Communication Service for hardware tasks executed on dynamic and partial reconfigurable resources

Surya Narayanan
Computer Engineering
Delft University of Technology
Delft, Netherlands
S.N.KhizakancheryNatarajan@student.tudelft.nl

Ludovic Devaux, Daniel Chillet, Sebastien Pillement
INRIA
Cairn Lab, France
Ludovic.Devaux@irisa.fr
Daniel.Chillet@irisa.fr
Sebastien.Pillement@irisa.fr

Ioannis Sourdis
Computer Science and Engineering dept.
Chalmers University of Technology
Sweden
sourdis@chalmers.se

***Abstract*—The recent developments in the partial and dynamic Reconfigurable Computing (RC) domain demand better ways to manage the simultaneous task execution. In this context, Operating System (OS) services like scheduling, placement, inter-task communication have been developed to make this type of platform more flexible. In order to provide efficient communication scheme between these hardware tasks, a high performance communication infrastructure must be developed and efficient communication services must be proposed. The contribution presented in this paper mainly focuses on the hardware communication service and the communication schemes supported by this new OS service. Performance and implementation cost of our hardware communication service are evaluated and comparisons with the state of the art are given. Compared to related work OS4RS, our proposal is $60\times$ faster to establish communication between hardware tasks.**

I. Introduction

Application evolution towards more complex algorithms and the ability to extract parallel execution of different parts of the algorithm have led to the use of the multiprocessor SoC, heterogeneous SoC and then towards reconfigurable SoC. These evolutions demand an embedded OS for managing the run-time activities in such a complex system.

In parallel with the software evolution, embedded hardware execution platforms have also evolved towards more flexible and high performance systems. Such embedded system platforms more often include dynamic and partial reconfiguration resources to support the flexibility needed by the applications. This flexibility must be managed by the OS, particularly to schedule and place tasks in the appropriate Partially Reconfigurable Regions (PRRs). Indeed, within a heterogeneous platform, some tasks can be defined for processor execution (software task), or for hardware execution on an Intellectual Property (IP) blocks. Due to the unpredictable execution of tasks, the OS must be able to decide the resource allocation for the tasks at run-time and must provide mechanisms to support Inter-Task communication. Software techniques use shared memory for Inter-Process Communication (IPC) but for communication between two hardware tasks, we propose a more efficient scheme without storing messages in shared memory. This proposition leads to complex cases of communication schemes but ensures that reconfigurable resources can manage the communication between its hardware tasks.

The contribution of this paper concerns the development of a complete communication service for hardware tasks placed within a reconfigurable part of a Reconfigurable System. To ensure high performance exchanges, the communication service is placed in the reconfigurable system. An implementation of the communication service is presented along with the implementation results.

The remaining of this paper is organized as follows. Context of this work and the related works in this domain are presented in Section II. Section III presents the overview of Flexible Operating System For Reconfigurable systems (FosFor) platform and the design of hardware tasks for flexible RC. Then, the duties of OS communication service is briefed in Section IV followed by the implementation results in Section V. Finally, Section VI concludes the paper and discusses future work.

II. Context and related works

A few survey studies in the past decade explored the properties that an OS should posses for the dynamic reconfigurable platforms such as [1]. The initial set of minimum requirements were scheduling and placing the application tasks and then loading and initiating their execution.

Later, as the complexity of the algorithms grew, the need of multitasking and multiprocessing emerged as the solution. To adapt the same in SoC, Dally et al. in [2] proposed the usage of Network on Chips (NoC) instead of directly using the bus/wire to communicate between the tasks. NoCs proved to be an efficient method for high bandwidth, low latency data communication. Such Inter-Task Communication is realized using NoCs.

