Hardware OS Communication Service and Dynamic Memory Management for RSoCs

Surya Narayanan #1, Daniel Chillet, Sebastien Pillement *2, Ioannis Sourdis $3

# *Computer Engineering, Delft University of Technology*
Delft, Netherlands
1 S.N.KhizakancheryNatarajan@student.tudelft.nl
* *University of rennes1 / CAIRN Labs, INRIA*
France
2 firstname.lastname@irisa.fr
$ *Computer Engineering,Chalmers university, Sweden*
3 sourdis@chalmers.se

***Abstract*—In a multiprocessor system, to gain the advantages of parallelism, efficient communication and memory management are highly required. The recent developments in the partial and dynamic Reconfigurable Computing (RC) domain require better ways to manage the simultaneous task execution. However, the requirements are bit different from the traditional software based systems. In this context, Operating System (OS) services like scheduling, placement, inter-task communication have been developed to make such platforms flexible and self-sufficient. In this paper we focus on the run-time communication service and dynamic memory management in the Reconfigurable System-on-Chips (RSoCs). We demonstrate the requirements and advantages of having a local memory task or a dynamically configurable memory task, how the communication service along with the DRAFT Network on Chip efficiently supports it. Compared to RECONOS [1], our proposal provides $3\times$ more throughput along with flexibility and scalability.**

I. Introduction

Operating System (OS) from its inception has grown widely and has found its importance in multiple software domains. It evolved as a software backbone for most computing devices and has simplified the application development. As the application complexity grew, research in OS took a new direction towards handling multiple tasks, parallel and distributed tasks which run on multiprocessors. Extending this, currently the OS is also required to manage multiple tasks in a RSoCs. Properties of OS which manages software system differs marginally from the ones managing hardware dynamic and partial reconfigurable (DPR) systems.

In a heterogeneous system, every processing node should be self-sufficient in handling the whole task in its execution environment to achieve high performance. For example, consider Fig.1, where a complex algorithm is divided into tasks and sub-tasks. Here, the tasks and sub-tasks have time constraints and data dependencies. One of the tasks, $Task2$, which is time critical and needs real time operation can be executed in the FPGA and rest tasks can be scheduled in the software. A self-sufficient system should be able to provide all the services for the execution of the task assigned to it. For example, the FPGA must manage the execution of $Task2$ by itself which demands the requirement of an OS.

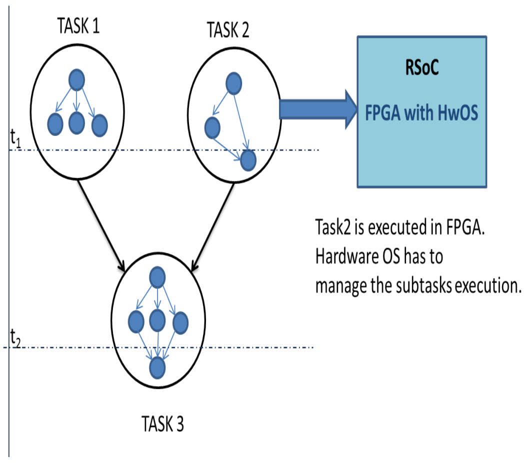
TASK 1
TASK 2
RSoC
FPGA with HwOS
t1
t2
Task2 is executed in FPGA. Hardware OS has to manage the subtasks execution.
TASK 3

Fig. 1. Data Dependence Graph showing that the critical real time task is executed in the hardware

Run-time operations in DPR systems require a simple OS which can provide required services for the management of tasks. In a DPR system the dynamic placement and scheduling of the tasks in FPGA requires high flexibility as the task can be configured in any available Partial Reconfiguration Region (PRR). **So, neither the location of the task nor the data traffic can be predicted at the compile time.** In order to support such dynamic systems, the underlying communication constraints like flexibility, scalability and throughput should be considered.

