HAL
open science

Microkernel dedicated for dynamic partial reconfiguration on ARM-FPGA platform

Tian Xia, Jean-Christophe Prévotet, Fabienne Nouvel

To cite this version:

Tian Xia, Jean-Christophe Prévotet, Fabienne Nouvel. Microkernel dedicated for dynamic partial reconfiguration on ARM-FPGA platform. The 4th Embedded Operating Systems Workshop (EWiLi'14), Nov 2014, Lisbon, Portugal. pp.31 - 36. hal-01113215

HAL Id: hal-01113215

https://insa-rennes.hal.science/hal-01113215v1

Submitted on 4 Feb 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Public Domain

Microkernel Dedicated for Dynamic Partial Reconfiguration on ARM-FPGA 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

This paper describes the first implementation of a custom micro-kernel on a ARM-FPGA platform capable of managing reconfigurable hardware parts dynamically. After describing the structure of the proposed micro-kernel, we will focus on a custom specific system task dealing with the reconfiguration management, which is associated to a dedicated scheduling mechanism. We will describe the hardware platform on which the microkernel has been ported and provide a use case application in order to demonstrate the feasibility of the approach. At the end of this paper, we will provide quantitative results in terms of reconfiguration overhead and microkernel timing performances.

Keywords

Microkernel, Real-Time Systems, FPGA, Embedded System, Reconfigurable Architectures

1. INTRODUCTION

During the last decades, with the development of commodity field-programmable gate array (FPGA), the technique of reconfigurable computing has gained increasing attention for its potential in exploiting hardware resources. Through time-multiplexed sharing of the FPGA fabric, a higher integration of functionalities can be achieved. The main drawback of traditional FPGA reconfiguration computing is the lack of flexibility, because the whole fabric is required to be reconfigured even when modification is only required for a part of the FPGA. As a consequence, enormous time overhead and power consumption are produced, which severely limits reconfiguration in embedded systems.

As a solution, a more advanced technique enabling to reconfigure particular areas of an FPGA while the rest continues executing has been proposed and is known as Dynamic Partial Reconfiguration (DPR). This technique has proved to be quite prospective in the embedded domain because of its runtime adaptivity for hardware algorithms and lower power consumption compared to large-scale static circuits [1]. With DPR feature, hardware accelerators can be dynamically dispatched and managed, becoming as flexible as software functions.

On the other hand, with the widespread applications of handheld devices, reliability and security of embedded systems have become a serious concern. Dealing with microkernels constitutes a promising idea because it allows the user to execute various applications (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 for many years[2].

In this paper, we describe and study a custom embedded microkernel on a hybrid ARM-FPGA Zynq-7000 platform [3]. This microkernel is a revised version of the NOVA microhypervisor [4], and is integrated with the management and scheduling of reconfigurable hardware resources. This proposed architecture allows for dynamic management of SW/HW tasks, secure task isolation and efficient SW/HW communication.

The remainder of the paper is organized as follows: Section 2 presents current researches in management of DPR architectures. In Section 3, an overall architecture of the proposed platform is introduced. Section 4 focuses on the design and implementation of the microkernel, with detailed introduction to the hardware tasks management and scheduling mechanisms. In Section 5, we present a case study to demonstrate the capabilities of the proposed microkernel. Finally section 6 concludes the paper.

2. RELATED WORK

Compared with the traditional full reconfiguration mechanism, the DPR technique benefits from the following major advantages [3]:

- Reduced hardware resource utilization
- Improved design efficiency
- Reduced reconfiguration latency and better robustness

Despite of the enhanced flexibility provided by DPR techniques, the reconfiguration overhead remains a crucial issue in practice. In modern high-end FPGAs which may have tens of millions of configuration points, one reconfiguration of a complex module will be very time-consuming. Numerous studies have been led to propose efficient hardware reconfiguration management with dedicated architecture and OS support. A custom DPR controller was introduced in [5] to realize high-speed on-chip reconfiguration. In [6], a

EWiLi'14, November 2014, Lisbon, Portugal.
Copyright retained by the authors.

specific operating system CAP-OS was proposed to provide clients with hardware task management and priority-based scheduling. Other researches were made in the OveRSoC project, which provided a model at high-level abstraction and allowed to efficiently simulate and validate embedded RTOS for reconfigurable platforms [7]. Most researches on traditional DPR devices (i.e. Virtex FPGA family) employ embedded processors such as MicroBlaze or PowerPC, whose computing ability is relatively limited.

