An ARM-based Microkernel on Reconfigurable Zynq-7000 Platform

Tian Xia, Jean-Christophe Prévotet, Fabienne Nouvel

An ARM-based Microkernel on Reconfigurable Zynq-7000 Platform

Tian Xia, Jean-Christophe Prévotet and Fabienne Nouvel
Université Europe de Bretagne, France
INSA, IETR, UMR 6164, F-35708 RENNES
{tian.xia; jean-christophe.prevotet; fabienne.nouvel}@insa-rennes.fr

***Abstract*—the combination of ARM processor and partially reconfigurable FPGA device is an emerging technology in the current embedded domain. In this paper we propose a custom microkernel on a hybrid ARM-FPGA platform, which is capable of managing reconfigurable hardware accelerators. We will introduce the hardware platform on which the microkernel has been developed and focus on the custom architecture supporting the management of partial reconfiguration and software tasks. An actual use case is studied and presented at the end of this paper to demonstrate the feasibility of our approach.**

***Index Terms*—FPGA, reconfigurable architectures, embedded system, microkernel.**

I. Introduction

In recent years, the commodity field programmable gate array (FPGA) has become a widely-applied technology for embedded applications. The FPGA fabric permits time-multiplexed sharing of the hardware resources so that more modules can be implemented in one chip. However, the traditional FPGA computing lacks flexibility since the whole fabric is required to be reconfigured even when modification is required for part of FPGA, thus causing enormous time overhead and power consumption. To deal with this drawback, the Dynamic Partial Reconfiguration (DPR) was proposed as a solution, which allows particular areas of an FPGA to be altered while the rest executes without interrupt. With DPR feature, an FPGA is capable of implementing more complex architectures by breaking it down into smaller mutually exclusive modules. In this case, hardware accelerators, which can be dynamically dispatched and managed, are becoming as flexible as software functions, being then considered as a parallel computing resource to processors rather than fixed accelerators. However, the application of DPR is currently limited due to the design complexity [1].

On the other hand, there have been increasing concerns about the reliability and security of embedded systems, especially for mobile devices. One response to this concern has consisted in elaborating an efficient management of such systems. In this context, dealing with microkernels constitutes a promising idea because it allows executing applications of different natures (commodity APIs, real-time tasks, etc.) in their own isolated container to ensure isolation and thus security. Consequently, it has been a popular research trend in the embedded systems domain [2]. In embedded circuits such as FPGAs, dealing with a microkernel may also be of interest since it is then possible to easily add a new DPR management service to the existing services already provided by the kernel. In this case, besides the kernel management, it is also necessary to implement a hardware architecture dedicated to the coordination of reconfigurable resources and to the cooperation with high-level applications.

In this paper, we propose an approach of DPR management by implementing a custom microkernel Mini-NOVA on a hybrid ARM-FPGA Xilinx Zynq-7000 platform [3]. Mini-NOVA is based on ARM architecture and integrated with the management service of reconfigurable hardware resources, which allows for dynamic SW/HW task management, secure execution environment and efficient communication..

The remainder of the paper is organized as follows: Section II presents backgrounds and works done in DPR domain. In Section III, an overview of the embedded platform is given, including the components of our architecture. Section IV demonstrates the design and implementation of the Mini-NOVA microkernel in details. In Section V, we present a case study to evaluate the functionality and performance of our architecture. Finally, the conclusion of our work will be given in Section VI.

II. BACKGROUNDS

As embedded systems become increasingly complex and as designers face more and more challenges, FPGA’s adaptivity has become a crucial asset. The principle of DPR is to permit specific areas of the FPGA to be reprogrammed with an alternative behavior while the rest of the fabric remains the same, so that the inherent flexibility of FPGA is improved.

Currently, there are several partially reconfigurable commercial FPGA series, which include the Atmel AT40K and the Xilinx Virtex FPGA family [4], and the newly-released Xilinx Zynq-7000. In Xilinx FPGA products, the size of reconfiguration data varies with the amount and types of reconfigurable hardware resources. For complex computation-massive modules, their data size could be quite large, meaning that the reconfiguration overhead could be quite considerable for these modules. Especially in a computing-intensive system, where several mutually exclusive components are sharing reconfigurable resources, the time lost on reconfiguration will severely degrade the overall performance [4]. Therefore, a dedicated efficient management is essential in DPR systems.