OS for RC has evolved over the past decade with the increasing needs to manage the dynamic task reconfiguration. Most of the previous works discuss the use of services offered by the software OS to manage the tasks configuration and to make RC platforms more flexible [3], [4]. In this case, all the services are provided by the OS running on general purpose processor. This is mostly achieved by exchanging some status words between the OS kernel and the RC tasks. The decisions

are taken by the programs running in the user kernel space. In heterogeneous systems, this method is not efficient due to the communication overhead between the software and hardware message transfers. In the past, there are a few projects which have contributed towards the hardware OS.

OS4RS [3] proposed a solution to handle Inter-Task Communication using two different NoCs: one responsible for application data communication and another responsible for OS Operation And Management (OAM). OAM is the control NoC which is used by the OS to manage the IP block functions in the platform. The Network Interface (NI) of the data communication NoC has a Destination Look-Up Table (DLT) which forms the data flow chart of the configured application. These DLTs can be modified at run-time by the OS, by sending an OAM message on the control NoC. Scaling of the DLT in the NI can be an overhead in such kind of configurations and in addition, the communication with the software OS to take decision is relatively slow.

BORPH [4] is an unified hardware/software platform for RC developed in UC Berkley. Based on UNIX OS kernel, BORPH offers services to both hardware and software processes. Communications between hardware and software are supported by using the IPC techniques available in the UNIX kernel.

Most of the platforms have tried to address the scheduling and placing of dynamic tasks but have not actually addressed the management of the run-time communication between multiple tasks running in both hardware and software. Our solution for this problem is hardware based and handles Inter-Task communication efficiently.

III. FosFor platform

Multitasking and multiprocessing have become common in software and are implemented in hardware too. There has always been a need for an OS to support multitasking in hardware which makes the system more flexible with dynamic reconfiguration. FosFor platform is an unified execution platform where the tasks can be mapped either on the general purpose processor core or on the reconfigurable resources. Software OS manages all the operations between the software tasks, but due to the heterogeneity aspect of the platform, it is also necessary to manage the operations between Hardware-Software (HW-SW) tasks and Hardware-Hardware (HW-HW) tasks. To support the communication between hardware tasks within the FOSFOR platform, we have developed a hardware CS which supports dynamic communication schemes.

In the spatial domain, high speed is always guaranteed as the execution is parallel in the hardware. In order to maximize this parallelism, high bandwidth and low latency communication are required between the hardware tasks. NoC is an efficient solution for this problem. In the FOSFOR platform, we have developed a Fat-Tree based high performance NoC, called DRAFT [5]. DRAFT provides the required network scalable performance with low resources consumption.

Design of a hardware task

The hardware tasks are partially and dynamically reconfigurable modules. Every hardware task must be able to call