Compared to classical devices, the Zynq-7000 platform integrates the ARM Cortex-A9 processor with various on-board resources and brings up enormous possibilities for embedded techniques. In this platform, the programmable fabric is considered as a unique auxiliary computing resource to this fully capable processing system, and the reconfiguration management is expected to be one of many tasks in the system. Hence, a specific kernel is the ideal solution to rationally dispatch both hardware and software resources.

While considerable efforts have been made to port microkernel techniques to traditional embedded systems, such as the OKL4 from Open Kernel Labs [2], most existing microkernels on ARM do not consider reconfigurable hardware. Instead, most of the works only use a micro-kernel to manage heterogeneous platforms i.e. software and static hardware parts. For example, in [8], a L4 kernel is ported to manage hardware and software tasks, but without using dynamic reconfiguration. In parallel, research in [9] discussed the reconfiguration management on Zynq platform at the application level, without using any operating system and thus with poor flexibility.

3. PROPOSED PLATFORM

The motivation of the proposed hybrid ARM-FPGA platform framework is to establish a user-practical environment with a highly abstract microkernel. The management of hardware resources is integrated as a user application, with relatively easy access. Both software and hardware tasks are registered and scheduled by a custom microkernel. A block diagram of the proposed platform is shown in Fig. 1.

On the proposed platform, computing resources are divided into Processing System (PS) and Programmable Logic (PL). On the PS side, a simplified microkernel hosts multiple software applications, including Guest OSes and user applications executing within the user space. Each application is housed in an individual isolated space of the microkernel, which is referenced as an execution context (EC). By scheduling and switching ECs, the ARM processor is shared among guests according to time multiplexing. The

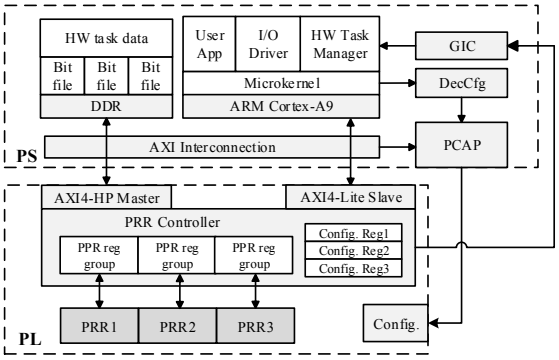

Figure 1: Diagram of the Proposed Hybrid Platform

FPGA fabric is engaged in several hardware acceleration operations, which are executing concurrently with SW tasks. A specific hardware task management routine is proposed to control and reconfigure the hardware accelerators dynamically. Such a routine runs as a guest to the microkernel and is scheduled whenever the management of HW tasks is required. The mechanism related to this part will be described in detail in Section 4. In this way, the FPGA resource is seen as a standard user application by the microkernel and thereby hardware and software tasks can be managed concurrently in our framework.

3.1 Hardware Tasks

In a reconfigurable embedded system, hardware tasks are implemented using functional fabric structures in the FPGA, which can be user-defined computing blocks or commercial IP cores. As shown in Fig. 1, the FPGA fabric is divided into multiple partial reconfigurable black boxes or containers, which are capable of housing hardware tasks independently. These containers are defined as partial reconfigurable regions (PRR). The hardware task which is running in each container is run-time switchable under the control of the hardware task manager. Different sizes of blocks are allocated to different PRRs for different task purposes.