Numerous studies have been led to propose efficient hardware architecture management with OS support. One typical embedded OS which based on Linux kernel was OS4RS [5], which aims for a dynamic relocation of tasks between a host processor and reconfigurable hardware. Other researches include run-time reconfigurable architectures [6][7] and hardware-implemented OS services [8][9], which have focused on providing effective online scheduling and hardware task communication. However, the matter that restricts performances of classical FPGA devices is the fact that most of them are employing embedded processors such as MicroBlaze or PowerPC [8][9], whose computing ability is relatively limited [10].

Unlike previous devices, the Zynq-7000 platform integrates a powerful dual-core ARM Cortex-A9 processor with various on-board resources and peripherals [3]. With this fully capable processing system, the CPU processes software data, while the programmable fabric is considered as a unique auxiliary computing resource. In this case, the reconfiguration management is expected to be one of many tasks in the system and a specific kernel is required to rationally dispatch both hardware and software resources.

Meanwhile, though microkernel technologies have been widely studied in the embedded system domain, its application in reconfigurable computing are relatively less studied. One research in this domain is introduced in [11], where a hardware-based microkernel is used to provide OS services. Compare to OS support, the advantage of the microkernel technique is that it offers system security, flexibility, task isolation and real-time capability [12]. These features are quite suitable for an embedded system, because in most applications' scenarios such as vehicles and mobile phones, both safety and commodity APIs are executing. While most existing microkernels on ARM do not consider the reconfigurable hardware, one possible solution is to port an appropriate microkernel from existing kernels. In [13], a L4 kernel is ported to manage both hardware and software tasks, but without high-performance hardware modules. Thus, the management of dynamic partial reconfiguration is still absent. In a microkernel, the key feature to focus on, is the size of its trust computing base (TCB), which determines the security level, the dispatching speed and the porting complexity.

III. Architecture Overview

The on-site reprogrammable FPGA fabric integrated with powerful ARM processors brings up enormous possibilities for embedded technique, while the approaches of fully exploiting and designing efficient methods are far from sufficient. In this paper, we propose a platform framework based on the hybrid ARM-FPGA platform of the Zynq-7000. The target of our approach is to establish a microkernel-based embedded system with flexible hardware tasks. A simplified block diagram of the proposed architecture is shown in Fig. 1.

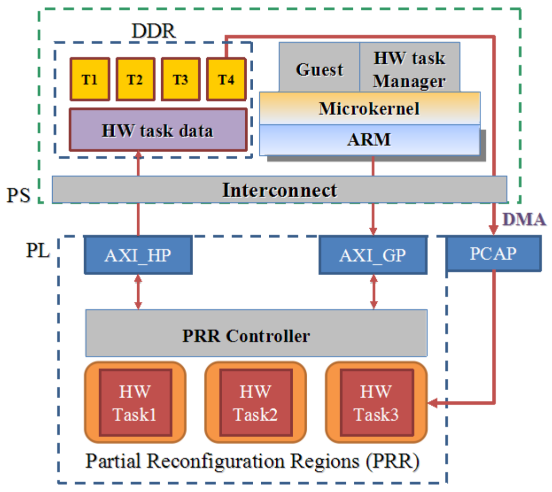

Fig. 1. Block diagram of architecture

The proposed architecture is divided into two domains: the Processing System (PS) and the Programmable Logic (PL). Processing System is the central processing unit and includes the software computing resources, such as the DDR on chip memory (OCM), the ARM Cortex-A9 processor and various peripherals. On the CPU, a simplified microkernel hosts guest applications/software and specific user services in the user space. Microkernel is responsible to schedule these components properly. The Programmable Logic mainly consists of the FPGA fabric, which houses different hardware accelerators and executes concurrently with the PS side. To control and reconfigure the FPGA modules dynamically, a specific user service routine Hardware Task Manager is proposed as a guest in the user space. In this way the FPGA resource can be dispatched as standards user application and thereby hardware and software tasks can be managed concurrently in our framework.

With DPR technique, it is possible to switch one or several HW tasks without interfering the rest of the accelerators. Such a feature can be applicable not only in increasing the system performance, but also in some scenarios where heterogeneous hardware structures are mandatory. For example, to offer the support for coexistence communication standards such as cellular communication standards and wireless LAN(WLAN), a reconfigurable platform could be quite suitable. We will present a practical use case in Section V.