In FOSFOR platform (shown in Fig.2), we consider the OS to be scalable and provide similar interfaces to all the OS running in both hardware and software domain. We consider two types of OS which makes the OS heterogeneous. First, the software OS that runs on each processor and handles the execution of software threads. Second, the hardware OS (HwOS) that runs on the FPGA and handles the execution of hardware tasks. In order to achieve homogeneity of the services between heterogeneous OS kernels a common communication unit is developed. Thus, we ensure that the whole platform abides the similar protocol and appear homogeneous.

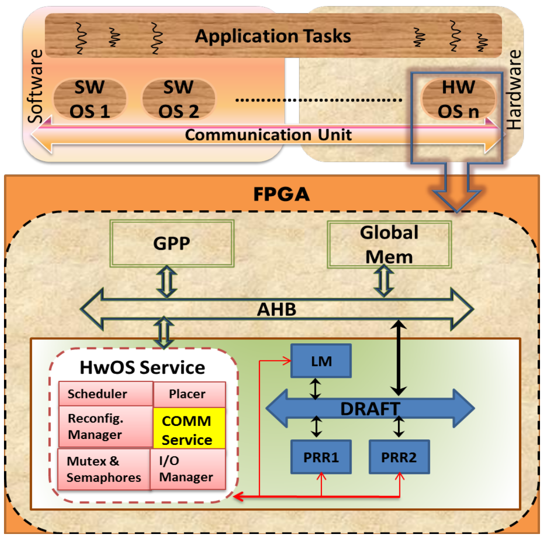

Fig. 2. FOSFOR Platform

The contribution of this paper focus on the communication schemes supported by the hardware communication service (CS) manager [2] which provides Inter-Task communication service between two hardware tasks dynamically configured in arbitrary Partial Reconfigurable Regions of a DPR system. We have explored all the possible communication scenarios i,e blocking and non-blocking communication which is explained in section III. Our proposal to support the blocking communication is based on the allocation of static local memory or dynamic local memory in the available PRR. On a whole using the DRAFT Network on Chip [3], which is developed to support DPR systems along with Communication Service (CS) improves the system flexibility and reduces the communication overhead.

Remaining of this paper is organized as follows. Context of this work and the related works in this domain are presented in Section II. Different communication schemes supported by CS are briefed in Section III followed by how CS manages memory task in Section IV. In Section V we discuss about evaluation of the work with a proof of concept application and its behavioral results. We also compare our results with state-of-art. Finally, Section VI concludes the paper.

II. Context and Related Works

The requirements of RSoC OS are slightly different from the OS used in general purpose computers. The need of OS to manage the reconfigurable system started in late nineties when partial reconfiguration was possible. Few survey studies [4] aimed to identify the properties that an OS should possess to manage partial and dynamic reconfigurable systems. The first OS for reconfigurable systems, OS4RC [5] was just a loader, that allocates FPGA area and that can dynamically partition, place and route applications at run-time.

As the RSoCs became multi-processing and multi-tasking, efficient schedulers and scheduling strategies were developed to manage the task execution considering the time and resource constraints [6], [7]. In [8], authors propose a hybrid run-time scheduling by pre-fetching the configuration data in parallel with the task execution. Along with scheduling, pre-emption is also one of the important features a multi-tasking system should possess. In [9], authors propose a hierarchical configuration mechanism to deal with the context switching and inter-task migration.

Most of the previous works discuss the use of services offered by the software OS to manage the tasks in DPR systems [10], [11]. In this case, all the services are provided by the OS running on a processor. This is mostly achieved by exchanging some status words between the OS kernel and the hardware tasks. The decisions are taken by the programs running in the user kernel space. In heterogeneous systems, this method is not efficient because of the communication overhead between the software and hardware message transfers.

Using a multiprocessing system can parallelize execution of many complex algorithms. But, communication and sharing of data between the multiprocessors remains a major bottleneck. In [12], author proposed the usage of Network on Chips (NoC) instead of directly using wires to communicate between the tasks. NoC proved to be an efficient method for high bandwidth, low latency data communication and scalability.

