Middleware Based Executive for Embedded Reconfigurable Platforms

Amel Khiar, Nicolas Knecht, Laurent Gantel, Soufyane Lkad, Benoit Miramond

Middleware Based Executive for Embedded Reconfigurable Platforms

A. Khiar* and N.Knecht* and L.Gantel*† and S.Lkad* and B. Miramond*

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

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

***Abstract*—This paper presents a method to virtualize the communications into a distributed heterogeneous embedded Multiprocessor Systems-on-Chip *(MPSoC)* platform containing reconfigurable hardware computing units. We propose a new concept of middleware, implemented in software and in hardware to provide the designer a single programming interface. The middleware offers some mechanisms like the accesses to distant operating system *(OS)* services and the interprocess communication. It abstracts both implementation and mapping. The embedded application then executes regardless of where or how processes are implemented. We are currently validating the concept on a real-time image processing application.**

***Index Terms*—FPGA, Partial and dynamic reconfiguration, virtualization, RTOS, Middleware.**

I. Introduction

A MPSoC is an integrated circuit customized for an application domain. It contains most hardware components of general-purpose computing system such as processors, memories, buses, inputs-outputs (I/Os) and specialized hardware devices. MPSoCs are composed of heterogeneous processing elements which are more and more complex to integrate and program. In fact, they have different specifications and implementation languages, simulation/execution environments, interaction semantics and communication protocols.

This work takes place in the case of the french ANR FOSFOR *(Flexible Operating System FOr Reconfigurable platforms)* [1] project, the objective of FOSFOR project is to lay the groundwork for a Real Time Operating System (RTOS) to a new kind of more flexible and scalable OS for software and hardware tasks. In the first break with conventional approaches, this OS will be fully distributed and appear homogeneous in terms of the application on the whole platform. Although application tasks are deployed either on hardware or software processing unit, they can equally access to every services on the platform. This solution offers to the developer an embedded platform which is more scalable and flexible. The OS will be deployed in a modular architecture: an operating system kernel and a middleware (MW) instance for each execution unit. With the proposed virtualization mechanisms application tasks run and communicate without a priori knowledge on their assignment.

In the literature, there are different middleware definitions. The most popular is the following: "The middleware is the software which allows an application to interact or communicate with other applications, networks, hardware, and/or operating systems. This piece of 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 being visible to the user" [2]. Middleware provides a higher-level abstraction layer for programmers than Application Programming Interfaces (APIs) such as sockets that are provided by the operating system. It reduces significantly the burden on application programmers by relieving them of this kind of tedious and error-prone programming. Middleware is designed to mask some of the of heterogeneity that programmers of distributed systems must deal with [1], such as networks heterogeneity and hardware processing units.

The rest of the paper is organized as follow: section II describes the architecture of the FOSFOR platform. The related works are introduced in section III. In section IV we present the middleware dedicated to this platform. Finally, the last section introduces the results on an image processing application. We conclude our work in section V.

II. The Fosfor Architecture

The goal of the FOSFOR project is to define an architecture supporting a new kind of OS more flexible and more scalable. Unlike classical approaches, this OS can be totally distributed and the associated middleware makes the whole platform homogeneous from the application point of view. Moreover it allows application tasks to be deployed either in software *(on processors)* or in hardware *(on reconfigurable units)* [3].

The architecture of the platform is depicted in Figure 1, and is composed of:

- A set of general purpose processors *(GPP)* : a processor can support the execution of OS services and is in charge, on decision of the OS scheduler, of the execution of software threads. All the GPPs are not necessarily homogeneous in terms of instruction set architecture and number of offered services.

