Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators

Tian Xia, Jean-Christophe Prévotet, Fabienne Nouvel

Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators

Tian XIA
IETR
INSA de Rennes
Rennes, France
tian.xia@insa-rennes.fr

Jean-Christophe Prevotét
IETR
INSA de Rennes
Rennes, France
jean-christophe.prevotet@insa-rennes.fr

Fabienne Nouvel
IETR
INSA de Rennes
Rennes, France
fabienne.nouvel@insa-rennes.fr

***Abstract*—In the last decade, the research on CPU-FPGA hybrid architectures has become a hot topic. One of the main challenges in this domain consists in efficiently and safely managing dynamic partial reconfiguration (DPR) resources. This paper focuses on the management of the reconfiguration by an hypervisor on an ARM-FPGA platform. Using the virtualization approach, virtual machines (VM) may access resources independently, being unaware of the existence of other VMs. The purpose of our work is to provide an abstract and transparent interface for virtual machines to access reconfigurable resources. The underlying infrastructure of partial reconfiguration management is hidden from the virtual machines, so that software developers do not need to consider the implementation details. We propose a framework where DPR accelerators are presented as virtual devices, which are universally mapped in each VM space as ordinary peripherals. The hypervisor automatically detects VM's requests for DPR resources and handles them dynamically according to a preemptive allocation mechanism. Our custom hypervisor guarantees the independent and isolation of VM domains. We also evaluate the efficiency of our framework by measuring the critical overheads during DPR management and allocations. The results demonstrate that our mechanism is implemented with low overhead.**

Keywords: virtualization, FPGA, real-time system, micro-kernel

I. Introduction

Today, the concept of CPU-FPGA hybrid processor has become more and more popular in both academic and commercial worlds. Unlike in traditional FPGA devices where CPU cores are synthesized in the FPGA fabric as *soft processors*, the hybrid approach provides System on Chip (SoC) architectures with CPU and FPGA domains that are independently implemented.CPU-FPGA hybrid processors have several advantages. On one hand, the high-end general purpose processors are capable of establishing complex computing systems, and an existing software stack can be directly used without any obstacle. On the other hand, the implementation of FPGA accelerators offers a compelling improvement in performance when performing intensive computations. Moreover, with the help of CPU processing, FPGA accelerators can be managed much more efficiently with more complex strategies, which inevitably optimizes the acceleration.

This emerging convergence point of conventional CPU and FPGA computing makes it possible to extend traditional CPU virtualization technologies into the FPGA domain to fully exploit the mainstream FPGA computing. Our research focuses on the virtualization technology, which has gained a lot of interests and achieved enormous progress in the embedded computing domain. It has been proven that it can provide users with increased energy efficiency, shortened development cycles and enhanced security [1][2]. In parallel, the dynamic partial reconfiguration (DPR) technology on FPGA has been playing an important role in high performance adaptive computing [3]. Therefore, the combination of both DPR and virtualization technologies may be an interesting idea to significantly accelerate the computationally intensive applications in embedded systems.

While being considered as quite promising, the exploitation of DPR-enhanced virtualization also brings up new challenges. In virtualization, guest OSs are executing in strongly-isolated environments called virtual machines (VM). Each VM has its own software tasks and virtualized resources, which are isolated from the physical resources. In this context, the usage of hardware accelerators from VMs will be dynamic and independent. Thus, it is preferred that DPR accelerators are shared by multiple VMs by means of abstract and transparent layer so that the isolation of virtual machines will not be undermined. The actual allocation and management should be performed via an additional hypervisor mechanism, and should remain hidden from the guest OSs.

In this paper, we present an approach to exploit the potential of CPU-FPGA systems by proposing a framework extending virtualization with DPR accelerators. We will describe and evaluate a virtual embedded system in a hybrid ARM-FPGA platform Xilinx Zynq-7000. The remainder of the paper is organized as follows: Section II presents the related works. Section III presents the mechanisms of the DPR management in a virtualized environment. In Section IV, we verify and examine the proposed system with practical hardware/software applications and analyze the results.

II. Related Works

The hybrid CPU-FPGA architecture generally features CPUs that are dedicated to the embedded systems domain. For example, Xilinx released the Zynq-7000 series which offers an ARM-FPGA SoC. ARM processors have also been

introduced in Cyclone-V and Arria-V Altera families. Intel propose its Atom processor E600C Series, which pairs an Intel Atom processor SoC with an Altera FPGA in the same package. Recently, Intel has taken a further step by releasing a Xeon/FPGA platform dedicated for data centers.

In the academic domain, the topic of embedded CPU-FPGA based systems has also been massively studied. Numerous works have consisted in providing current reconfigurable FPGA devices with OS support ([4][5][6] [7]). One successful approach in this domain is ReconOS [8], which is based on an open-source RTOS (eCos) that supports multithreaded hardware/software tasks. ReconOS provides a classical solution for managing hardware accelerators in a hybrid system in standard thread model. However, the possibility of virtualization was not fully discussed in these works.

In [9], this concept is implemented by providing the OS4RS framework in Linux. The virtual hardware permits the same hardware devices and the same logic resources to be simultaneously shared between different software applications. However, this approach is proposed in the context of a single OS, without considering any virtualization features. Another study is given in [10], one of the earliest researches in this domain. The authors try to extend the Xen hypervisor to support the FPGA accelerator sharing among virtual machines. However, this research focuses on an efficient CPU/FPGA data transfer method, with a relatively simple FPGA scheduler that provides a FCFS (*first-come, first served*) sharing of the accelerator, without including the DPR technology.