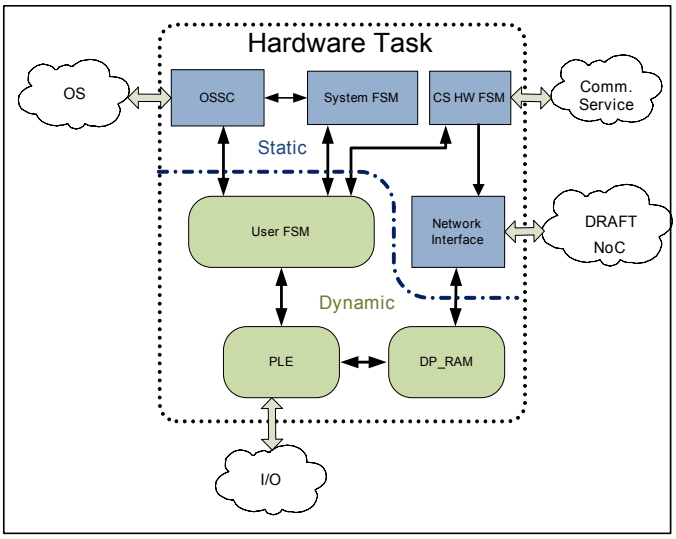

Fig. 1. A hardware task with static and dynamic part

the OS services developed for the platform. Therefore, the hardware task shown in inset of Fig. 1 is decomposed into two parts: the first one is a static part which consists of the control interface that connects the hardware task to the operating system for scheduling and placement of task, to the NoC for the data communication and to the communication service for establishment of communication. The second one is a dynamic part which usually consists of the User FSM (where system calls are initiated), Process Logic Element (PLE) (the computational part of the task) and an Internal Memory (IM based on Dual Port RAM, DP_RAM, where the local data are stored for communication and computation purpose). The dynamic part is the hardware application developer's space. The PLE can compute on the already available data in the IM or it can get the inputs from the I/O ports directly depending on the application. Similar to software programs, the hardware task designer has to call the system functions to invoke the OS Services. The system call signals are decoded in the static part and are sent to the OS services to be executed. The User FSM is connected to the OS through System FSM and OS Service Component (OSSC) in the static part. The static part also contains a Network Interface (NI) which connects the task to the DRAFT NoC for the data communication.

IV. Communication Service

OS supports a number of IPC techniques, such as signals, pipes, semaphores and shared memory which are the usual techniques. Similarly, this kind of communications can be implemented for hardware tasks to improve the flexibility of the RC system. In order to support such communications, we developed a Communication Service (CS) manager which enables and manages the communications between pairs of hardware tasks. The communication is supported by establishment of a virtual channel (VC) between two tasks. CS manager abstracts the information about the PRR location where the tasks are configured and establishes the data communication

between the tasks. Thus, the communications are ensured without any designer's knowledge about the location of the hardware tasks. A basic study on the possible communication schemes in the reconfigurable resource was presented in [6].

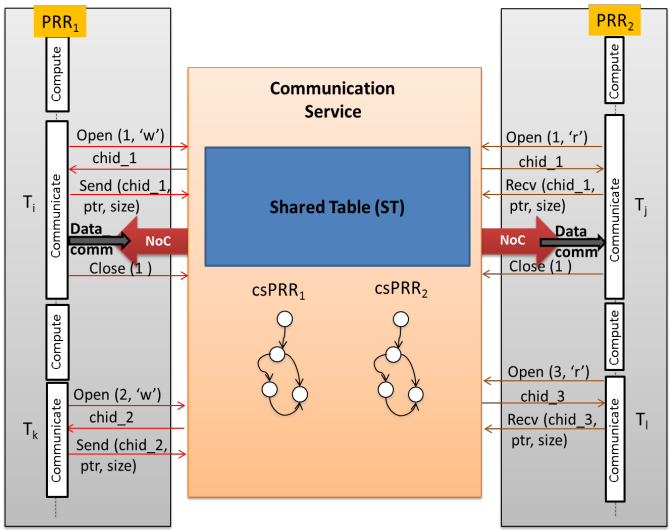

Fig. 2. CS module with Parallel FSM handling USER Syscalls and updating the service table

To manage the task communication, the CS hardware block must provide the followings functions:

- **Info / Signal Handling:**
  The hardware CS should have the information about the current state of all the tasks to take appropriate decisions on the system calls or signals from the other tasks. The status is maintained in the shared service table (ST). This table shown in Fig. 2 is the central database where all the system calls are registered. The CS shown in Fig. 2 has a dedicated FSM for each PRR ($csPRR_i$) monitoring the communication specific system calls from the executing tasks. Thus, it can support the simultaneous communications between different pair of tasks.
- **Service Initialization and Termination:** As mentioned earlier, the CS creates a VC between the pair of tasks which requires a connection. The tasks use the `OPEN` system call to open the channel. If CS finds an open request from sending task and receiving task with the same channel number, a VC is established between them. This is referred to as service initialization.
  `OPEN(Channel number, r/w mode)` - Open system call requests CS to open a particular channel in either read or write mode. Whenever a task requests to open a new channel, CS acknowledges it with a channel ID (`Ch_id`) for further communication using this reserved virtual channel.
  Once the data communication is over between a pair of tasks, the VC can be closed by using a `CLOSE` system call. This is referred as service termination.
  `CLOSE(Channel number)` - Close system call requests the CS to close the virtual channel. CS checks whether the data communication is over between the two tasks before closing the channel. Once the channel is closed, it is available for the other tasks to establish a new communication.
- **Inter-Task Communication:** Once the virtual channel is opened and acknowledged by the CS, a Send-Data or Receive-Data signal along with the Task ID, Memory Pointer, Port Address, Data Size are sent to the CS. These informations are updated in the service table. The `SEND` and `RECEIVE` system calls are explained below:
  `SEND(Ch_id, memory pointer, data size)` - Send system call requests CS for the physical address (NoC port address) of the receiving task along with the channel ID received during `OPEN` call. CS acknowledges the send system call with the physical address and initiates data communication through DRAFT NoC.
  `RECEIVE(Ch_id, memory pointer, data size)` - Receive system call requests CS for the physical address (NoC port address) of the sending task with the channel ID received during `OPEN` call. CS acknowledges the call in the same way as send request.