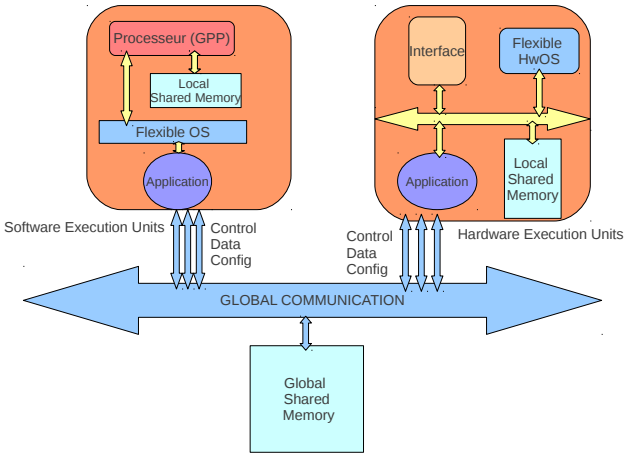

Fig. 1. The FOSFOR Architecture OS composed of 2 heterogeneous domains executing///// a shared memory paradigm

- A set of dynamically reconfigurable areas *(called Reconfigurable Region or RR)*. A RR is an area which is in charge of the simultaneous or sequential execution of a set of hardware tasks. As for a GPP, a RR can support the execution of OS services thanks to a hardware OS *(HwOS)*. These regions can correspond to fine-grained *(FPGA)* or coarse-grained *(reconfigurable processor)* architectures.
- A global communication medium between the two domains, software and hardware, based on a dedicated Network-on-Chip [4]. As we have a uniform memory access, all the execution units share the physical memory uniformly. That means that any memory block can be used as either configuration memory or program and data memory.

III. Related works

Middlewares are used in architectures to address the problems encountered today by MPSoCs: heterogeneity, execution environment, interaction semantics and communication protocols. When programming heterogeneous platforms, differences between hardware and software have to be hidden, so a virtual layer has to be added. DNA Operating System [5] 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. Petrot et. al in [6] 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 software processing elements in the platform. When dealing with the heterogeneity of hardware-software co-design, and so that the application is defined as a set of hardware and software tasks, a solution to face the management of tasks in a homogeneous way, is proposed by [7]. BORPH is a Linux based operating system which provides software drivers allowing to execute hardware tasks like the software ones. Agron et. al go further and offers a complete implementation of a POSIX thread in hardware. However, teh generation of the thread was done with a dedicated compiler from C to RTL, what is very limited in terms of portability and maintenance [8]. To abstract the heterogeneity, the reconfigurable data-stream hardware software architecture *(Redsharc)* [9] is a programming model and Network-on-Chip solution designed for MultiCore System-on-Programmable-Chip *(MCSoPC)*. This programming model relies on both a high level API, and lower level mechanisms directly implemented in the network interfaces. Two NoCs are used: one for control and the other for data transfer. Every interfaces on the platform are not homogeneous in terms of invocation services, for both HW and SW domain: for IPs interfaces they were defined uniformly, but it is different in the software domain. Our contribution in this paper is to extend this approach to the hardware reconfigurable domain, and to provide a higher abstraction layer which masks the operating system communication resources. To do so we offer a virtual channel API permitting the tasks to communicate directly. Moreover, in the same way that what has been done at high level in the OverSoC project [10], where the objective was to develop a model of heterogeneous platform for managing dynamic and reconfigurable platforms, we propose a programming model dedicated to this kind of platform, which abstract the developer from both the heterogeneity and the location of the application tasks.

IV. Middleware

The middleware approach has emerged as a promising solution for heterogeneity and distribution problems in the complex distributed systems. It provides a standard programming interface and protocols to the application layer. We are interested in the FOSFOR project, by providing a global communication framework.

The middleware must provide a set of system features available to the application to access to the execution resources, communications and memory in a transparent way. The role of the middleware layer is the virtualization of the access to needed operating system services. This transparency is reached by a standard API which homogenizes the communication process *(Fig. 2)*. 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 mechanism of invocation.