DPR virtualization is much more popular on cloud servers and data centers, which generally have a higher demand for computing performance and flexibility. For example, in [11], the authors use partial reconfiguration to split a single FPGA into several reconfigurable regions, each of which is managed as a single Virtualized FPGA Resource (VFR). Based on the similar principle, the work of RC3E [12] provides several vFPGA models, which permits users to require for DPR resources as full FPGA, virtual FPGA or background accelerators. However, DPR virtualization on these platforms are inappropriate for embedded systems, in which available resources are really limited compared to those available in servers or data centers.

Another interesting research [13] consists in proposing a framework dedicated to hardware task virtualization on a hybrid ARM-FPGA platform. In this work, the authors modified the CODEZERO hypervisor to manage reconfigurable accelerators. However the classical DPR technology is not employed in this work for hardware reconfiguration. Instead, reconfigurable computing components are quite simple and are more appropriate to systems with light but frequently-switched computations.

In this paper, we proposed an original approach of DPR virtualization on an embedded hypervisor named Ker-ONE, an updated version of a custom micro-kernel [14]. Efforts have been made to provide efficient DPR resource sharing among virtual machines, while meeting the applications constraints.

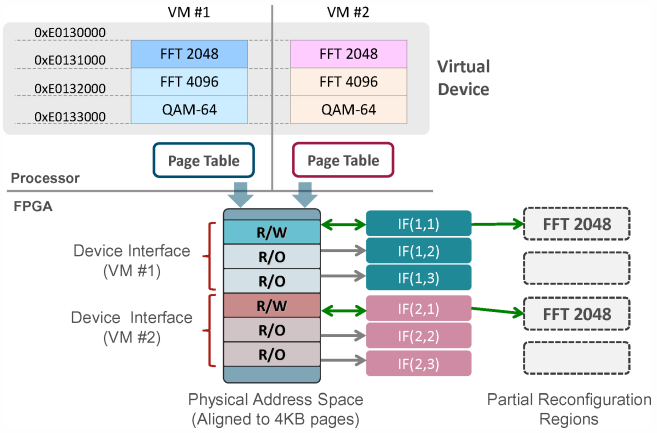

Fig. 1. Allocation of virtual devices for virtual machines via manipulating the mapping of IFs.

III. DPR Management Framework

Ker-ONE is a custom small-sized hypervisor which provides para-virtualization on ARM processors. It allows multiple guest OSes to be hosted concurrently with different priority levels. Each guest OS is running in an isolated domain named virtual machine, and is managed by an underlying virtual machine monitor (VMM). The VMM provides fundamental functions such as VM scheduling and inter-process communication (IPC). In The proposed framework, Ker-ONE is extended with dedicated management mechanism for partially reconfigurable (PR) accelerators.

Ker-ONE is designed to host several OSs of different priorities. We assume that all critical tasks are hosted in a high-priority VM, with higher timing and performance requirements. Non-critical tasks are running on low-priority VMs, for which long-latency and resource blocking is tolerable. To keep the behavior of critical tasks predictable, we also assume that the FPGA resources are always sufficient for the high-priority VM, whereas they can also be shared and reused by low-priority VMs. This assumption seems reasonable in practice, since critical tasks are pre-determined in most embedded systems.

A. Accelerator Mapping

In our system, reconfigurable accelerators are hosted in pre-determined partial reconfiguration regions (PRR), which are serving as containers. We denote these accelerators as HW tasks. A HW task can be implemented in a PRR by downloading the corresponding bitstream into the targeted PRR area via the PCAP interface [15]. HW tasks are presented as a virtual devices (VD) in the VM domain, and is completely isolated from the implementation details. Therefore, one specific virtual device can be hosted in different PRRs.

Fig. 1 describes how virtual devices are mapped to the fixed addresses in all guest OSs and considered as ordinary devices. A unified interface, i.e. a standard structure of registers, is provided to users. Like other peripherals in ARM systems, OSs access these devices by reading/writing at the address of the corresponding device interface. Note that the physical

positions of these virtual devices are not known since they can be implemented in different PRRs.

We introduce an intermediate structure, called PR interface (IF), on the FPGA side, which can be seen as an intermediate layer between the logical virtual devices and the actual accelerators. These IFs are in charge of connecting the virtual machines with accelerators so that software can control their behavior. Each IF is exclusively associated to a specific VM. Thus, mapping of reconfigurable accelerators is performed in two steps: first, the IF is mapped to the VM address space as a virtual device interface. Second, on the FPGA side, the IF is connected to the target PRR that implements the corresponding device function.

As shown in Fig. 1, IFs are initiated on the FPGA side and are mapped to the physical memory space of the processor. Their physical addresses are configured to be aligned to independent 4KB memory pages. VMs access IFs via independent page tables, which maps IFs as memory pages in virtual address space. Therefore, though a virtual device is mapped to the same virtual address for all VMs, it is implemented by using separated IFs in the FPGA. The mapping between a particular IF and the VM space of a virtual device is fixed. An IF has two identifiers: *vm_id* and *dev_id* (i.e. referred to as IF(*vm_id*, *dev_id*)) to identify the virtual machine and the virtual device to which it is associated.

An IF has two states, *connected* to a certain PRR or *unconnected*. When an IF is *connected*, the corresponding virtual device is implemented in the PRR and it is ready to be used. Being in the *unconnected* state means that the target accelerator is unavailable. We leverage the ARM paging mechanism to control VM's access to IFs. When a IF is *connected*, its page is marked as read/write so that virtual machine can control this accelerator by manipulating IF registers. On the other hand, for unavailable devices (with an *unconnected* IF), the IF pages are set as read-only, and a VM cannot configure or command this virtual device by writing to its interface. This mechanism guarantees the monopoly use of accelerators.