The OS, composed of a placer and scheduler, may dynamically configure the tasks in PRR. If data communication has to be established between two tasks, the NI of each task has to know their physical locations (PRR number). Now, it is the role of CS to inform the NI about the physical location of the tasks. There are three different scenarios which a CS has to handle. Figure 2 illustrates them.

- The first case is simple where both sender and receiver are available. The send-task T_i is instantiated on PRR_1 and opens the channel number 1. The receive-task T_j is instantiated on PRR_2 and opens the same channel in read mode. These two `open` system calls are captured by the respective CSFSMs (`csPRR1` for T_i and `csPRR2` for T_j) which update the service table (ST). Then T_i raises `send` system call to its CSFSM which updates the ST. At this step, CS knows that T_j is configured and has opened the same channel but is not ready to receive data. So, CS keeps T_i in wait state until T_j calls the `receive` system function for the communication to be established. The task T_i (respectively T_j) receives the port address of the task T_j (respectively T_i) and the NI of the two tasks can ensure the data communication. The data are sent from the internal memory of task T_i in PRR_1 and are received by the task T_j in PRR_2.
- The second case appears when the receiver has not yet opened a channel when the sender calls the `send` function. Here, the send-task T_k is instantiated on the PRR_1 and opens the channel number 2. As this channel is not open in read mode by a receiver, when the task T_k calls the `send` function, the CS must propose a solution to avoid stalling of the task T_k. In this case, we propose to store the data in a local memory (LM) task connected to NoC and when receive task wants to communicate through same channel, data is retrieved from LM.
- Finally, the third case occurs when the send-task has not yet opened the channel while the receive-task is ready to receive

data. The send task is not instantiated and receive task T_l is instantiated and has opened the channel number 3. In this case, T_l is suspended until some other task establishes a virtual channel and calls the `send` system function. In such case, the behavior is the same as the first simple case.

The method adopted for the Inter-Task communication is similar to software IPC techniques. By handling the above mentioned scenarios, the CS developed makes the RC platform more flexible as it can handle non-blocking communications.

V. Implementation Results

To evaluate the performances and the cost of our proposal, the whole platform has been implemented in Xilinx Virtex-5 XC5VSX50T FPGA. The experimental set up contains two tasks performing a simple communication. Messages were sent from internal memory of task T_i to the internal memory of task T_j. This section presents the results in terms of latency and area of CS and compares them with OS4RS results [3].

Several factors determine the latency of the whole data communication between tasks, such as, size of data, traffic in the NoC, location and availability of the sending and receiving tasks. When considering the communication establishment time for the OS communication service, we can ignore the time needed for data communication. So, the latency here refers to the time required by the CS to provide the system call and all the necessary parameters to the NI. An ideal communication refers to the situation when both tasks are available to communicate at the same time.
The CS developed takes 5 cycles to establish the communication in ideal condition. As the communication service is provided by the hardware, it has shown a substantial improvement in the performance compared to the software based CS provided by OS4RS. The communication latency in our hardware-based OS and in the OS4RS OS service are shown in Table. I. OS4RS uses control registers and status words to transfer information between software OS and hardware OS which operates in different frequencies. If compared to OS4RS which runs at 114MHz, our proposal is $60\times$ faster.