The resource that holds the fabric information of hardware tasks is contained in a bitstream file. Different bitstream files can be stored in various memory devices and be accessed via a simple look-up table. Note that, the container corresponding to each HW task has always the same constrained location in the FPGA. A HW task is dispatched by transferring the corresponding bitstream file to the assigned PRR. Normally, HW tasks with similar or close functionalities should be distributed to the same PRR, so that the coherence of HW task interfaces can be guaranteed. Each HW task should have one corresponding SW application to monitor and control its behaviour.

One of the crucial features regarding hardware tasks is the reconfiguration overhead, which is linearly correlated to the size of the bitstream, thus, the PRR size. This means all HW tasks implemented in the same PRR will have the same time overhead for reconfiguration.

3.2 HW/SW Task Communication

To connect PL with PS, two interface types based on the standard AXI bus protocol are employed. Offering a unified mapping to the processor and being accessed as a normal memory access, the AXI_GP is intended for low-speed general purpose communication. As in Fig. 1, the processing system takes control of two master AXI_GP interfaces as main methods to configure and read back the states of the HW tasks.

AXI_HP is aimed for high performance data exchange with burst transfer, which may transfer data blocks as large as 4KB in one burst, and is sufficient for generic data processing applications. On our platform, 4 AXI_HP interfaces are used and in charge of accessing both on chip memory(OCM) and DDR. Since HW tasks access AXI_HP as masters, data is fetched and written back without acknowledging the processor, allowing the processor to run simultaneously with HW tasks.

3.3 Reconfiguration Interface

Two methods for partial reconfiguration are supported on

Table 1: Description of PRR Configuration Registers

Reg Name	Width	Description
Proc_status	16	Mark process status: Bit[0]: start data processing Bit[1]: pause data processing Bit[2]: interrupt handling over
PRR_Int_status	16	Mark interrupt status: Bit[0:7]: PRRs interrupt enable Bit[8:15]: PRRs interrupt status
PRR_status	16	Mark PRR enable status: Bit[0:7]: PRRs enable Bit[8:15]: PRRs switch_enable.
PRR_Reco_rdy	16	PRR's status for reconfiguration: Bit[0:7]: PRR ready
PRR_delay	32	Time overhead for current reconfiguration
PRR_gpr[7:0]	32	General-purpose registers defined by user: HW task ID, working mode, parameters, etc.

the Zynq platform: Processor Configuration Access Port (PCAP) and Internal Configuration Access Port (ICAP). Using PCAP, as shown in the datapath of Fig.1, PS is enabled to initialize bitstream transfers from memory to PL through the Device Configuration Interface (DevCfg) at high throughput (130MB/s). In contrast, ICAP is designed for self-configuration from the PL side with a AXI4-Lite as transfer port. Such a mechanism severely limits the reconfiguration speed (19MB/s). ICAP is less interesting also because it requires additional hardware resources and will occupy at least one AXI interface. On our platform, PCAP is selected for its better compatibility with software applications and higher throughput.

3.4 PRR Controller Block

As shown in Fig. 1, a PRR controller block is introduced to monitor and manage the states of HW tasks. This block runs as a state machine under the supervision of the HW task manager. Through the AXI_GP interface, we have implemented a group of configuration registers(PPR reg group) which are mapped into memory space and accessible to the processor. By configuring these registers, a SW service is able to set up HW tasks, such as defining working modes, and data address. Since the number of PRRs is pre-fixed, we provide each PRR a PPR reg group for configuration. The context of the registers is left for user-definition to adjust to different HW tasks. Table 1 describes the configuration of this register group.

3.4.1 Reconfiguration security

In case of a PRR reconfiguration, a switch of HW task is normally required. The PRR Controller is proposed to guarantee the HW task security, avoiding invalid data output and undesired task state. Based on these considerations, following features are included:

- In case 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, the PRR controller avoids reconfigurations interrupting of data frames.
- A reset should be asserted to initialize the reconfigured PRR before being allowed to be activated.

3.4.2 Interrupts Management

The PRR controller is able to generate general-purpose interrupts through the Shared Peripheral Interrupts (SPI) connected to the generic interrupt controller (GIC). 8 SPI resources are used to provide the PS with different HW task information such as task completion or critical errors.