OS4RS [10] proposed a solution to handle Inter-Task Communication using two different NoCs: one responsible for application data communication and another responsible for the control. The data communication is performed by updating the Destination Look-Up table (DLT) in the Network Interface (NI). Scaling of the DLT in the NI can be an overhead in such kind of configurations and more over the communication with the software OS to take decisions for dynamic communication establishment is comparatively slow.

RECONOS [1], is based on eCos real-time OS which supports inter-task communication between hardware - software and hardware - hardware task and analyzes the incurred area and latency overhead. In this work, two ways of hardware-hardware communication are proposed. 1. Bus Master Access-by using PLB interface a hardware task can directly access memory and other bus connected peripherals in the system, 2. Hardware FIFO IP core to realize the Inter-Task communication between the two defined hardware tasks. The limitation of these techniques are lack of flexibility and scalability.

Current works for providing communication service to hardware tasks executing in the reconfigurable devices are under the control of a software-based OS or through bus based communication. These ideas were to extend the services provided by software OS to the HW tasks. In FOSFOR, we provide all OS services by designing and developing the hardware IP which can provide these services to the hardware tasks. The scope of this paper is to show the performance benefits of the hardware CS manager along with the DRAFT NoC [3] we have developed and how a scheduler designed for this platform can take advantage of the communication schemes and the memory management provided by CS. DRAFT is a NoC specifically designed to support the constraints induced by the dynamically reconfigurable nature of the hardware tasks.

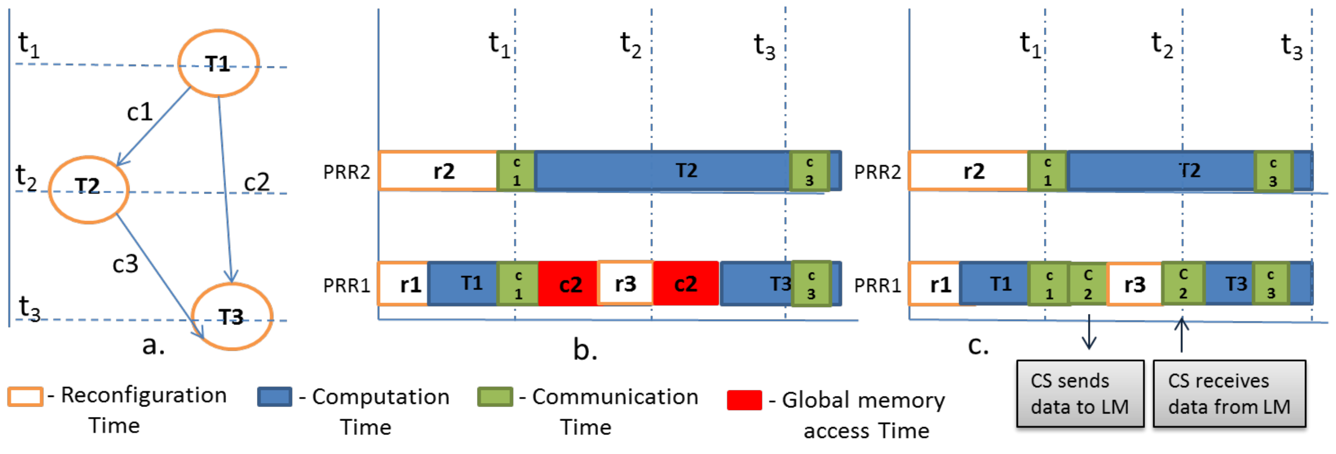

Fig. 3. a. Shows the data dependency graph, b. Unable to schedule the tasks within time limit due to the dependency, resource constraint and absence of local memory in the system, c. Scheduler along with CS overcomes the problem in the presence of a local memory.