An example is presented in Fig. 1. In VM #1, an application is free to program and command Dev #1 (FFT-2048) as the IF associated with it is currently connected to PRR #1, where the algorithm is implemented. Meanwhile, VM #1 cannot give orders to Dev #2 and #3 since these interfaces are currently read-only. Any writing on these interfaces will cause a page-fault exception to the VMM.

One major characteristic of virtualization is that virtual machines are totally independent from each other. In our case, however, VMs are obliged to share DPR resources. This can unfortunately lead to resource sharing issues that are well known in computing systems. In traditional OS, such problem can be solved by applying synchronization mechanisms such as semaphores or spin-locks. For Ker-ONE however, such mechanisms are not suitable since they may undermine the independence of VMs. Therefore, our system introduces additional management mechanisms to dynamically handle the virtual machines' request for DPR resources. Note that such requests may occur randomly and are unpredictable.

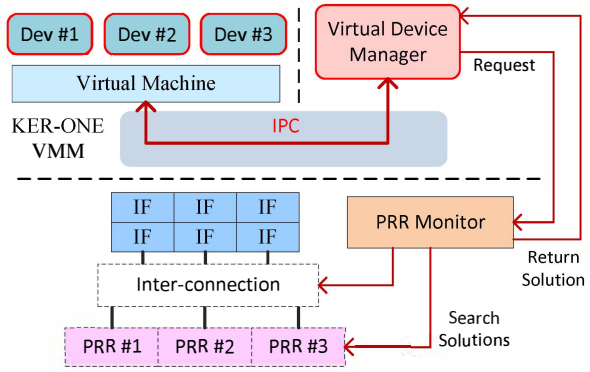

Fig. 2. Overview of the DPR management framework in Ker-ONE.

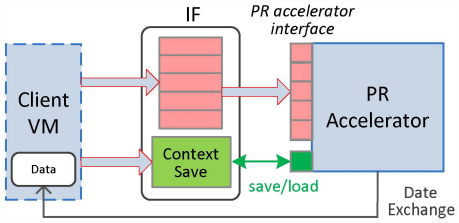

Fig. 3. Model of reconfigurable accelerator.

In Fig. 2, the proposed management mechanism is described. A *Virtual Device Manager* and a *PRR Monitor* component are made available in both software and hardware parts of the FPGA. The *Virtual Device Manager* is a particular software service implemented in an independent VM domain, which aims at detecting and handling the requests coming from VMs that want to use the virtual devices. This is performed through an IPC mechanism. In the static part of the FPGA, a *PRR Monitor* is created and is in charge of maintaining the connections between IFs and PRRs. This monitor runs in cooperation with *Virtual Device Manager*, to dynamically monitor reconfigurable accelerators and search proper solutions for VMs' requests.

B. Hardware Task Model

HW tasks see PRRs as containers, which provide FPGA resources to implement their algorithms. In this case, a given PRR may not be compatible with some virtual device, if its area (i.e. resource amount) is insufficient to implement the corresponding HW task logic. Therefore, the compatibility information of HW tasks must be foreseen beforehand.

An *HW Task Index* table is created to provide a quick look-up search for HW tasks. In this table the compatible virtual devices for each PRR are listed. For each compatible virtual device, a *HW Task Descriptor* structure is given, which stores the information of the corresponding bitstream, including its ID, memory address and size. This information is used to correctly launch PCAP transfers and perform reconfiguration.

Fig. 3 depicts the model of HW task and its interaction with VM client. As shown, Virtual machines access HW tasks via IFs. We proposed a standard interface to facilitate the multiplexing of DPR resources, denoted as *PR accelerator interface*. It is implemented in both IFs and HW tasks, and conveys the register values from the IF to HW tasks. Once the

TABLE I
LIST AND DESCRIPTION OF PORTS IN *PR accelerator interface*

Register	Width	Description
STAT	32-bit	HW task status register
START	8-bit	Start flag
OVER	8-bit	Computation Over flag
CMD	32-bit	Command register
DATA_ADDR	32-bit	Data buffer address register
DATA_SIZE	32-bit	Data buffer size register
RESULT	64-bit	Computation result register
INT_CTRL	32-bit	Interrupt controller register
Custom Ports	8*32-bit	Provide 8 IP-defined ports

IF is connected to an HW task, a virtual machine can write commands or configurations into the IF registers to control the HW task behavior.

In TABLE I, the structure of the *PR accelerator interface* is listed. Virtual machines start the process by setting the *START* flag. When the required computation is over, the *OVER* flag is set and the result is returned in the *RESULT* register. HW tasks can be programmed to perform a DMA data transfer or to generate interrupts. Note that a *PR accelerator interface* structure is implemented in IF. When an IF is disconnected from a PRR, the states of virtual device execution (e.g. results, status) are still stored in this IF. In this way, the consistency of the virtual device interface is guaranteed.

The VMs that are currently using HW tasks are denoted as *clients*. HW tasks inherit the priorities of VM *clients*. We use preemptive policy for HW tasks, meaning that requests from higher-priority VMs can preempt low-priority HW tasks.