4. REAL TIME MICROKERNEL

To facilitate the management of multiple guest SW applications and HW tasks, we developed a simplified microkernel based on Mini-NOVA, one revision of the NOVA hypervisor. In this section, we propose a specific HW task manager service and a scheduling strategy to support dynamic PR management.

4.1 Microkernel Description

The microkernel runs on top of bare-metal hardware. By implementing the basic OS functionalities, the microkernel establishes an abstraction layer of the hardware platform to user applications. The application of microkernel benefits in the way that higher security level can be achieved with virtualization technique. One of the essential features of this microkernel is security, so the principle of least privilege is strictly followed in our framework to make sure that a minimal tested computing base (TCB) is achieved. Such a feature will also improve performance with a quicker execution of context switches.

The proposed microkernel has simplified functionality and reduced complexity, which makes it more suitable for embedded systems and also more adaptable. Since the initial Mini-NOVA is designed for x86 architecture, several modifications have been made to execute on the ARM Cortex-A9 which is available on the Zynq-7000 platform. Besides, additional mechanisms and a new scheduling strategy have also been provided to the system. The main features of the proposed microkernel are:

- Modified bootloader and boot sequence for both Zynq platform (e.g. FPGA initialization, DDR initialization, etc.) and ARM Cortex-A9 processor(e.g. kernel boot, user boot, paging table, exception vector, etc.)
- Separate virtual memory mapping for kernel and user space while providing isolated execution context for each user application
- System calls and IRQs provided to user applications
- Specific Priority-based round-robin to support PR
- Supporting virtualized OS(e.g. uC/OS-II)

We should note that, to minimize the TCB size of the kernel and guarantee system security, most board-specific support APIs and services are implemented in user space, including HW task manager, AXI support, and supports for on-board peripheral resources (UART, SD card, interrupt controller, TCC Timer, etc.).

The virtual memory space of our system is divided into several domains. As described in Table 2, the kernel space and user space are access-isolated by virtual mapping. A range of 256MB memory space on the upper side is distributed to the microkernel, whereas user applications execute in the lower memory space. Besides the user space and kernel space, an extra space up to 256MB is allocated to store the bitstream files dealing with HW tasks. This area is programmed to be only accessible from the user space.

The execution context (EC) is the major kernel object, which is the abstraction of user threads or applications in

Table 2: System Address Mapping

Name	Addr Range	Accessibility	Description
Kernel	0xC0000000 - 0xDFFFFFFF	Kernel	Kernel space
User	0x0 - 0x2FFFFFFF	Kernel, User	User space
HW Task	0x30000000 - 0x3FFFFFFF	Kernel, User, PL	Bitstreams, HW task data
PL	0x40000000 - 0xBFFFFFFF	User (AXI_GP)	PL Memory Space
Peripheral	0xE0000000 - 0xFDFFFFFF	Kernel, User	Platform and Peripheral regs

Table 3: Structure of the *bit_descriptor* Class

Obj. member	id	addr	len	delay	prr_id
Contents	HW task ID	Bitfile Address	Bitfile Length	Reconfig. Overhead	PRR ID

the kernel space. Each EC is exclusively attached to one user application and is able to maintain and manipulate user applications' features such as the CPU/FPU register state, stack location, and scheduling sequence. By resuming its EC, a given task can be completely restored. When sensitive operations (page allocation, thread creation, cache operation, etc.) are required, the user space may access the kernel services by generating system calls, which are also handled through an EC.

4.2 HW Task Manager

The HW task manager is defined as a special user application serving other applications. Though executed in user space, this service cooperates closely with the kernel and is an essential part of the PR control flow in the system. In the following, we describe its different features.