A. Hardware task organization

In reconfigurable embedded systems, hardware tasks are FPGA modules with different functionalities and fabric structures, which are either custom-designed computing blocks or commercial IP cores. In our system, hardware tasks are pre-defined and synthesized by Xilinx XPS design flow, generating different bitstream-format files which hold the modules' fabric information. These bitstream files can be stored in memory device and implemented in certain areas of the FPGA.

As shown in Fig.2, within the FPGA fabric there exist multiple pre-defined areas to house hardware tasks separately,

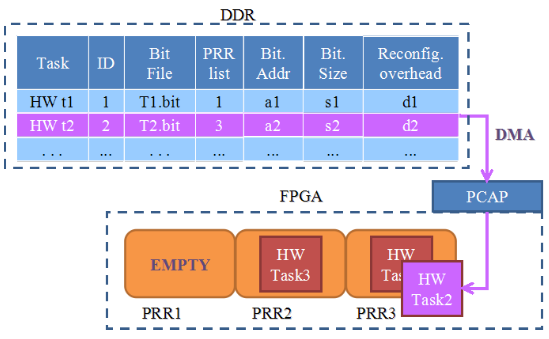

Fig. 2. Hardware task organization

which are called partial reconfiguration regions (PRR). The PRR corresponding to each HW task is pre-determined. These hardware task containers execute under the supervision of the PRR controller, a special function block to control the behavior of hardware tasks. All bitstream files are indexed in a look-up table (LUT) by the unique IDs of each HW task, which contains the descriptors of each hardware task. The descriptor entry indicates the bitstream file's ID, name, address, size, reconfiguration overhead and pre-defined PRR. Different HW tasks are dispatched by downloading its bitstream file into its assigned PRR. Normally more frequently-called HW tasks may have multiple PRRs so that they can be used by different guests concurrently.

We should also note that since the bitstream size is determined by the PRR's size, the reconfiguration overhead is also linearly correlated to the PRR's area and then can be precisely predicted.

B. PL/PS communication port

To connect PL with PS, we applied 2 types of interface based on the standard AXI interface. Some technical statistics related to this interface are listed in TABLE I. The general-purpose AXI port (AXI_GP) is designed for low-speed general purpose communication, while the high performance AXI port (AXI_HP) is defined for high performance with burst transfer. AXI_GP offers a unified mapping to the processor and can be accessed as a normal memory access. AXI_HP may transfer data blocks as large as 4KB for one burst, which is sufficient for general data-processing application.

On our platform, HW task manager takes control of two master AXI_GP interfaces as main method to configure and read back the states of HW tasks. It is also possible to have 4 AXI_HP interfaces that are used at HWs service and in charge of accessing both on chip memory (OCM) and DDR. Since AXI_HP is working in the slave mode, data are fetched and written back without acknowledging the processor, allowing the processor to run simultaneously with HW tasks.

TABLE I. PL/PS COMMUNICATION INTERFACE

Type	Num	Mode (PS)	Access	Speed
AXI_GP	4	2 Master, 2 Slave	Unified Addr space	600MB/s
AXI_HP	4	4 Slave	DMA	1200MB/s

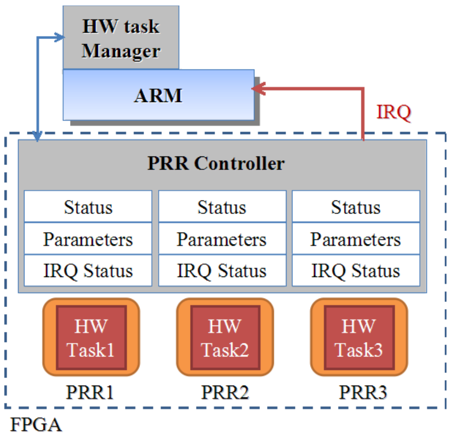

Fig. 3. PRR Controller Structure

C. Reconfiguration interface

Two methods for partial reconfiguration are supported on the Zynq platform. As shown in the datapath of Fig. 1, PS (Processing System) is enabled to initialize bitstream transfers from memory to PL (Programmable Logic) through the Device Configuration Interface (DevCfg), which will launch a DMA transfer via the Processor Configuration Access Port (PCAP). Another available reconfiguration datapath is the Internal Configuration Access Port (ICAP), which is capable of self-configuration from the PL side with an AXI4-Lite as transfer port. Such a mechanism severely limits the throughput of data reconfiguration to 19MB/s. Another drawback of ICAP is the additional FPGA resource consumption for its implementation, since it requires a hardware structure and will occupy at least one AXI interface. To achieve better performance and reduce resource consumption, PCAP is used in our platform.