III. Communication Service and Schemes

In order to manage the Inter-Task communication between the tasks configured in different PPRs, we need a communication manager or a middleware which monitors the communication channel requests from the tasks and establishes a communication between them. Execution in spatial domain are parallel, so, a dedicated hardware which monitors, controls the communication and at the same time interacts with other hardware OS service is developed [2]. This CS manager abstracts information from the system call raised by the hardware tasks and supports Inter-Task communication between multiple tasks simultaneously.

In most types of communication, the following 3 scenarios have to be handled usually. They are:

1) Both sending and receiving task are ready simultaneously
2) Sending task is ready but the receiving task in not ready
3) Receiving task is ready but the sending task is not ready

The first case corresponds to non-blocking communication while the next two are blocking communication.

- **Non-blocking Communication:** In non-blocking Communication, both the sending and receiving tasks are available simultaneously. In this case, CS can directly establish the data communication between the tasks and the execution continues.
- **Blocking Communication / Memory Management:** The scheduler may schedule the tasks in the most efficient way given the time and resource constraints. In a resource constrained system, a sending task may get stalled due to the non-availability of receiving task, we refer this to be a blocking communication. In such cases, the scheduled task is blocked. Presence of a local shared memory can solve this problem with less latency overhead. Using NoC, we took the advantage of connecting a memory task to one of the ports of the NoC. Thus, when such scenario arise, CS takes the decision to store the computed data in the local memory by creating a virtual channel with the memory task. Thus, CS makes the communication non-blocking. Later, the receiving task can be configured in the same or any other available PRR and can retrieve the data from the memory task.
Moreover, if the local memory is already used, we also propose a solution to reconfigure a dynamic memory task in the available PRR and store the data in this specific task. Such management needs interaction between CS and scheduler. Considering that run-time scheduling is possible, the scheduler can utilize the available PRR to configure dynamic memory task. Then, intimate the CS about the presence of dynamic memory and its location. CS can now manage both the local and dynamic memory tasks.

We demonstrate how CS handles all these cases with the help of the Data Dependency Graph (DDG) as shown in Fig. 3 and its corresponding pseudo code for each task.

Program for task T_1

```
1: Compute
2: $Ch\_id\_1 = OPEN(1, w)$
3: $SEND(Ch\_id\_1, memorypointer, datasize)$
4: $CLOSE(1)$
5: $Ch\_id\_2 = OPEN(2, w)$
6: $SEND(Ch\_id\_2, memorypointer, datasize)$
7: $CLOSE(2)$
```

Program for task T_2

```
1: $Ch\_id\_1 = OPEN(1, r)$
2: $RECEIVE(Ch\_id\_1, memorypointer, datasize)$
3: $CLOSE(1)$
4: Compute
5: $Ch\_id\_2 = OPEN(3, w)$
6: $SEND(Ch\_id\_2, memorypointer, datasize)$
7: $CLOSE(3)$
```

Program for task T_3

```
1: $Ch\_id\_1 = OPEN(2, r)$
2: $Ch\_id\_2 = OPEN(3, r)$
3: $RECEIVE(Ch\_id\_1, memorypointer, datasize)$
```

```
4: $CLOSE(2)$
5: Compute
6: $RECEIVE(Ch\_id\_2, memorypointer, datasize)$
7: $CLOSE(3)$
8: Compute
```

For the given DDG shown in the Fig. 3.a, T_1 is the initial task, the data computed in T_1 are utilized by T_2 and T_3, T_3 requires data from T_1 first and then later from T_2. A resource constrained scheduler with only 2 PRRs to configure hardware tasks and a global memory will schedule the DDG as shown in fig. 3.b which violates the timing constraints due to the absence of local memory task. In the absence of any form of memory the task is blocked and the scheduler cannot configure another task in the respective PRR.

The first communication is between T_1 and T_2 where T_2 is configured in parallel to the execution of T_1 and is ready to receive data from T_1 via channel.1 (c1) as shown in Fig. 3.a. Once the computation is over, T_1 opens the c1 in write mode to send the data. CS handles the system calls and establishes the data communication.