4.2.1 Bitstreams Management

The switch of HW tasks is based on the download of different bitstream files. As introduced in Section 3, each bitstream corresponds to one HW task, and its PRR container is pre-fixed (but not exclusive), which also determines its reconfiguration overhead. All bitstream files are loaded to the HW task memory space shown in Table 2, at the kernel bootload stage. A descriptor is provided to each available bitstream file by defining a *bit_descriptor* class. We also created a look-up table for all *bit_descriptor* objects indexed by a unique ID number. In fact, the object members given in Table 3, *bit_descriptor::id* is the only information that a normal user application should know about HW tasks. Other pieces of information such as location and length are only used by the HW task manager.

4.2.2 Calling the HW Task Manager

Any attempt to dispatch, reconfigure, modify or disable HW tasks should be accomplished by the HW task manager. In other words, operations towards HW tasks are isolated from other user applications. We employed this mechanism to ensure the security of the FPGA fabric. For user applications which are cooperating with HW tasks, the only accessible memory space is the HW task data section, which is used for massive SW/HW data exchange.

As described in Section 3, the behavior of HW tasks are controlled by writing parameter values to their corresponding PRR configuration registers, for which the contexts of parameters are defined by user application and are not in the concern of the HW task manager. All the information required by the HW manager are the ID of HW task and the arguments to be transferred to the register group.

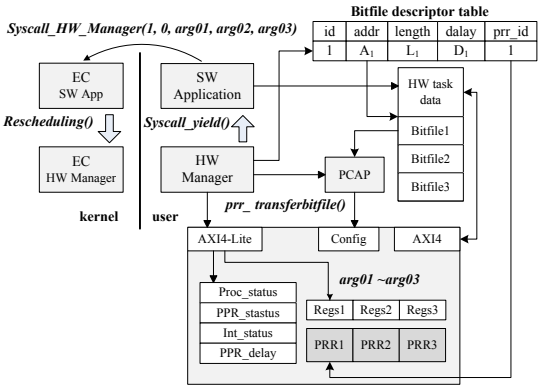

Figure 2: Execution of the HW Task Manager

A block diagram describing the execution of the HW task manager is shown in Fig. 2. As demonstrated, a specific system call from user space will require the kernel to launch the HW task manager. Arguments are passed through to the HW manager. The prototype of this specific system call is:

Syscall_HW_Manager(HW_id, irq_en, arg01, arg02, arg03)

By handling this system call, the kernel invokes a reschedule process and returns to user space, passing control to the HW task manager. In this process, arguments are also delivered to the HW manager. The HW manager will compare the *HW_id* with the executing HW task. If it is already implemented in PRR, then only the parameters are changed by writing arguments to the register group, otherwise a PCAP transfer will be configured to reload the target PRR with the desired HW task. The *irq_en* argument will indicate whether the PL interrupt is enabled for the corresponding PRR by setting values in the *PRR_Int_status* register. After accomplishing the required operation, the HW task manager gives back control to the previously interrupted application.

In some cases, a PR request cannot be acknowledged immediately. As the scenarios described in Section 3, a HW task may be in the middle of a data frame process and not ready for reconfiguration. In such situations, to avoid monopolizing the CPU, the HW task manager will be pulled up and give up its CPU usage to other SW applications. When the data frame is completely processed, the target PRR informs the HW task manager by triggering an IRQ *IRQ_Reco_rdy*, then the service will be relaunched to start the PCAP bitstream download.

One major drawback of the PR technique is its significant reconfiguration time overhead. To reduce its effect on performance, we abort the polling-for-done mechanism. Instead, the completion of a PCAP transfer is not acknowledged to the HW task manager. Once the HW task manager launches the PCAP transfer, it gives up the CPU control and wait for the next call. A HW task is set to automatically start an operation as soon as reconfiguration is done, thereby the reconfiguration time overhead is overlapped by CPU operations. SW applications are able to be synchronized with a HW task state by its general-purpose IRQ. This functionality is enabled by the *PRR_Int_status* register. For example, imagine a simple application with an image displayer SW task that is using a HW Image filter accelerator. It will fetch the target image and write the results back to memory through AXI4 automatically. Once the image processing is