D. PRR Controller

In the PL domain, a partial reconfiguration region (PRR) controller block is proposed to monitor the behavior of hardware tasks. It cooperates with the special user service Hardware Task Manager to coordinate the execution of software and hardware tasks. As demonstrated in Fig.3, the PRR controller allocates to each PRR a group of registers, which are mapped into the unified memory space via AXI_GP ports, so that CPU may access to them directly. By reading and setting values of these registers, CPU is able to monitor and change the HW task's behavior.

1) *Reconfiguration :* One major responsibility of the PRR controller is to monitor and guarantee the security of HW task reconfiguration. It should control the state of PRR and avoid unsafe reconfigurations which may cause undesired states such as invalid data output or incomplete data frame. Thus when a PRR reconfiguration is required, several concerns are involved:

- If the HW task to be reconfigured is part of a certain multi-block pipeline structure, the pipeline should be emptied before any HW task switch, so that invalid output data are avoided.
- To maintain the integrity of the data structure being processed, considering that for certain computation

processes it is mandatory to work with packages of data with determined length, PRR reconfigurations should be launched in interval of data frames to protect data and ensure a smooth HW task switch.

- A reset should be inserted to the reconfigured PRR to put it into a desired state. The new HW task is allowed to be activated only after the first complete reset.

The states of PRRs are presented in their register groups. If PRR is not ready to be reconfigured because of the above situations, then PRR controller would set the *PRR_Reco_Rdy* bit in the state register to zero so that the CPU won't try to reconfigure it. Once the PRR is ready to be reconfigured, the PRR controller sets this bit to high again.

2) *AXI interface :* The PRR controller also manages HW tasks accesses to the AXI_HP interface. Normally, HW tasks are given access to the AXI_HP interface on their own. However, in cases where AXI interfaces are insufficient for the HW tasks, the PRR controller manages the time-multiplexed sharing of this AXI interface. In this case, the PRR controller works as a crossbar of datapaths among HW tasks.

3) *Interrupts :* the PRR controller is able to generate general-purpose interrupts through Shared Peripheral Interrupts (SPI) connected to the generic interrupt controller (GIC). These IRQs are used to acknowledge important events to the CPU, or to synchronize the software tasks with the hardware task's states. They are handled by the microkernel and passed to the Hardware Task Manager for proper handling.

IV. Mini-NOVA Microkernel

As discussed in Section II, applying microkernel in ARM-FPGA architecture can greatly improve the management of SW/HW tasks since it offers higher security and better flexibility. Mircokernel runs on top of bare-metal CPU with the basic OS capabilities, serving as an abstract layer between physical machine and user applications. The principle of least privilege should be strictly followed o guarantee that a minimal trust computing base (TCB) is achieved for our microkernel. Based on these considerations, we propose a simplified microkernel Mini-NOVA; which has dedicated user service and scheduling strategy to support DPR management.

A. Mini-NOVA Overview

Mini-NOVA is revised from the NOVA micro-hypervisor [14], with simplified functionality and reduced complexity, which makes it more suitable for embedded systems and also more adaptable. Since NOVA is originally designed on x86 platform, several modifications are made to port it to ARM Cortex-A9 architecture, with additional supports for the Zynq-7000 platform. The overview Mini-NOVA structure is shown in Fig.4. The software space is divided into kernel space and user space, with different privilege levels. The kernel code runs in the higher level, while user applications and some user services run in the lower level (user space), which are referred to as the user guests. As the host, the microkernel code is restricted to the basic functionalities such as memory management and scheduler, to minimize the TCB size. Most board-specific support APIs and services are implemented in user