TABLE I
Latency comparison

Platform	Operational Frequency (MHz)	CS latency (ns)
FOSFOR	114	43.5
OS4RS	HW OS - 22 SW OS - 50	12800 320

As the CS is generic, area utilization of the CS was studied by increasing the number of tasks gradually from 2 to 8 as the 8 port DRAFT NoC was used for data communication. The number of LUTs needed for the CS and percentage area overhead of the CS compared to the rest of the platform are shown in the Fig. 3. The area taken by CS handling 8 tasks is 467 slices which is 5% of total number of slices in Virtex-5 SX50T. In comparison, OS4RS consumes 250 Virtex-II slices for the control NI and the router on each task.

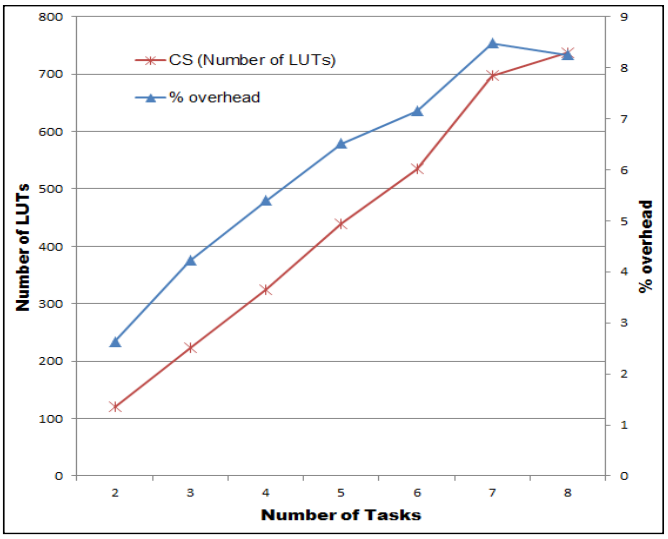

Fig. 3. Graph showing area occupied by CS (left Y-axis) and the percentage area overhead of CS (right Y-axis) by scaling the FOSFOR platform with respect to the number of tasks (X-axis)

VI. Conclusion and Future work

In this paper, we have presented the design of a hardware Communication Service for a heterogeneous platform containing a reconfigurable hardware. Our proposal extends the capabilities of a classical OS by supporting efficient communication between hardware tasks: i.e. tasks instantiated in the reconfigurable resource. CS supports the dynamic configuration of tasks in PRR and ensures the establishment of virtual channels without any specific restriction regarding the task locations. Our proposal of hardware CS is $60\times$ faster in establishing the communication between the hardware tasks compared to software based OS4RS and consumes small part of the classical Xilinx circuit.

References

[1] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers," vol. 23, (Los Alamitos, CA, USA), pp. 130–137, IEEE Computer Society Press, January 2001.

[2] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, 2001.

[3] T. Marescaux, J. y. Mignolet, A. Bartic, W. Moffat, D. Verkest, and S. Vernalde, "Networks on chip as hardware components of an os for reconfigurable systems," in *In Proceedings of 13th International Conference on Field Programmable Logic and Applications*, pp. 595–605, 2003.

[4] H. K.-H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, (New York, NY, USA), pp. 259–264, ACM, 2006.

[5] L. Devaux, S. B. Sassi, S. Pillement, D. Chillet, and D. Demigny, "Flexible interconnection network for dynamically and partially reconfigurable architectures," vol. 2010, (New York, NY, United States), pp. 6:4–6:4, Hindawi Publishing Corp., January 2010.

[6] L. Devaux, S. Pillement, D. Chillet, and D. Demigny, "OS services for Reconfigurable System-on-Chip Communications," in *Design of Circuits and Integrated Systems*, (Lanzarote Spain), 2010.