The preemption of HW tasks requires extra attention since their computations can only be stopped when they reach some point of their execution paths where the integrity of the data frame is preserved. These points are denoted as *consistency points* where the execution path is safe to interrupt and may resume without a loss of data consistency. Designers of HW tasks have to identify the *consistency points* that allow the accelerators execution to be preempted and to save the interrupt state. This mechanism is shown in Fig. 3. In each IF, a 1KB buffer is implemented to store the accelerator intermediate execution context at the *consistency points*. This data can later be used by the VM to resume the execution. In our case, the format of saved context is openly defined by designers. However, considering the extra overhead, the size of context data should be kept as light a possible to enable fast context switch.

C. PRR State Machine

A PRR houses HW tasks to implement different devices, and behaves as a state machine. The state determines if a PRR can be allocated, and how should it be allocated. There exist five states:

- ***Idle***: The PRR is idle without any ongoing computation and is ready for allocation.
- ***Busy***: The PRR is in the middle of a computation

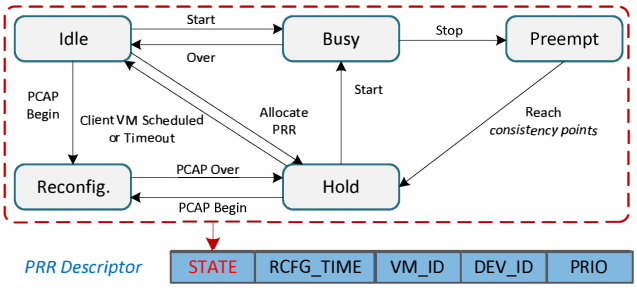

Fig. 4. The behavior of PRRs as a state machine.

- ***Preempt***: The PRR is running, but the computation will be stopped (preempted) once it reaches a consistency point.
- ***Hold***: The PRR is allocated to a VM and is preserved for a certain amount of time
- ***Reconfig***: The PRR is in the middle of a PCAP reconfiguration.

The PRRs behaviour can be described according to the flow chart given in Fig. 4. As depicted in this figure, a PRR can only be directly allocated to VMs when it is in *Idle* state and requires no reconfiguration. In other situations, the allocation process requires extra overheads caused by PCAP transfers or preemption.

We also have introduced *Hold* as an intermediate state. The PRRs that are allocated to a VM will first enter this state. This indicates that the PRR is reserved to a certain VM client. PRRs in the *Hold* state will block any re-assignment and will wait to be used by the VM. PRRs will be released and return to the *Idle* state under two conditions: the target VM is scheduled into the CPU, or the pre-set waiting time *Expire* runs out.

PRR holds the essential information in a *PRR Descriptor* data structure. This list indicates PRR state (see Fig. 4). It also includes the information of the currently-hosted HW task: the client VM ID, the virtual device ID (i.e. accelerator ID) and the HW task priority, which are used to make allocation decisions. Note that, in our context, the bitstreams size is strictly predefined by the size of the reconfigurable area. Therefore, the reconfiguration time of each PRR can be predicted. This factor is also included in the *PRR Descriptor*.

D. PR Resource Requests and Solutions

Every time that a VM tries to use an unavailable virtual device, a page-fault exception will be triggered and then handled by *Virtual Device Manager* as a PR resource request: *Request (vm_id, dev_id, prio)*, which is composed of the VM ID, the virtual device ID and a request priority. The request priority is equal to the priority of the calling VM.

The *PPR Monitor* on the FPGA side is responsible to search for appropriate allocation plans for such requests. This plan is referred as a *solution*. A complete *solution* is formatted as:

$$Solution\{vm, dev, Method(prr\_id), Reconfig\}, \quad (1)$$

which includes the target VM, the required device, the actual allocation method and reconfiguration flag. The different methods include:

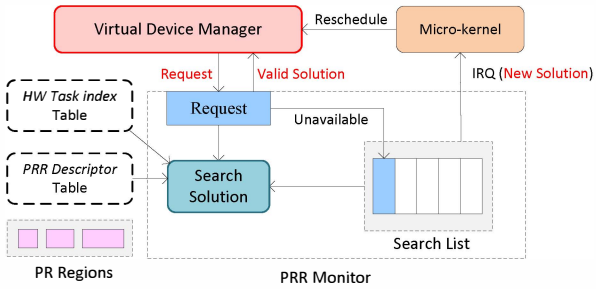

Fig. 5. Solution searching mechanism in the *PPR monitor*.

- ***Assign****(prr_id)*: this solution directly allocates the returned PRR (i.e. *prr_id*) to the request VM. If the requested device *dev_id* is not implemented in this PRR, a *Reconfig* flag will also be added.
- ***Preempt****(prr_id)*: no PRR can be directly allocated, but the returned PRR (i.e. *prr_id*) can be preempted and reallocated. If the requested device *dev_id* is not implemented in this PRR, a *Reconfig* flag will also be added.
- ***Unavailable***: currently no PRR is available for *Request(vm_id, dev_id, prio)*.

The *PPR Monitor* searches for the best *solution* by checking the *PRR Descriptors* (see Fig. 4). For a given *Request (vm_id, dev_id, prio)*, the *PRR Monitor* first obtains the list of compatible PRRs for the target device (*dev_id*) by checking the HW task index table. The states of these compatible PRRs are then checked for possible solutions. Then it searches the states of PRRs for solutions. If multiple solutions are found, the best one is chosen according to the selecting policy. In our algorithm, *Idle* PRRs are considered to be best solutions. Preemptions are chosen only when no *Idle* PRR exists. Besides, the selector always chooses the solution with a minimal PRR size since it causes the minimal reconfiguration overhead and power consumption. However, these policies can be easily modified and adapted.