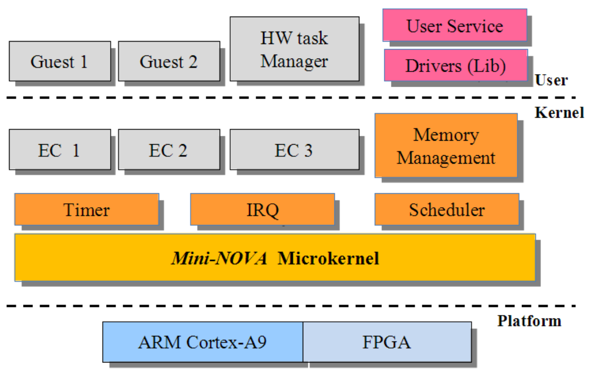

Fig. 4. Mini-NOVA architecture

space, including HW task manager, application bootloader, and supports for on-board peripheral resources.

To provide isolated execution environment, Mini-NOVA creates a kernel object Execution Context (EC) as the abstraction of user threads or applications. Each EC is exclusively attached to one user guest and saves its execution state such as the CPU/FPU register state, stack location, and scheduling sequence. By saving and resuming its EC, a given task can be scheduled. Since EC is governed by Mini-NOVA in kernel space, it is protected from any attacks from user space and thus guarantees the isolation and security of each guest. Normally user guest are not authorized to perform sensitive operations (i.e. page allocation, thread creation, cache operation, etc.), which should be handled by Mini-NOVA by generating system calls.

The main features of Mini-NOVA are:

- Small TCB size (3.5 KLOC in total);
- Multiple system calls and IRQs provided to user guests to handle privileged operations;
- Separate virtual address spaces for kernel and guest, and separated execution environment for each user application;
- Specific Priority-based round-robin scheduling to support DPR;
- Specific user service Hardware Task Manager.

B. Scheduling Strategy

Mini-NOVA implements a priority-based round-robin scheduling mechanism, which permits the user guests with the same priority level to equally share the CPU resource, while the higher priority application can always preempt the lower ones. The scheduler of Mini-NOVA schedules the user guests by manipulating their Execution Contexts (EC). Each EC is assigned with a fixed priority level value at its creation. Basically, all general guest applications are given the same priority level (1 by default) and occupy the CPU in turn. Kernel scheduler allocates to each EC a fixed time quantum, and forces it to switch to the successor when its time slot is used up.

However, based on the consideration that hardware tasks always require tighter time constrains, a quick response to hardware task management should be guaranteed. Thereby, we introduce higher priority levels to the specific services which requires hard real-time constrain, such as the HW Task Manager. In this case, once scheduled, it can always preempt lower priority users can execute immediately.

As presented in Fig.5, the scheduling strategy is implemented by managing the *run_queue* list, which is composed of all executable ECs. ECs at the same priority level are organized as double-linked queues. Multiple priority levels may coexist in the *run_queue*, while the CPU is always occupied by the highest level ECs. Kernel functions *Enqueue()* and *Dequeue()* are used to add/delete certain EC to/from the *run_queue*. Whenever the *run_queue* is changed the kernel always invokes the *reschedule()* function to re-pick the highest priority level EC. For example, initially the HW Task Manager is created with the priority level 2 and not included in *run_queue*, while general applications equally shares the CPU. (Fig.5 (b)) However when HW task management is required, the kernel adds the HW Task Manager service into the *run_queue*, and invokes *reschedule()* to dispatch it as higher priority EC. After the requirement is properly handled, the *Dequeue()* is called to remove the HW Task Manager from the *run_queue* and the interrupted round-robin scheduling is permitted to continue. This strategy ensures a quick response to any hardware task requirement.

C. *Hardware Task Manager*

Hardware Task Manager is proposed as a user service provided to guest applications. It cooperates closely with the microkernel and is responsible for the DPR management in our system. As described in Section III, hardware tasks, or DPR modules are stored as bitstreams in the DDR memory, which is only accessible by the Hardware Task Manager. Any request to reconfigure or dispatch hardware tasks are governed and performed by the HW Manager, so that the hardware task resources are isolated from other user guests, ensuring the the security of the FPGA fabric. To facilitate general user guests to requires for hardware task management, we provide a specific system call to call the HW Task Manager, whose prototype is:

Syscall_HW_Manager (HW_task_id, arg01, arg02, arg03)