Therefore, the middleware must provide the lesser functionality of virtualization communications *(control and data)*, that is an essential support to access services on the entire platform. Its role is to enable communication between system resources ensuring synchronization of data exchange. The concept of control includes synchronization aspects,

events sending and receiving or access to mutual exclusion semaphores.

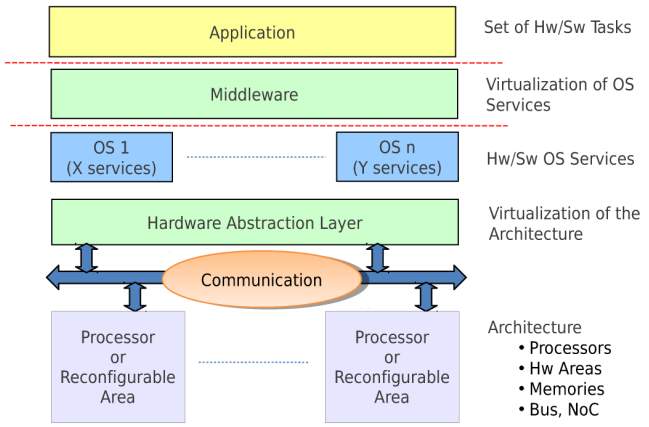

Fig. 2. The programming model for the FOSFOR platform is based on abstraction layers, from the HAL to the application description

Our innovation is reflected in the virtual channels, which offer both in hardware and software additional services on the top of the existing operating system services. Tasks can use these services by subscribing to one or more channels.

A. Types of communication within the platform

Communications can be respectively homogeneous or heterogeneous. The first type is related to communications between two application tasks implemented simultaneously in the hardware domain *(Hw/Hw)* or simultaneously in software *(Sw/Sw)*. The second type concerns communication between two application tasks implemented separately in the hardware domain and in the software one. For our project we took as a starting point the existing middlewares, and the principle of the Object Request Broker (ORB) which, in our case, will be represented by a 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 OS. Two approaches are possible for the communications:

A-The first one is when the two tasks are present during the communication, in which case the exchange or transmission of data is done by directly sending data packets from transmitter to receiver.

B-The second type of communication occurs between two application tasks, one of which is not present in the system at the time of communication. There are then two possible scenarios:

B1-The first one is that the current task waits for the other one (either the transmitter or the receiver). However when a timeout *(defined as a parameter)* is reached, the control is handed back to the system through the middleware, so that it continues its normal execution.

B2-The second allows the MW to temporarily store data in global memory and share it in order to free resources from the first task once the copy is complete.

B. Implementation

The platform includes many components of different nature *(hardware/software)*. Operating systems are also heterogeneous because the hardware part executes on a hardware OS. This HW OS is composed of the same services existing in the software part, which executes on top of the Real Time Executive for Multiprocessor Systems *(RTEMS)* software OS. This OS was chosen for its MPCI layer, *(MultiProcessor Communication Interface)*, which offers low level services to synchronize distant services in the software domain. This layer has been extended to the hardware domain in order to allow a low-level communication protocol between the two operating systems [1]. Each instance of the middleware is on the top of operating systems. The middlewares communicate by control messages thanks to the Message Queue service of their respective OSs, and are thus synchronized. This is done thanks to the distribution of the task tables and the virtual channels laid out all over the platform *(in the shared memory for the software side and in the memory on the hardware OS for the hardware domain)*. In *Fig. 3* we illustrate the separation between middleware services and OS services in the hardware domain. For the demonstrator, we choose the Xilinx ML605 board. The RTEMS OS allows the communication between two Leon3 [11] processors. There are two different types of calls: ones dedicated to the hardware OS and others dedicated to the hardware middleware.