Table 4: HW Task Manager API

API	Description
XDcfg_Initialize();	Instantiate DevCfg
AXI4_lite_Init();	Instantiate master AXI4-lite
fpga_start(); fpga_pause(); fpga_interrupt_done()	Pass signals to whole fpga fabric by writing values to corresponding PPR controller regs.
check_current_ppr(HW_id)	Check current implemented HW tasks' IDs to determine whether PR is necessary.
check_reco_rdy(HW_id)	Check if target PRR is ready for PR, if not, use sys_yield() to quit HW manage.
prr_set_mode(HW_id,irq_en, arg01,arg02,arg03)	Set up PRR_Int_status and register group of specific PRR.
prr_transferbitfile(HW_id)	Launch PCAP to transfer target bitstream.
prr_register_read(off,val) prr_register_write(off, val)	Basic access method to all prr controller registers by master AXI4-Lite

finished, the HW filter will generate an IRQ to inform the displayer task that another operation can be executed.

4.2.3 HW Task Manager API

The driver API of DevCfg is supported by the Xilinx SDK tool, which deals with the non-secure/secure PCAP transfer. Besides of the DevCfg API, several additional functions are developed to facilitate and simplify the HW management. In Table 4, the API supporting HW task management is listed and described.

4.3 SW Tasks Scheduling

The scheduling strategy of SW tasks in Mini-NOVA is a priority-based round-robin mechanism. The scheduler manages the execution sequence by manipulating ECs. Each EC obtains its own priority level at its creation, which is changeable afterwards. Within the same priority level, SW tasks share the CPU through round-robin scheduling. Among different priority levels, high-priority tasks will always preempt low-priority tasks since the scheduler always selects the highest priority EC and dispatches the SW task attached to it.

Basically, all general SW tasks execute at the same priority level (1 by default). However, to fulfill the timing constraints for specific requests such as real-time tasks and PR requests, different priority levels are introduced. In this case, specific tasks should be of higher priority so that they can be dispatched in time. Since our current system mainly deals with HW management, only the HW task manager is being discussed here.

Fig. 3 presents the scheduling mechanism based on priority. At each priority level, ECs are organized as a double-linked queue, which is indexed by a *list_prio[]* structure. *list_prio[]* is a list of EC pointers indexed by a priority level. Each *list_prio[]* element points to a certain priority level EC queue. The *run_queue* is composed of different priority level EC queues, and the *prio_top* signal identifies the highest priority level in current *run_queue*. When *reschedule()* is invoked, *prio_top* is used to access the highest-priority-level EC queue by dispatching *list_prio[prio_top]*. Once dispatched, the queue will keep executing until another *reschedule()* is invoked.

As shown in Fig. 3, the EC of the HW task manager is registered in the microkernel at its creation with a default priority level 2. Initially, the HW manager is not included in *run_queue* as Fig. 3(a). When *Syscall_HW_Manager()* is executed, the microkernel will launch *HW_Manager_Enqueue()* to add the HW task manager into the *run_queue* as shown in Fig. 3(b). Then, the *reschedule()* function is launched to update the schedule and dispatch the HW task manager as the highest priority EC by selecting *list_prio[2]*. When the HW task manager finishes its task or enters the pull-up s-

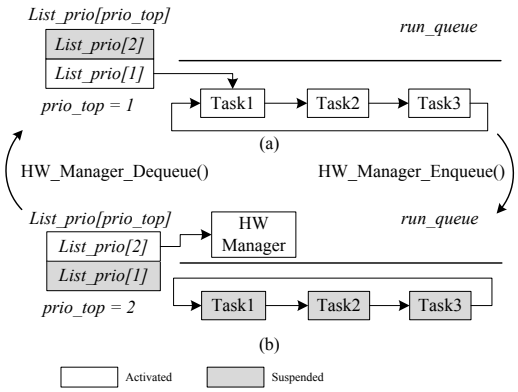

Figure 3: Microkernel Scheduling Mechanism. (a) *prio_top*=1; (b) *prio_top*=2

tate, *HW_Manager_Dequeue()* is called to remove the EC of the HW task manager from the *run_queue*, as shown in Fig. 3(a), thus low-priority SW tasks are permitted to execute. Through this strategy, the PR of an HW accelerator is able to preempt other SW tasks and a quick response for the HW task management is guaranteed.