Guests may invoke this system call to require for the HW

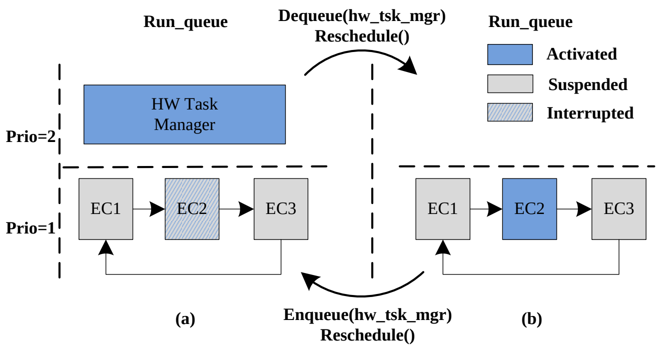

Fig. 5. Scheduling Strategy. (a) Preemptive scheduling; (b) Round-robin scheduling

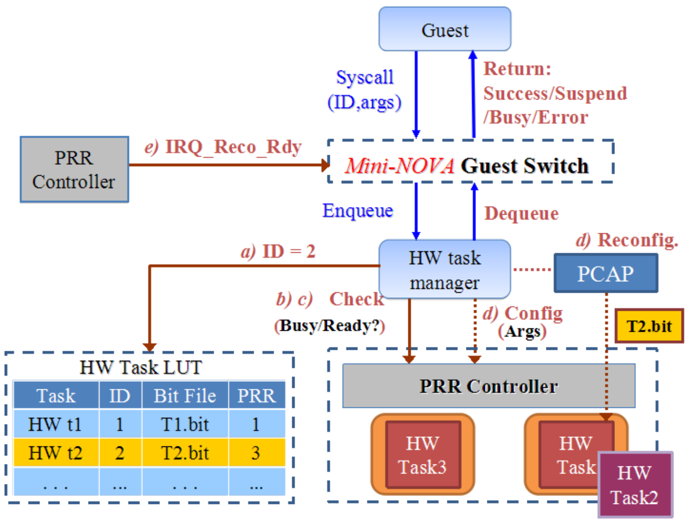

Fig. 6. HW Task Manager Process

task it desires to implement, by indicating the ID number of the target hardware task, and the initial parameters it would like to set to the register group of the task (as described in Section III). The calling process is demonstrated in Fig. 6.

As we described, the HW Task Manager is initially not activated and stays in suspension. On receiving the system call, Mini-NOVA enqueues the HW Task Manager to preempt the caller guest, while passing the target hardware task ID and arguments to the Manager, too. HW Task Manager then handles the caller's requirement by reconfiguring the desired HW task. Then the HW Task Manager generates another system call and dequeues itself from the *run_queue*, giving back control to the interrupted caller guest. Different return value (*Success*, *Busy*, *Suspend* or *Error*) is also returned to the guest to indicate the status of its requirement. The detailed process sequence is listed as following:

a) First, according to the HW task ID, the HW Manager walks through the hardware task LUT to get the information of the target HW task, i.e., its container PRR, and the address of its bitstream file.

b) Then the HW Manager checks the state of the PRR container, verifying if its available to be reconfigured. If it is currently occupied by another guest, then the HW Manager quits execution and returns to the caller guest as Busy, meaning that its requirement can't be handled right now.

c) In other cases, the container PRR is available but not ready to be reconfigured yet, since it may be in the middle of data processing or a running pipeline, which status is indicated by the *PRR_Reco_Rdy* bit in the PRR register group. In this case the HW Manager suspends itself and gives up the CPU control to the caller guest with Suspend.

d) If the container PRR is ready to be reconfigured, then the HW Manager writes the arguments to the register group and launches a PCAP transfer to download the target bitstream file from the DDR into the container PRR. Then it returns to the caller guest with the flag *Success*.

e) Following step *b)*, when the HW Manager is suspended waiting for the target PRR to be ready for reconfiguration, the PRR controller keeps monitoring the target PRR and generates an IRQ (*IRQ_Reco_Rdy*) to acknowledge the HW Manager as soon as PRR is ready (i.e. at the completion of data frame or pipeline). When receiving this IRQ, microkernel resumes the HW Manager to complete the PRR's reconfiguration, repeating step *d)*.