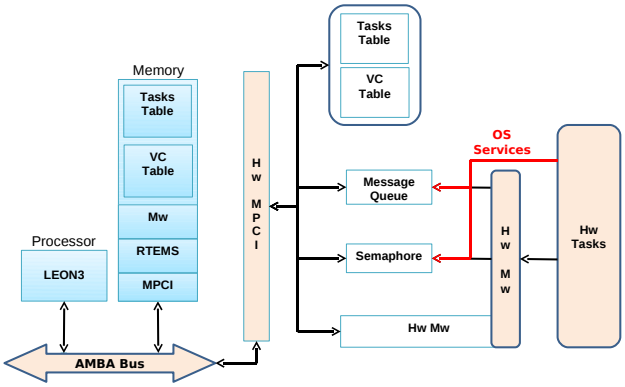

Fig. 3. The FOSFOR OS Mw Services are defined as a set of independent components allowing communication and synchronization between HW and SW tasks

As shown in *Fig. 3*, communications can allowing communication and synchronization between HW and SW tasks be done between heterogeneous tasks:

A-When the two tasks are present during the communication, there is two possibilities:

A1-If the receiving task is a software one, the data are always transferred using a buffer in the global memory. This buffer is allocated by the middleware service using the memory allocation service provided by the operating system.

A2-Else if the receiving task is a hardware one, we can take

advantages of the fact that a hardware task manages its own private memory. So, data are transferred from the sender to the receiver through the network-on-chip, without any copy.
B-When one of the tasks is not present in the system at the time of communication. There are then two possible scenarios:
B1-When the timeout is reached, the middleware returns the appropriate status code to the present task.
B2-Otherwise, if no timeout has been requested, the middleware allocates a buffer in the global memory for the sender. Once the receiver is ready, the buffer address is directly transmitted to this task which will get the data back in its private memory.

The middleware consists in a table of virtual channels, a static task, a FIFO and a finite state machine *(FSM)*. The FIFO manages the backup service request that comes from the Hw tasks. The Hw middleware is seen as a traditional service of the OS, which can access to the other services, as depicted in c.

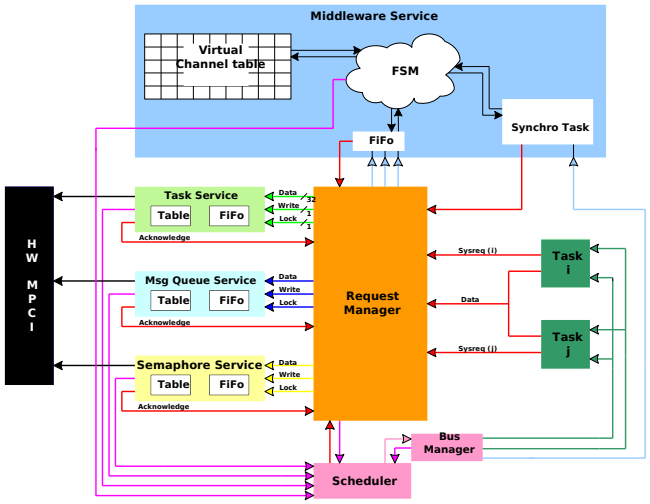

Fig. 4. The Hardware Middleware service interacts with HWOS services in order to update the VC tables distributed all over the platform

The request manager recognizes the identifier of the middleware service and then routes the calls to it. Then, the internal FSM must decode the request. To initiate a communication, the task uses the same API as its software counterpart. This results in the User FSM task by a system call to the same interface as for conventional calls. When the request is read by the Request Manager, it is forwarded to the Hw Mw which provides the location information for the other task. This information can be found in the VC table, composed of several fields that may made of other fields themselves. *Fig. 5* shows the VC tables organization.