Next, T_1 has to send data to T_3 but due to the resource constraints T_3 cannot be configured. But, T_1 opens c2 to send the data. Here, CS handles the blocking call by storing the data in the local memory (LM) task Fig. 4. Once the data is secure, scheduler can configure T_3 in PRR1 and continue with its execution. Now, when T_3 opens c2 to receive the data, CS manages the communication by providing the stored data in LM for further computation. Thus, the whole DDG can be scheduled as shown in Fig. 3.c. How CS manages the memory task and the communication are explained in Section IV.

The final communication is between T_2 and T_3. As T_3 has already opened c3 it can receive the data when T_2 opens c3 to send the data.

IV. Memory Task Management

To support the above mentioned communication schemes, we need a memory task. Memory tasks are the storage tasks which can be configured into any available PRR connected to the NoC. These tasks are different from the traditional tasks as the operation like reading and writing into these tasks are controlled by the signals from the CS according to the requirements i,e they can be a sending or receiving task accordingly. These memory tasks can be of two types:

1. Statically configured Local Memory task (LM) - As the CS is generic, during the system design, the designer can choose to have one or more LM depending on the complexity of the algorithm and these LMs are connected to one of the ports of the NoC. As it is statically implemented the size of the memory can be varied according to the designers choice.

2. Dynamically configured memory task (DM) - The hardware CS has dedicated csFSMs for each PRR which monitors the system calls from the corresponding tasks. This csFSM can also handle if the PRR is configured with a memory task. Every PRR in the system should have a DM synthesized for itself in addition to the regular tasks. During a blocking communication, CS always looks for the availability of LM, and if LM is occupied by some other task, the decision to configure a memory is escalated to the scheduler. However, it depends on the decision of the scheduler either to configure a DM or to store the data in the global memory. The decision of the scheduler depends on the availability of PRR and the size of the data that needs to be stored. The most important fact to be considered is the cost and latency of implementing a DM, the size of the DM cannot exceed the size of the dynamic part of the PRR.

As the Local Memory tasks are connected to one of the ports of DRAFT NoC, bandwidth and latency of accessing the memory depends on the traffic in the NoC. The memory task shown in Fig. 4 consists of a LM_FSM which communicates with the CS. The memory block inside the LM is connected to the NI, so, reading or writing of the memory is possible only through the NI. This kind of memory task are either mapped on to BRAM or on a single port distributed RAM along with additional logic for LM_FSM and NI in the LUTs. A Dynamic Memory task works similar to LM task with a wrapper around it which makes it compatible to be configured in dynamic part of any PRR.

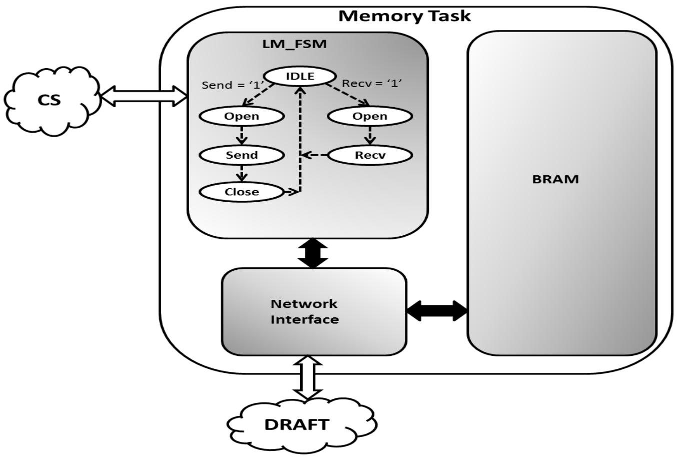

Fig. 4. A Local Memory task that temporarily stores data and supports the CS system calls.