We should note that the DPR overhead remains an major drawback for embedded systems. As a solution, after launching a PCAP transfer, we abort the polling-for-done mechanism, meaning that the CPU control is directly given back to the guest without waiting for the reconfiguration completion. The PCAP completion can be acknowledged via PCAP interrupt or the guest checking PCAP state.

V. Use-Case Study

To verify and evaluate the architecture we proposed, a use case based on real application scenario is studied in this section. In the use case, a mobile wireless terminal alters the configuration of its communication modules to adapt better to the channel conditions, which is implemented by dynamically reconfigure the hardware accelerators. For example, according to the condition of the noise in the channel, different QAM modulations are required by the transmitter so that the throughput can rapidly adapt to the environment.

A. *Implementation*

The implementation is depicted in Fig.7. In the CPU user space, the *Channel_Sensor* keeps estimate the best level of performance in terms of throughput and error rate. It invokes the HW Task Manager via system call whenever it decides to alter the hardware task modules to adapt to the channel condition. In the FPGA fabric, two hardware blocks *HW_QAM* and *HW_IFFT* are running in pipeline, which respectively handles the modulation scheme and the IFFT in the OFDM context. For both modulation and IFFT blocks, several optional hardware tasks are provided: three constellation-sizes QAM modules (4, 16 and 64) and IFFT of different points (from 256 to 8192 points).

Note that, since QAM and IFFT blocks run in pipeline, the reconfiguration of either one will stall the pipeline and thus significant overhead. Thereby, we introduced a multi-path structure, implementing a pair of identical PRRs to QAM and IFFT receptively, so that during the reconfiguration the pipeline continues because only the idle PRR is being reloaded. Thus the DPR overhead can be overlapped by the pipeline execution.

B. *Result and evaluation*

We obtained the evaluation result under the following settings: 100 MHz FPGA Clocking, 18,800 Bits data frame size. The execution of different tasks has been recorded in the Gantt chat in Fig. 8. Initially, a QAM4 modulation scheme (PRR0) and a 256 points I-FFT (PRR2) are running, when the *Channel_Sensor* calls the HW Task Manager to switch I-FFT mode to 512 points for better performance (t_1-t_2). Then while PRR1 continues running, a PCAP transfer is launched to load

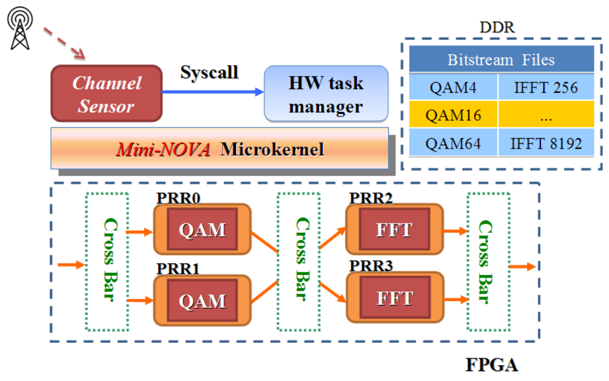

Fig. 7. Use-case implementation

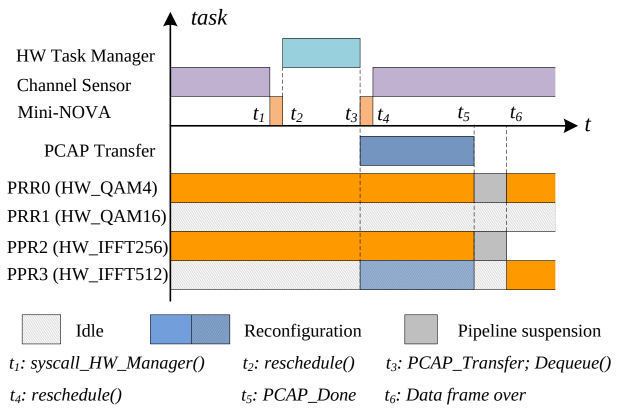

Fig. 8. Use-case execution Gantt chat

the *HW_IFFT512* module to PRR2, which is currently idle (t_3-t_5). At the completion of PCAP transfer (t_5), the new IFFT task has been implemented in PRR2, but the pipeline goes to a suspension to ensure the currently-processed data frame is completely processed (t_5-t_6). Then the *HW_IFFT512* is activated and the new pipeline starts to execute (t_6).