Fig. 5 depicts the interaction between the *PPR Monitor* and the *Virtual Device Manager*. Normally the selected solution is sent to the *Virtual Device Manager* for further handling. However, if there is no valid solution (i.e. *Unavailable*), this unsolved request will be added to the *Search List*, which is a waiting list buffer of all unsolved requests. *PPR Monitor* keeps searching solutions for requests in this list, and acknowledges the *Virtual Device Manager* whenever a new solution is found. The searching runs in parallel with VMs, following priority-based FIFO principle, so that when a requests conflict occurs, the *PPR Monitor* always chooses the highest priority request.

E. Virtual Device Manager

The *Virtual Device Manager* is a special service provided by Ker-ONE, running in an independent VM a with the highest priority. Running in isolated VM improves the security of its functioning. This service stores all the HW task bitstreams in its memory and is the only component that can launch PCAP reconfigurations. The main task of this manager is: (1) to communicate with VMs and manage the virtual devices in their space; (2) to correctly allocate PR resources to VMs.

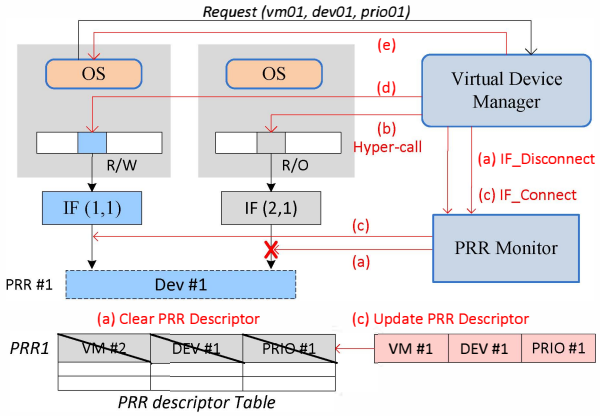

Fig. 6. Execution flow for solution to directly allocate an accelerator.

We already know that any writing operations on an unavailable virtual device interface will trap to a VMM as a page-fault exception. We assume that, in order to command a virtual device, a VM always need to configure the device interface in the IF. In this case, any VMs' attempt to use unavailable virtual devices will be automatically detected by the VMM, and then passed to the *Virtual Device Manager*.

Since the virtual devices are pre-determined and identically mapped in all VM virtual address spaces (see Fig. 1), it is easy to identify the target device by simply checking the page-fault address. Then this exception is translated into the *Request (vm_id, dev_id, prio)* format to search solutions. The *Virtual Device Manager* allocates DPR resource to VMs according to different solutions.

The *allocation/ de-allocation* of DPR resources consist in manipulating IF connections and VM page tables. In Fig. 6, we depict the complete flow to allocate an accelerator to VM as a virtual device. In this example, after a given $Request(vm01, dev01, prio01)\}$, a solution {*Assign (prr01), non-Reconfig*} is performed. We assume that PRR #1 was previously used by VM #2 and that it is currently in the *Idle* state. It can then be directly re-allocated in following steps:

(a) Command *IF_Disconnect* is given to the *PRR Monitor* to disconnect the IF of VM #2. Meanwhile, the *PRR descriptor* entry of PRR#1 is cleared. (b) Set the no-more-available device interface as read-only in VM #2's page table (via hyper-call). (c) Use *IF_Connect* to connect PRR to the IF of VM #1. *PRR Monitor* also updates the *PRR descriptor* entry with the new client VM #1. (d) Change VM #1's dev01 interface as read-write (via hyper-call). (e) VMM suspends *Virtual Device Manager* and resumes VM #1 to the exception point and VM #1 continues to use this device.

For guest OSs, the ideal solution is {*Assign, non-Reconfig*}, because a PRR can be allocated immediately as shown in the previous example, and the allocation is totally transparent. On other solutions which require reconfiguration or preemption, the target virtual device need to wait for several additional steps before it is ready to use. In these cases, the *Virtual Device Manager* informs the requesting VM with IPC messages, and suspends itself to wait for the completion of reconfiguration

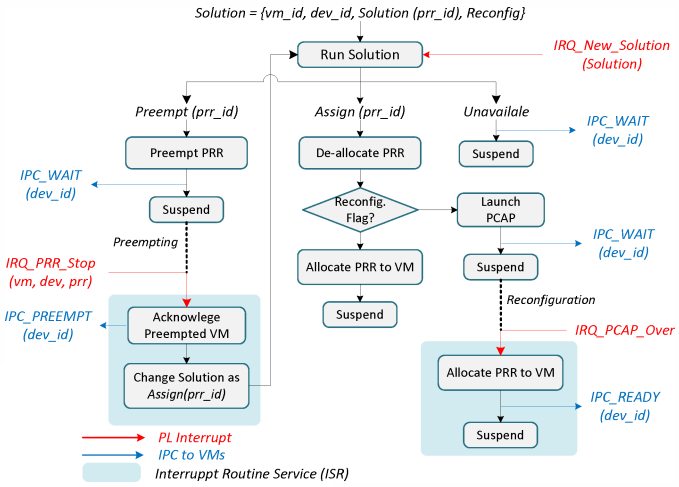

Fig. 7. The process of Virtual Device Manager handling *Solutions*.

