70.
- [17] Y. Cheng, "Mean shift, mode seeking, and clustering," in *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 17, 1995, pp. 790–799.
- [18] (2011) Xilinx ml506 board. [Online]. Available: <http://www.xilinx.com/products/devkits/HW-V5-ML506-UNI-G.htm>
- [19] (2011) Gaisler research library. [Online]. Available: <http://www.gaisler.com/>