The table is composed of the following fields: **Name:** this is the name given to the channel which is opened on the first call of the primitive **OPEN** at the allocation of the

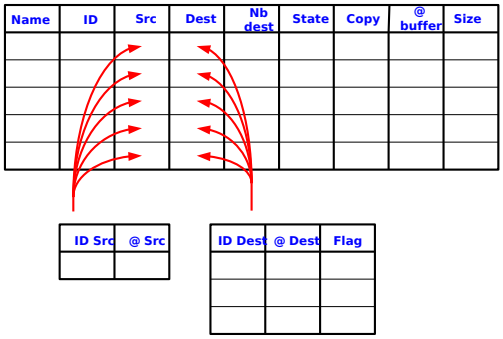

Fig. 5. The Virtual Channel Table stores the informations related to the tasks location

channel. **The identifier (ID):** this is the identifier that is automatically assigned when the channel is opened for the first time. **The source table (Src):** this is the space reserved for the sending task, it's necessary to store its name and address. **The recipient table (Dest):** this is the placeholder for receiving tasks, several tasks can register as receiver on the same channel. Flags will be used to separate allowance (Open Service) and request to receive (Receive Service). **The number of recipients (Nb Dest):** it will be incremented each time a new task allocation will be received. This allows to quickly determine the number of receiving tasks in the case of broadcast transfers.

C. The Virtual Channel Services

In addition to the historical services of the OS, the virtual channel services are currently viewed as additional services that are mapped on existing OS services beyond those offered by the MPCI layer, which does not cover services such as the naming service or the migration of tasks.

It includes the opening primitive **OpenVirtualChannel**, which establishes a point to point communication between tasks and determines the type of communication (with or without copy). The primitives of sending and receiving from the virtual channel: **VirtualchannelSend** and **VirtualchannelReceive**, which allows sending and receiving messages on the VC, from memory when copying the contents of transfer, or directly on the AMBA bus otherwise. The primitive **EndCommunication**, used to indicate the end of data transfer to make tasks preemptable again and the possibility of a future use of the channel. The primitive **CloseVirtualChannel** allows the closure of a channel. A call to these primitives available on this platform requires an update of the virtual channel tables distributed both in software and hardware.

V. Application

The proposed application represents a target tracking application. The application detects tracks and recognizes moving targets in a scene. This application has been selected for its dynamicity characteristics that will validate the ability of the OS to manage the proposed partial dynamic reconfiguration under real time constraints. The acquisition part reads the

video stream frame after frame in a buffer named M1 *(Fig. 6)*. The detection part carries out the research and the labelling of the areas in the new frames, writes resulting blobs in the M1 memory, then writes characteristics of the areas in shared memory blob list *(Track list)*. Finally, it activates the connected components management *(CCM)* task and the tracking tasks. Following these processing: filtering, detection of contours, closing of contours, labelling, the image is summarized with a list of areas with three characteristics: number of label, coordinates in the image of the center of the area and the height, width of the area. The CCM task is in charge of the maintenance of consistency between the detected areas for each frame. The tracking tasks are activated in broadcast by the detection task. They implement the Camshift algorithm [12]. In result they write the new coordinates in the shared memory in a linked list. If a track is lost, the corresponding target is removed from the list and the task destructs itself. The CCM task parses the list of the tracks in shared memory, then posts in the M1 memory a rectangle with the new coordinates provided by the tracking task. Finally it starts the detection task to process a new frame.

In order to be able to make an allocation of the functions on the various executions units, we proceed to the profiling of the application tasks [3]. According to the execution time of each of them, tasks with higher execution time will be put on a hardware component and those taking relatively less lower execution time will be put on to the processors. Initially, in order to emphasize the flexibility of the platform and not wasting time in the development process, only the tracking component has been ported in hardware. Indeed, as a tracking task is created for each existing targets in the scene, the management of these tasks emphasizes the use of the dynamic and partial reconfiguration in the platform. In addition, the hardware implementation of this task allows to get a speedup of X in term of execution time. Table I below shows the profiling of the resources of the platform built to support this application.