TABLE II
COMMUNICATION SIGNALS FOR DPR ALLOCATION

Signal	Type	Recv.	Message
IPC_WAIT	IPC	VM	Device currently unavailable
IPC_READY	IPC	VM	Device is ready
IPC_PREEMPT	IPC	VM	Device is preempted
IRQ_New_Solution	IRQ	VDM	New solution is found
IRQ_PCAP_Over	IRQ	VDM	Reconfiguration over
IRQ_PRR_Stop	IRQ	VDM	Preemption is complete

or preemption before making further operations. Meanwhile, the *PRR Monitor* keeps tracking the procedure of unfinished solutions on the FPGA side, and sends interrupts to the *Virtual Device Manager* whenever further operations are needed.

This mechanism is explained in details in Fig. 7, which demonstrates the role of the *Virtual Device Manager*. The routine is composed of one main function *Run_Solution()* and several interrupt service routine (ISR). We can notice that preemption and reconfiguration solutions are performed in two stages: (1) the manager launches the reconfiguration or preemption and then goes to sleep, (2) the manager is awakened to complete the unfinished solution in ISR. Note that for *Preempt* solution, the manager first stops the preempted accelerator, and then handles it as *Assign* solutions.

In Fig. 7, different signals are used to facilitate the allocation process, helping the synchronization among components. Some communication signals are destined to the requesting VMs, and indicate the state of the required device. Others are sent from the *PRR Monitor* to the *Virtual Device Manager*, to acknowledge the events for unfinished solutions. These signals are listed in TABLE II.

F. User Policy

With our framework, coding is significantly simplified for software applications to use virtual devices. From their point of view, the use of virtual devices is performed by a series of write/read operations on the interface registers as ordinary devices. In an ideal situation, the interrupted virtual device would immediately be allocated and the task would continue seamlessly from the interrupted point. In other cases, specific user policies are required for the VMs. The virtual device may currently be not ready. In this case the usage of virtual device should be suspended until it is allocated. This can be implemented by blocking/unblocking the involved tasks.

Another situation is the preemption of virtual devices so that a VM no longer controls the accelerator. In this case, it is the user's responsibility to re-launch the interrupted accelerator. For example, user can create a dedicated task to detect the preemption (via *IPC_PREEMPT*) of accelerators and to restart them. Note that, for high-priority VM, such situation will not occur according to our assumption that there always exist sufficient FPGA resources for high-priority tasks. Therefore, in the domain of high-priority VM, the problem of resource blocking can be ignored.

IV. PERFORMANCE EVALUATION

A. Experimental Description

Our experiments were performed on the Xilinx ZedBoard, which provides a dual-core ARM Cortex-A9 processor and a partially reconfigurable FPGA fabric. The operating frequency of the CPU and FPGA logic are 667 MHz and 100 MHz respectively.

An experiment is shown in Fig. 9. The FPGA fabric is initially implemented with three PRRs of different sizes. Four accelerators, i.e. *QAM16, QAM64, FFT512, FFT1024,* have been synthesized into bitstream files. During the initialization stage of Ker-ONE, these files have been loaded into the RAM memory and are only accessible to the *Virtual Device Manager*. This experiment is taken from an OFDM receiver that is intended to be very flexible by considering several configurations of modulators and mappers according to the channel condition. QAM blocks aims to take a complete frame of incoming bits into account and generate 16-bit width I and Q symbols. FFT blocks work on the outcoming QAM I and Q symbols to perform demodulation. To simplify the experiment, we assume that FFT always works in sequence of QAM-16 algorithm. The data frame is set to be 18,800 bits, according to actual OFDM requirements. Therefore, the incoming frame sizes were 18,800 bits for QAM16/QAM64 and 4700 16-bit width symbols (as the outcome of QAM16) for FFT512/FFT1024, respectively. In each PRR, a DMA-supported data buffer keeps transferring frames from VM memory space to the accelerators.

Regarding the guest OSs running in virtual machines, we have modified the μC/OS-II RTOS to execute it on top of Ker-ONE. In our experiment, two μC/OS-II guests are hosted with different priority levels. For each guest OS, four available virtual devices are instantiated. Two and three tasks run respectively in both guest OSs to periodically command virtual devices to process data frames containing 18,800 bits, which causes requests for allocations during the experiment. Accelerators are then allocated at run-time. The experiment executes for several hours continuously. A custom monitor is built to measure and record the various costs of allocation mechanisms. Based on the collection of numerous samples, the overall cost for our allocation mechanism can be then estimated.