CS embeds dedicated FSM ($csFSM_i$) monitoring the system calls from the tasks, similarly, the LM task is also monitored by a dedicated FSM ($lmFSM_1$) as shown in Fig. 5. Consider the DDG in Fig. 3.a and the blocking call that occurs between T_1 and T_3 as they both needs to be configured in same PRR due to the resource constraint. In this case, LM is used to avoid the blocking and the efficient scheduling scheme looks like in Fig. 3.c. In Fig. 5, when $csFSM_1$ finds that there are no receiving task which has requested to open the same channel, it wakes up the idle $lmFSM_1$ with the corresponding signals which in turn activates the LM as a sending or receiving task and updates the Shared Table (ST). LM and ($lmFSM_1$) remains busy and occupied till the receiving task reads the data and terminates the communication. Currently, the CS restricts the LM to be used by one blocking task i,e even if the memory is not full, a second blocking task cannot use the LM till the first communication is over. This choice prevents

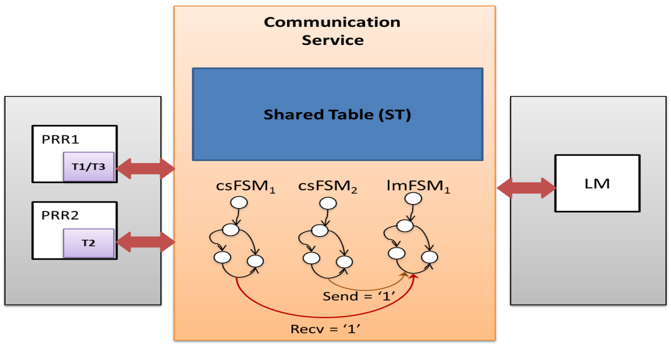

Fig. 5. Blocking communication where Local Memory task is used to store data till receiver is ready

us from the efficient usage of LM and avoid complex memory management.

There can be many scenarios where an existence of a local memory task can improve the performance of the system. For example, consider a resource constrained system which has multiple PRRs with different sizes. If there exists 2 tasks T_1 and T_3 which are of same size and can be configured only in one particular PRR, then, both the tasks has to be scheduled and configured in the same PRR. Here, due to the size incompatibility, tasks cannot be configured in other available PRRs. In such case, having local memory task is efficient compared to storing the data in the global memory.

In case, if the local memory is already occupied and if any PRR is available, scheduler can configure a dynamic memory task and inform the CS about the location of DM. In this case, the regular $csFSM$ also manages the communication between the blocked task and a dynamically configured memory task.

The above mentioned implementations are possible because of the flexibility given by the DRAFT NoC and the CS which can establish a communication between the 2 tasks configured at any PRR.

V. Evaluation

To evaluate the flexibility, performance and the cost of our proposal, we implemented the experimental set-up as shown in Fig.6 in Xilinx Virtex-5 XC5VSX50T FPGA. The set-up consists of a 8 port DRAFT NoC which is responsible for data communication between 8 PRRs that are configured with a static local memory task, AES128 encryption, AES128 decryption and a random traffic generator. We have considered that scheduler utilizes an empty PRR to configure DM and informs the CS about its configured location. This experiment demonstrates the functioning of the CS when the tasks are configured in any available PRR and its dynamic memory management capabilities. We have also compared the results with the traditional software mechanisms by performing similar experiment on a system containing 2.53GHz Intel processor and Ubuntu Linux as OS with 2 GB RAM.

A. Experiment and performance

In this experiment, AES encryption task is configured in PRR1, the traffic generator in PRR2 and AES decryption in

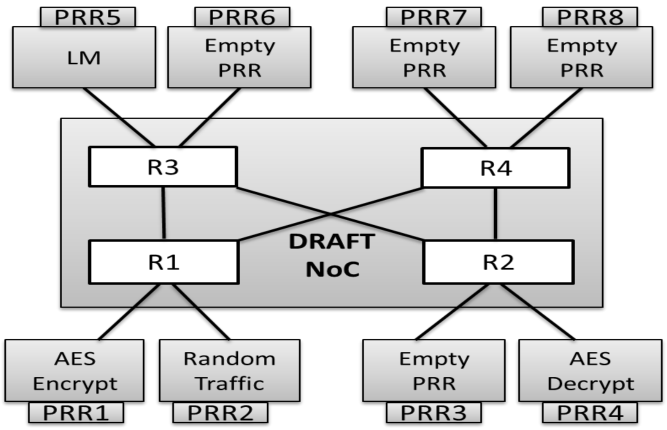

Fig. 6. Experimental set-up

PRR 4. All the 3 communication schemes explained in section III are realized in this experiment.