	Registers	***LUTs***	***BRAMs***	***Memory***
Leon3 based MPSoC	4463 *(16,95%)*	7951 *(11,26%)*	9 *(52,94%)*	-
Hardware OS	11286 *(42,86%)*	26292 *(37,23%)*	0	-
Hw Middleware	2412 *(9,16%)*	9118 *(12,91%)*	0	-
4 Camshift Hw tasks	8172 *(31,03%)*	27256 *(38,60%)*	8 *(47,06%)*	-
NoC 8 ports	920 *(3,38%)*	2902 *(3,95%)*	0	-
Sw Middleware	-	-	-	123 KB
RTEMS	-	-	-	$>$ 400 KB

TABLE I
MIDDLEWARE RESOURCES USAGE FOR 4 ALLOCATED CHANNELS

To validate the concept of heterogeneous system and reconfigurable platform for soft embedded real-time systems, we choose to map the tasks over the heterogeneous processing

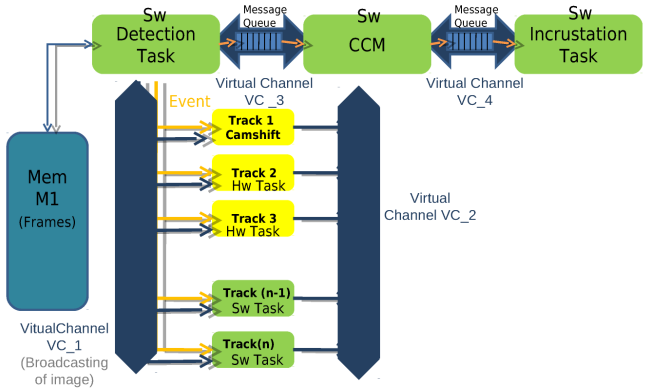

Fig. 6. Mapping of application Tasks

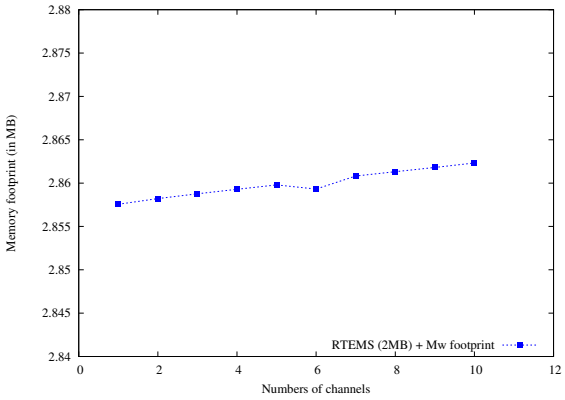

Fig. 7. Memory footprint according to the number of allocated VC in the software table

units as described in *Fig. 6*. So that communications are transparent and flexible regardless of the code or implementation spots on the heterogeneous execution units, and to provide remote access services on the platform, we virtualized communications on this platform by making a protocol communication, that will be translated into a dynamic number of virtual channels.

For the sake of efficiency, we evaluated the impact of the number of VC offered to the designer on the global resources utilization. On the software side, we found the corresponding memory footprint of the Middleware as shown in *(Fig. 7)*.

By default, a value of 2,855 MB corresponds to a simple application using RTEMS without the middleware. This size can be reduced to 400 KB for a stripped elf file *(symbol table and debug information are removed)* as shown in table I. The curve shows that the addition of the middleware use few memory resources on the software side.

Considering the number of needed virtual channels in our application described in figure *(Fig. 6)* the overhead over the RTOS is only about a few percent.