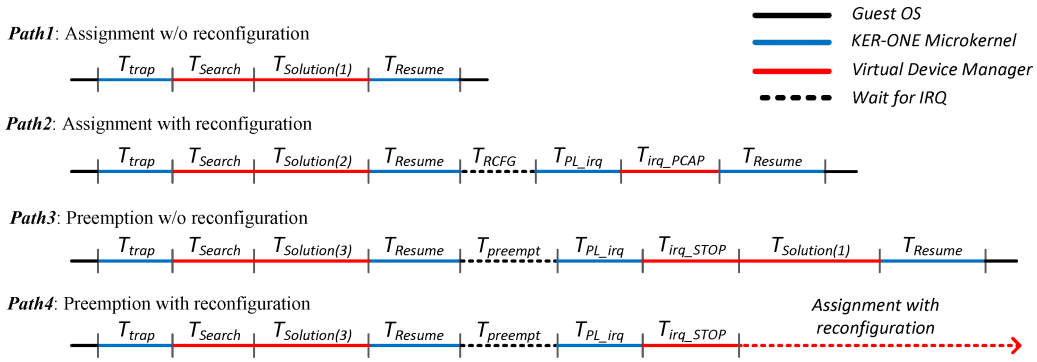

Fig. 8. Execution paths of DPR resource allocation

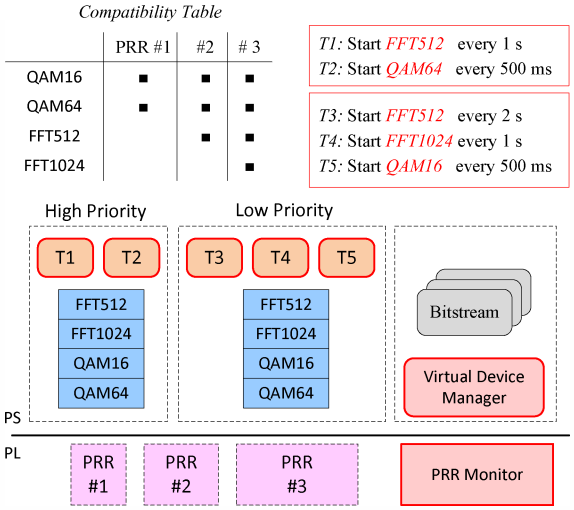

Fig. 9. Experimental architecture for performance evaluation.

B. Overhead Analysis

Our evaluation focuses on the allocation latency, i.e. the delay that occurs before the accelerator is properly allocated and ready to start. This latency is considered as the response time of DPR accelerators, and plays an important role for the OS tasks timing.

Allocation latency comes from two main sources: the allocation mechanism itself and the Ker-ONE micro-kernel. Additional overheads will be caused if the allocated accelerator requires reconfiguration. Besides, the virtualization mechanism takes up extra time. For example, the page-table faults handling, IPCs and VM scheduling will noticeably contribute to the total allocation latency. The models of execution paths in different solutions can be calculated according to the diagrams displayed in Fig.8. In these models, the allocations consist of four different solution paths that can be decomposed into smaller atomic execution overheads:

- T_{trap}: Time required by Ker-ONE to detect a page-table exception in VM domain and to invoke the *Virtual Device Manager*.
- T_{resume}: Time required by Ker-ONE to schedule back to a VM.
- T_{PL_irq}: Time required by Ker-ONE to receive IRQs from the *PRR Monitor* and to redirect them to the *Virtual Device Manager*.
- T_{Search}: Time required by the *Virtual Device Manager* to receive the VM requests and to search for solutions.
- $T_{Solution(1)(2)(3)}$: Execution time to handle different solutions: (1) direct assignment, (2) assignment with reconfiguration, (3) preemption.
- T_{irq_pcap}, T_{irq_stop}: Time required by the *Virtual Device Manager* to handle the following IRQs (i.e. *IRQ_PCAP_Over*, *IRQ_PRR_Stop*).
- $T_{preempt}$: Overhead of waiting for preemption.

TABLE III
MEASUREMENTS OF OVERHEADS DURING DPR ALLOCATION

Micro-kernel		Virtual Device Manager	
Operation	**Overheads (μs)**	**Operation**	**Overheads (μs)**
T_{trap}	0,76	T_{Search}	0.50
T_{resume}	0,64	$T_{Solution(1)}$	1.13
T_{PL_irq}	0.81	$T_{Solution(2)}$	2.77
		$T_{Solution(3)}$	0.34
		T_{irq_pcap}	0.64
		T_{irq_stop}	0.28

The results of our measurements are listed in TABLE III. Note that since Ker-ONE provides an efficient virtualization mechanism, virtual machine scheduling and virtual interrupt emulation are performed with an overhead less than $1\mu s$. The heaviest overhead is caused by $T_{Solution(2)}$, referring to the PRR assignment with reconfiguration. This is because this process includes the launch of a PCAP transfer which is composed of complex operations to set up the DMA transfer.

According to the experiment measurements, allocation overheads can be estimated as:

$$\begin{aligned} T_{Path1} &= 3.03\mu s, \\ T_{Path2} &= 6.76\mu s + T_{RFCG}, \\ T_{Path3} &= 5.10\mu s + T_{preempt}, \\ T_{Path4} &= 9.96\mu s + T_{preempt} + T_{RFCG}. \end{aligned} \tag{2}$$

Note that the estimated overheads in Eq. 2 demonstrates the allocation overheads for the high priority OS, since high priority request can always get a valid solution in our system. It can be clearly noticed that direct allocations can be efficiently performed with 3 μs latency, whereas other solutions suffer from extra overheads of preemption or reconfiguration. The

TABLE IV
RECONFIGURATION AND PREEMPTION DELAYS

Virtual Device	$T_{preempt}(\mu s)$ (WCET)	$T_{RCFG}(\mu s)$		
		PRR#1	PRR#2	PRR#3
QAM16	47.0	231	810	1,206
QAM64	31.0	231	810	1,206
FFT512	24.1	-	810	1,206
FFT1024	33.6	-	-	1,206

TABLE V
COMPARISONS BETWEEN SW AND HW IMPLEMENTATION

Algorithm	$T_{HW}(\mu s)$ (per frame)	$T_{SW}(\mu s)$ (per frame)	FPGA Resource Usage
QAM-16	47.0	1,513	2%
QAM-64	31.0	1,174	2%
FFT-512	71.1	6,582	8%
FFT-1024	90.6	12,784	13%

worst-case solution corresponds to the overhead T_{Path4}, which may result in hundreds of microseconds.

The costs of preemption ($T_{preempt}$) and reconfiguration (T_{RCFG}) are mostly depending on the implementation and application of accelerators.In TABLE IV we provide the value of these costs. T_{RCFG} is determined by the size of the bitstream, and therefore corresponds to three PRR areas. The worst-case preemption time $T_{preempt}$ is determined by the computation granularity. In our case, in order to respect the integrity of the OFDM process, QAM and FFT modules are set to be preemptive only when their determined data frame is completely processed. Note that, since the implementation of the PRRs and accelerators are fixed beforehand, these costs can then be predicted and considered for guest OS tasks schedulability.