1. To illustrate the simple direct non-blocking communication, we configure only the AES encryption and decryption tasks and initiate its execution. Here, decryption task stays in the receiving state till it receives data from the encryption task. Once the encryption is over, CS takes care of the communication between the two tasks. The total execution time of this non-blocking task for different size of data is shown in the graph in Fig.7.

2. To illustrate how the CS manages blocking communication using the LM and DM tasks, we configured the sending tasks i,e random traffic generator and AES encryption first and the receiving task i,e AES decryption task later. When CS does not find any receiving task for the generated random traffic, CS directs the data to the memory task and marks the memory task busy. As explained earlier, the first blocking communication is always directed to LM if it is available. Later, when AES decryption is configured and requires the encrypted data for its computation, it requests a receive system call with the same channel number to the CS. CS configures the LM or DM as receiving task and thus the data is retrieved. The total execution time of this non-blocking task for different size of data is shown in the graph in Fig.7.

Moreover, the graph shows the total execution time taken by the software. The execution times are measured for 2 different software Inter Process Communication (IPC) techniques. One, in the user level, the IPC can be realized using files. Other, in the system level we can use unix $pipe()$ system call to perform the IPC. Experiments were conducted by forking the parent process into two where the child process executes the AES Encryption and the parent process executes the AES Decryption, the encrypted values are communicated to the parent process by either using files or $pipe()$. Plot AES SW (files) shows the total execution time of AES using files for IPC. Plot AES SW (pipe) shows the total execution time of AES using $pipe()$ system call. The difference between AES HW non-blocking and blocking is very small and the plots

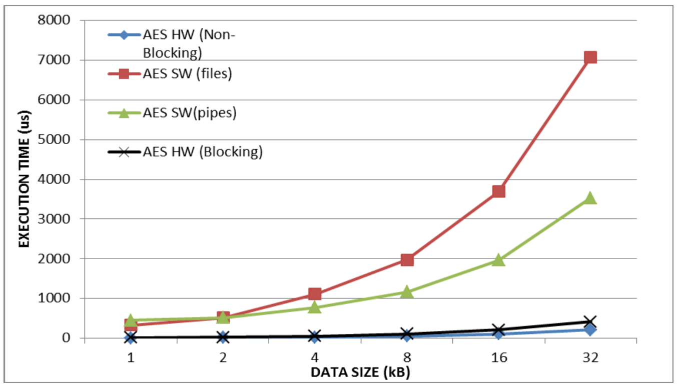

Fig. 7. Graph showing the total execution time of the AES algorithm using different methods

almost overlaps because the communication latency doubles during blocked communication which is comparatively less than the AES computation time.

B. Communication Latency

The hardware system operates at 100MHz clock frequency. There are few important factors that determine the latency of the data communication between the 2 hardware tasks, such as, size of the data, traffic in the NoC, location and availability of the sending and receiving tasks. As the communication service is provided by the hardware CS IP block it takes only 5 cycles to establish communication if both the sending and receiving tasks are available at the same time. It takes 11 cycles to establish communication with the LM or DM. Total communication latency of our system includes the communication establishment time and the data communication time. Plot Total communication latency of communicating different size of data between tasks configured in different PRRs are presented in Fig.8. Graph in Fig.8 also shows the total communication latency in different environments. Plot pipe syscall shows the total communication latency when unix system call $pipe()$ is used to communicate data between two forked process. As mentioned earlier, in RECONOS Inter Task communication between hardware tasks is established using hardware FIFOs, communication latency of this environment is shown in the plot RECONOS HW-HW.