5. USE-CASE IMPLEMENTATION

In order to test the SW/HW scheduling mechanism on the platform, a use-case application based on a real scenario has been proposed. In this scenario, a mobile wireless terminal is capable of dynamically change its configuration in order to obtain the best level of performances according to the channel conditions. For example, if the channel is very noisy, the transmitter will deal with a simple but very efficient QAM modulation to the detriment of the throughput. As soon as the channel conditions allow to increase the throughput, the mobile device may reconfigure itself to change its inner hardware modulator and rapidly adapt to the environment.

5.1 Implementation Description

In the proposed use-case scenario, the application is divided into two main software tasks running on the processor and two additional hardware tasks running in the FPGA.

The *SW_ChannelSensor* task performs a channel estimation in order to evaluate the maximum level of performance to be obtained in terms of throughput and error rate. The *SW_HardwareManager* is an instance of the HW task manager, as described in Section 4.

Concerning the hardware parts, two reconfigurable HW tasks sets have been considered which respectively deal with the modulation scheme and the IFFT used in the OFDM context. The *HW_Modulation* task deals with the nature of modulation to be implemented i.e. the constellation size. In this work, three constellations sizes have been considered: 4-QAM, 16-QAM and 64-QAM. Regarding the second hardware task, *HW_IFFT*, several configurations have also been implemented according to the number of points to consider. In our application, a range of number of points for I-FFT (from 256 points to 8192 points) was implemented depending on the channel bandwidth to be considered. All HW task execute in their corresponding PPR (PRR0 - PRR3).

Since *HW_Modulation* and *HW_IFFT* execute in pipeline, the reconfiguration of these HW tasks will suspend the entire pipeline. To minimize the significant time overhead, we propose a multiple-path structure. A block diagram depicts this

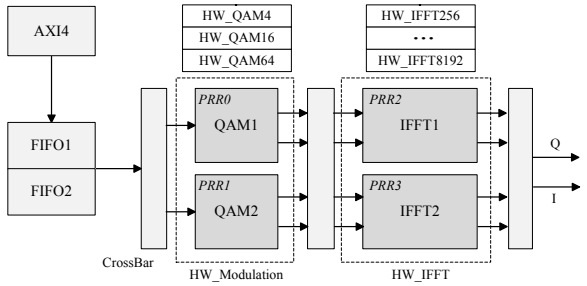

Figure 4: Use-Case Implementation

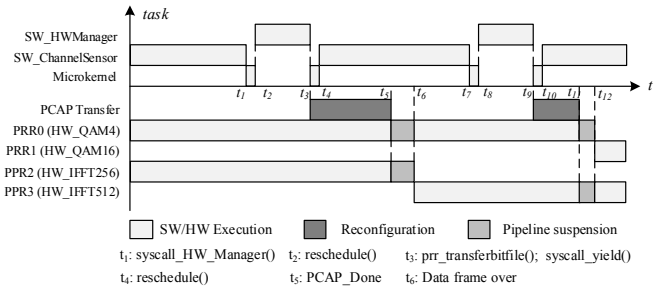

Figure 5: Gantt Chart of the Tasks' Execution

structure in Fig. 4. Both *HW_Modulation* and *HW_IFFT* consist of a pair of identical PRRs. While the current PPR continues working, *SW_ChannelSensor* may alter the HW task by reconfiguring the other PRR, and activating the new datapath after reconfiguration. Thus, the overhead caused by reconfiguration is reduced.