In TABLE IV, for the accelerators used in our experiment, $T_{preempt}$ is significantly lower than T_{RCFG}. Actually, this is the case in most DPR accelerator implementations. Therefore, a preemptive allocation of DPR resources can effectively reduce the heavy reconfiguration overheads, which, however, will also considerably undermine the schedulability of low priority OS tasks. In a system where preemptions occur frequently, low priority virtual machine may never use DPR resources. The trade-off between preemption and consistency has been considered when we chose the solution selecting policy in the *PRR Monitor*. In our current searching policy, allocating *Idle* PRRs are preferred than preempting PRRs, as we want to respect the execution of tasks in low priority virtual machines. However, in a system where the higher priority OS is in charge of critical tasks, an alternative policy which encourages preemption should be applied, so that high priority tasks are guaranteed to acquire DPR resources with minimum latency.

In TABLE V, we also compare with the software computation time when processing a complete data frame. It shows that accelerator performance of heavy computation (i.e. FFT512/1024) significantly surpasses software implementations. Even though these accelerators suffer from allocation latency that may prolong the execution time, their benefit is still considerable. On the other hand, for relatively light computation, QAM, though hardware accelerators still have advantage in processing speed, this advantage gets undermined when taking T_{RCFG} into account. This is because when implemented in large PRRs, i.e. PRR#2 and PRR#3, time is wasted on reconfiguring large unused FPGA area. This results indicates that DPR technology is more suitable for large complex computation algorithms. Furthermore, the FPGA area only implements 3 PRR areas, taking around 23% of the available resources. Compared to static circuits with accelerators for both VMs, which may take up to 50% resources, the usage of FPGA is greatly reduced.

V. CONCLUSION

In this paper we have introduced an hypervisor which facilitates the DPR resource management in a virtual machine system. Our framework is based on Ker-ONE, a micro-kernel running on the ARMv7 architecture. This micro-kernel is able to host multiple OSs. In each virtual machine, DPR accelerators are mapped as universally-addressed peripherals, which can be accessed as ordinary devices. Through dedicated memory management, our kernel automatically detects the request for DPR resources and allocates them dynamically. Dedicated management components are implemented on both software and hardware sides to handle allocations at run-time. We also propose an efficient preemptive allocation mechanism that emphasizes the sharing and enhances security for virtual machine systems. In this paper we have described implementation details and presented extensive experiments to evaluate the overheads of allocation in our framework. Through evaluations and analysis, we have demonstrated that the proposed framework is capable of virtual machine DPR allocation with low overhead. As prospects, we would like to evaluate our framework more deeply by applying real-scenario implementations, e.g. complex communication systems with real-time tasks, to discuss the capability and schedulability of hosted guest OSs. We would also like to develop more sophisticated searching algorithms, so that the overall performance may be improved.

REFERENCES

[1] 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.

[2] L. Xu, Z. Wang, and W. Chen, "The study and evaluation of arm-based mobile virtualization," *International Journal of Distributed Sensor Networks*, vol. 2015, pp. 1–10, 2014.

[3] J. Becker, M. Huebner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and partial fpga exploitation," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438–452, 2007.

[4] K. Jozwik, S. Honda, M. Edahiro, H. Tomiyama, and H. Takada, "Rainbow: An operating system for software-hardware multitasking on dynamically partially reconfigurable fpgas," *International Journal of Reconfigurable Computing*, vol. 2013, p. 5, 2013.

[5] D. Göhringer, M. Hübner, E. N. Zeutebouo, and J. Becker, "Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[6] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, "Run-time services for hybrid cpu/fpga systems on chip," in *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*. IEEE, 2006, pp. 3–12.

[7] D. V. Vu, O. Sander, T. Sandmann, J. Heidelberger, S. Baehr, and J. Becker, "On-demand reconfiguration for coprocessors in mixed criticality multicore systems," in *High Performance Computing Simulation (HPCS), 2015 International Conference on*, July 2015, pp. 569–576.

[8] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "Reconos: An operating system approach for reconfigurable computing," *Micro, IEEE*, vol. 34, no. 1, pp. 60–71, 2014.

[9] C.-H. Huang and P.-A. Hsiung, "Hardware resource virtualization for dynamically partially reconfigurable systems," *Embedded Systems Letters, IEEE*, vol. 1, no. 1, pp. 19–23, 2009.

[10] W. Wang, M. Bolic, and J. Parri, "pvfpga: accessing an fpga-based hardware accelerator in a paravirtualized environment," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE, 2013, pp. 1–9.

[11] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. IEEE, 2014, pp. 109–116.

[12] O. Knodel and R. G. Spallek, "Rc3e: Provision and management of reconfigurable hardware accelerators in a cloud environment," *arXiv preprint arXiv:1508.06843*, 2015.

[13] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell, "Virtualized execution and management of hardware tasks on a hybrid arm-fpga platform," *Journal of Signal Processing Systems*, vol. 77, no. 1-2, pp. 61–76, 2014.

[14] T. Xia, J.-C. Prévotet, and F. Nouvel, "Mini-nova: A lightweight arm-based virtualization microkernel supporting dynamic partial reconfiguration," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 71–80.

[15] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual (UG585)*, 2014. [Online]. Available: http://www.xilinx.com