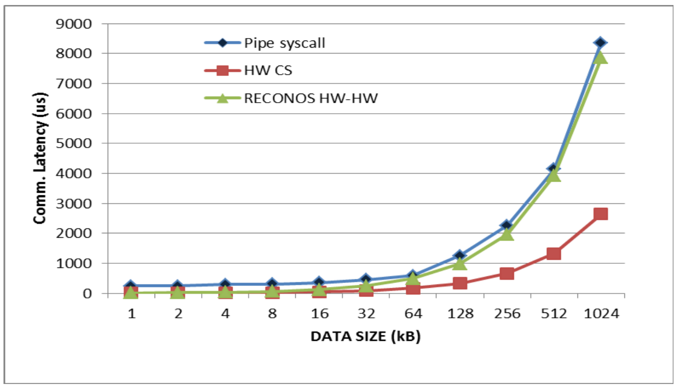

Fig. 8. Graph shows the communication latency for different data size

On comparing the results of CS with the RECONOS approach using hardware FIFO, our method of communication using hardware CS and NoC is 3× faster. Moreover, hardware FIFO communication is fixed between two hardware tasks. On the other hand, in FOSFOR the communication system is flexible and scalable and can establish communication between hardware tasks configured in any PRR without the knowledge of the physical location of the tasks.

VI. Conclusion

In this paper we discussed about the usage of CS to manage the different communication schemes by utilizing the local memory or a dynamically configured memory tasks in the system and the advantages of it. We consider that in the future the OS for RSoCs should be fully self-sufficient to manage the task requirements locally. We have compared the communication latency of our system with the state-of-art RECONOS and software pipes. Our approach of hardware communication service outperforms them by 3× times and proves to be a flexible and scalable design.

References

[1] E. Lbbers and M. Platzner, “Communication and synchronization in multithreaded reconfigurable computing systems,” in *Engineering of Reconfigurable Systems and Algorithms*, 2008, pp. 83–89.
[2] SuryaNarayanan, L. Devaux, S. Pillement, D. Chillet, and I. Sourdis, “Communication service for hardware tasks executed on dynamic and partial reconfigurable resources,” in *Proceedings of the 2011 VLSI-SoC conference*, Hong Kong, China, 2011.
[3] 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: Hindawi Publishing Corp., January 2010, pp. 6:4–6:4.
[4] G. Wigley and D. Kearney, “The first real operating system for reconfigurable computers,” vol. 23. Los Alamitos, CA, USA: IEEE Computer Society Press, January 2001, pp. 130–137.
[5] ——, “The development of an operating system for reconfigurable computing,” in *Field-Programmable Custom Computing Machines, 2001. FCCM ’01. The 9th Annual IEEE Symposium on*, 29 2001-april 2 2001, pp. 249 –250.
[6] C. Steiger, H. Walder, and M. Platzner, “Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks,” *IEEE Trans. Comput.*, vol. 53, pp. 1393–1407, November 2004.
[7] J. A. Clemente, J. Resano, C. Gonzalez, and D. Mozos, “A hardware implementation of a run-time scheduler for reconfigurable systems,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1 –14, 2010.
[8] J. Resano, J. A. Clemente, C. Gonzalez, D. Mozos, and F. Catthoor, “Efficiently scheduling runtime reconfigurations,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, pp. 58:1–58:12, October 2008.
[9] T. Marescaux, V. Nollet, J. Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, “Run-time support for heterogeneous multitasking on reconfigurable socs,” *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 107 – 130, 2004.
[10] 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*, 2003, pp. 595–605.
[11] 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*, ser. CODES+ISSS ’06. New York, NY, USA: ACM, 2006, pp. 259–264.
[12] W. Dally and B. Towles, “Route packets, not wires: on-chip interconnection networks,” in *Design Automation Conference, 2001. Proceedings*, 2001.