With a 18,800 bits data frame size and 100MHz FPGA clock frequency, a Gantt chart for the result of proposed scenario on our platform is given in Fig.5. The application begins with the *SW_ChannelSensor* task deciding to change the hardware configuration because the the channel's conditions are not suitable for the default configuration (a QAM4 modulation scheme and a 256 points I-FFT). In this case, the task calls the *SW_HardwareManager* to manage its request of switching I-FFT mode to 512 points (t_1 - t_2). Since PRR3 is idle and ready for reconfiguration, *SW_HardwareManager* launches the PCAP transfer to implement *HW_IFFT512* to PRR3 while the QAM4-IFFT256 pipeline continues computing (t_2 - t_5). After the completion of PCAP transfer (t_5), the pipeline holds for currently-processed data frame to be completely processed (t_5 - t_6) before the *HW_IFFT512* is activated at t_6. The same procedure is executed again, when *SW_ChannelSensor* decides to switch from QAM4 modulation to QAM16 (t_7 - t_{12}). Some attributes of SW/HW tasks are listed in Table 5.

5.2 Discussion

As shown in the Gantt chart, the major overhead of reconfiguration is fully circumvented by both SW and HW tasks running in parallel. For data processing, the only overhead caused by the HW task switch is the delay required to process a complete data frame(worst case 0.168 ms, in case of 8096 points I-FFT). Due to the simplified kernel and scheduling mechanism, a quick response to PR is achieved (0.0119 ms). We should note that the tremendous reconfiguration overhead of I-FFT tasks result from the massive computing-intensive structure of I-FFT blocks. Implemented by Xilinx Planahead synthesis tool, it consumes 5600 LUTs and 1600 SLICEs, which takes up to 13% FPGA resources on chip. For static FPGA circuits, implementing multiple I-FFT blocks with different points will cost considerable FPGA area, while on our platform only 26% FPGA resources (2 I-FFT blocks) are used to hold multiple I-FFT blocks. Thus the chip cost is significantly reduced.

Table 5: SW/HW Tasks' Attributes

Task name	Type	Execution Time(ms)	Reconfig. Time(ms)	Resource Usage
SW_ChannelSensor	SW	3	no	no
SW_HW_Manager	SW	0,0096	no	no
EC_Switch	SW	0,00232	no	no
HW_QAM (4/16/64)	HW	0,09-0.03(1 frame)	0.231	2%
HW_IFFT (256-8192)	HW	0,006-0,168(1 frame)	2.72	13%

6. CONCLUSION

In this paper, we have presented a custom ARM-specified microkernel on a partially reconfigurable FPGA platform. This approach allows to dynamically manage reconfigurable HW accelerators and SW tasks by developing a specific scheduling mechanism. Efforts have been made to maximize the performance of the FPGA fabric and minimize the overhead caused by partial reconfiguration. We are currently working on the virtualization of guest OS. By implementing different OSes based on the microkernel, we intend to establish a complete virtualizable embedded system

7. REFERENCES

[1] D. Thomas, J. Coutinho, and W. Luk, "Reconfigurable computing: Productivity and performance," in *Asilomar Conference on Signals, Systems and Computers*, pp. 685–689, 2009.

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

[3] "Ug585: Zynq-7000 all programmable soc technical reference manual," Xilinx Inc., March 2013.

[4] U. Steinberg and B. Kauer, "Nova: a microhypervisor based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*, pp. 209–222, 2010.

[5] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *Field Programmable Logic and Applications*, pp. 535–538, IEEE, September 2008.

[6] D. Göhringer, M. Hübner, E. N. Zeutebouo, and J. Becker, "Operating system for runtime reconfigurable multiprocessor systems," *International Journal of Reconfigurable Computing*, vol. 2011, January 2011.

[7] J. C. Prevotet, A. Benkhelifa, and e. a. B. Granado, "A framework for the exploration of rtos dedicated to the management of hardware reconfigurable resources," in *International Conference on Reconfigurable Computing and FPGAs*, pp. 61–66, IEEE, 2008.

[8] K. D. Pham, A. K. Jain, J. Cui, and et al, "Microkernel hypervisor for a hybrid arm-fpga platform," in *24th International Conference on Application-Specific Systems, Architectures and Processors*, pp. 219–226, IEEE, 2013.

[9] K. Vipin and S. A. Fahmy, "A high speed open source controller for fpga partial reconfiguration," in *FPT*, pp. 61–66, IEEE, 2012.