At the hardware level, the impact on the resources occupied becomes critical for 10 channels. Here, the number of registers

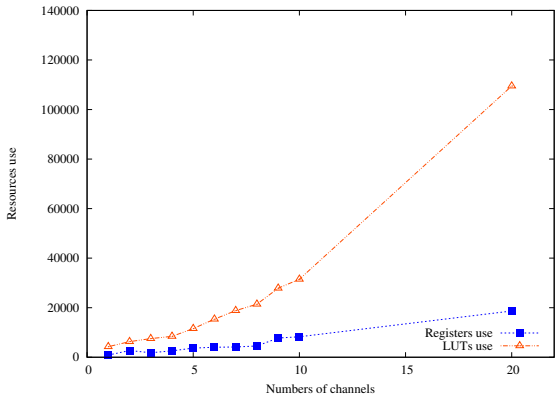

Fig. 8. Hardware middleware resources

will get closer to 60 thousands, which is almost the size of the target chip. Even if we just need 4 VC in our current application, we will be able to deal with this case in near future since the FPGA size has been constantly increasing for several years. The amount of resources plotted in figure 8 corresponds to the total size of the hardware executive, precisely the sum of the HwOS size and the Hw middleware size.

The *Figure 6* depicts the position of the VC in the application. After loading the frames from the M1 memory, tracking can be implemented on hardware or software components. Depending on the number of detected targets, there will be as many task as tracked targets, therefore they are created dynamically. The virtual channel between the tasks *(Detection, CCM and Incrustation)* are implemented using DMA triggering. The communication established between the Detection task and the tracking tasks, is represented by a virtual channel which takes as argument the size of data transferred, the recipient address and the sender address, that must first register themselves on the channel. Once the registration and the synchronization are established, the data transfer can then be done. The virtual channel between the tasks of tracking and CCM is opened in writing mode. At a given time, only one task can represent the sender, while all other tasks will be recorded.

A profiling of the overhead impacts in terms of execution time has been realized on this platform. Middleware primitives are evaluated and results are shown in Table II.

Primitives	***Perf. sw***	***Perf. hw***
OpenVC (VC not yet created)	73 μs	0.343 μs
OpenVC	7 μs	0.345 μs
CloseVC	5 μs	0.358 μs
CloseVC (VC deleted after close)	92 μs	0.358 μs
SendVC	24 μs	0.384 μs
ReceiveVC	77 μs	0.648 μs

TABLE II
EXECUTION TIME OF THE SW (SPARC V8 AT 80MHZ) AND HW IMPLEMENTATIONS OF THE MIDDLEWARE

Overheads are acceptable in software implementation, and the hardware implementation allows to not penalize the hardware tasks.

VI. CONCLUSION

We focused in this paper, on the communication aspect of embedded middlewares. This abstraction layer can be used to enforce the ease of programming of image processing application, and so target new hardware/software architectures. We have seen that this approach is a natural continuation of established parallel, heterogeneous, distributed computing systems. For that, we have described the Virtual Channel approach to virtualize communications into a distributed, heterogeneous embedded architecture, and we extend this approach to the dynamically reconfigurable architectures, in order to provide a uniform programming model for this kind of system.

VII. 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] 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.

[2] T. A. Bishop and R. K. Karne, "A survey of middleware," in Computers and Their Applications, 2003, pp. 254–258.

[3] L. Gantel, A. Khiar, B. Miramond, A. Benkhelifa, F. Lemonnier, and L. Kessal, "Data-flow Programming for Reconfigurable Computing," in 6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011, pp. 1–8.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] W. Kritikos, A. Schmidt, R. Sass, E. Anderson, and F. M., "Redsharc: A Programming Model and On-Chip Network for Multi-Core Systems on a Programmable Chip," International Journal of Reconfigurable Computing, vol. 2012, 2012.

[10] F. Verdier, J.-C. Prvotet, M. E. A. Benkhelifa, D. Chillet, and S. Pillement, "Exploring rtos issues with a high-level model of a reconfigurable soc platform," in ReCoSoC 2005: European Workshop on Reconfigurable Communication-centric SoCs, June 2005. [Online]. Available: http://publi-etis.ensea.fr/2005/VPBCP05

[11] (2011) Gaisler research library. [Online]. Available: http://www.gaisler.com/

[12] G. R. Bradski, "Computer vision face tracking for use in a perceptual user interface," 1998.