We also measured the performance of HW task manager through large number of iterations of different cases, and the result is listed in TABLE II. EC Switch measures the response time from the guest's requirement to the HW Task Manager's reply. Due to the low complexity and scheduling strategy of Mini-NOVA, the hw task requirement can be answered within 0,0023 ms. We also should note that although the DPR overheads of hardware modules are significant, the pipeline

TABLE II. Performance of HW Task Management (ms)

Task	Execution	Reconfig.	Resource
Channel_Sensor	3	/	/
HW Task Manager	0.0096	/	/
EC Switch	0.0023	/	/
Pipeline suspension	0.03-0.168	/	/
HW_QAM (4/6/64)	0.03-0.09 /frame	0.231	2%
HW_IFFT(256-8192)	0.006-0.168/frame	2.71	13%

suspension is limited (0.168 ms in worst case) because pipeline is not stalled during the reconfiguration. The advantage of DPR technology can be proved by the consumed FPGA resources. For example, implemented by Xilinx Planahead synthesis tool, the computing-intensive IFFT module takes up massive FPGA resources (i.e. 5600 LUTs and 1600 SLICEs for 8196 points IFFT). With static FPGA circuit, implementing IFFT modules from 256 points to 8196 points consumes up to 50% FPGA area, while in our system, by reusing the DPR fabric, only 26% resources (2 PRRs) are used. Thus the chip cost is significantly reduced.

VI. Conclusion

In this paper, we have proposed a microkernel based on the ARM-FPGA architecture, devoted to an efficient management of dynamic partial reconfiguration. Specific architectures and scheduling strategy have been introduced to the microkernel for better performance and higher security. In the use case study, we have evaluated our system with practical applications and analyzed the results, which proved that our microkernel system is able to manage SW/HW tasks and minimize the performance degradation caused by the DPR overhead.

References

[1] C. Beckhoff, D. Koch and J. Torresen, “Go Ahead: A partial reconfiguration framework,” in 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2012, pp. 37-44.

[2] G. Heiser, “The role of virtualization in embedded systems,” in Proceedings of the 1st workshop on Isolation and integration in embedded systems, ACM, 2008, pp. 11-16.

[3] UG585: Zynq-7000 All Programmable SoC Technical Reference Manual, Xilinx Inc., Mar. 2013.

[4] S. Hauck, and A. DeHon, “Reconfigurable computing: the theory and practice of FPGA-based computation,” Morgan Kaufmann, 2010.

[5] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, “Designing an operating system for a heterogeneous reconfigurable SoC,” in Proceedings of International on Parallel and Distributed Processing Symposium, 2003, IEEE, 2003, pp. 174-180.

[6] K. Vipin, and S. A. Fahmy, “A high speed open source controller for FPGA Partial Reconfiguration,” in FPT, 2012, pp. 61-66.

[7] D. Gohringer, M. Hubner, E. N. Zeutebouo, and J. Becker, “Operating system for runtime reconfigurable multiprocessor systems," International Journal of Reconfigurable Computing, vol. 2011, January 2011.

[8] C. Steiger, H. Walder, and M. Platzner, “Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks,” IEEE Transactions on Computers, vol. 53, no. 11, pp. 1393-1407, Nov. 2004.

[9] K. Danne, R. Miihlenbernd, and M. Platzner, ”Executing hardware tasks on dynamically reconfigurable devices under real-time conditions,” in FPL’06, IEEE, 2006, pp. 1-6.

[10] K. Vipin, and S. A. Fahmy, “ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq,” in IEEE Embedded Systems Letters, 2014, vol. 6.

[11] J. Agron, and D. Andrews, “Building heterogeneous reconfigurable systems with a hardware microkernel,” in Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, ACM, 2009, pp. 393-402.

[12] G. Heiser, ”The role of virtualization in embedded systems,” in Proceedings of the 1st workshop on Isolation and integration in embedded systems, ACM, 2008, pp. 11-16.

[13] K. Dang Pham, A. K. Jain, J. Cui, S. A. Fahmy and D. L. Maskell, “Microkernel hypervisor for a hybrid ARM-FPGA platform,” in ASAP, IEEE, 2013. pp. 219-226.

[14] U. Steinberg and B. Kauer, ”NOVA: a microhypervisor-based secure virtualization architecture,” in Proceedings of the 5th European conference on Computer systems, ACM, 2010, pp. 209-222.