Ker-ONE A new hypervisor managing FPGA reconfigurable accelerators

T. Xia, Y. Tian, Jean-Christophe Prévotet, F. Nouvel

Ker-ONE: a New Hypervisor Managing FPGA Reconfigurable Accelerators

Tian Xia[a] Ye Tian[b] Jean-Christophe Prévotet[b] Fabienne Nouvel[b]

a - Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, Huawei, China

b - Univ Rennes, INSA de Rennes, CNRS, IETR, UMR-6164, F-35000 Rennes, France

Abstract

In the last decade, research on CPU-FPGA hybrid architectures has become a hot topic. One of the main challenges in this domain is to efficiently and safely manage Dynamic Partial Reconfiguration (DPR) resources. This paper focuses on the management of reconfiguration by a custom hypervisor named Ker-ONE, on an ARM-FPGA platform. Using a virtualization approach, virtual machines (VM) may access resources independently, being unaware of the existence of other VMs. Our custom hypervisor guarantees the independence and isolation of VM domains. The purpose of our work is to provide an abstract and transparent interface for virtual machines to access reconfigurable resources, while meeting real-time constraints. This means that software engineers do not need to focus on implementation details. In this paper, we present a complete architecture in which hardware accelerators are seen as virtual devices which are universally mapped in each VM space as ordinary peripherals. The hypervisor automatically detects VMs' requests for DPR resources and handles them dynamically according to a preemptive allocation mechanism. We also evaluate the efficiency of our framework by measuring the critical overhead during DPR management and allocations. The results demonstrate that our mechanisms are implemented with low overhead compared to other approaches and that they are compatible with real-time scheduling.

Keywords: Embedded Systems, FPGA, Partial Reconfiguration, Virtualization, Hypervisor

March 1, 2019

1. Introduction

Today, the concept of CPU-FPGA hybrid processors has become more and more popular in both academic and commercial worlds. Unlike in traditional FPGA devices in which 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. First, general purpose processors are able to implement complex and flexible computing systems, with a huge variety of applications. Second, FPGA accelerators offer a constant improvement in performance of intensive computations and act as a powerful support for processors. Additionally, the dynamic partial reconfiguration (DPR) technology on FPGA has been playing an important role in high performance adaptive computing [1].

Meanwhile, in the embedded computing domain, virtualization has gained a lot of interest and achieved enormous progress. This technique allows to separate tasks into isolated domains without extra porting efforts. It has been proven that it can provide users with increased energy efficiency, shortened development cycles and enhanced security [2] [3]. Therefore, we made the assumption that the combination of both DPR and virtualization is an interesting idea to significantly accelerate applications and guarantee flexibility.

While considered as quite promising, the exploitation of DPR-enhanced virtualization also brings up new challenges. In virtualization, guest OSs usually run in strongly-isolated environments called virtual machines (VM). Each VM has its own software tasks and virtual resources which abstract physical resources. In this context, the use of hardware accelerators by VMs must be dynamic and independent. Note that these accelerators could be shared by multiple VMs. This means that an abstract and transparent layer has to be provided so that the isolation of virtual machines will not be undermined.

Ideally, the actual allocation and management should be performed by an hypervisor, and should remain hidden from guest OSs. Furthermore, in addition

to the complex problem of real-time scheduling that is often met in embedded systems, the sharing of FPGA resources among multiple virtual machines may significantly increase the management complexity. This constitutes a real challenge for designers to guarantee real-time capability.

In this paper, we address these challenges by proposing a framework extending virtualization with DPR management. This framework features a new resource mapping and management mechanisms to provide transparent virtual FPGA resources to the VMs. Ideally, the FPGA accelerators are designed to fit in the reconfigurable area and are implemented with dedicated preemption mechanisms and context save/restore methods. Then, virtual machine tasks may be programmed to access these accelerators as native devices. No further details are required for the user to deploy tasks in virtual machines.

The major contributions of this paper are listed as follows:

- We propose a lightweight micro-kernel that is compatible with real-time constraints.
- We describe a new approach for FPGA resource virtualization which maps resources as native accelerators in the VM domains to ease the programming of user tasks.
- We present an original FPGA management framework and a new hardware accelerators model which enable preemptive scheduling of FPGA resources among multiple VMs.
- We run extensive experiments on an ARM-FPGA platform [4] to evaluate the performance of our proposed approaches. Analysis and proof of the real-time capability of our framework are also provided.

The remainder of the paper is organized as follows: section 2 presents the related works. Section 3 describes the architecture of the proposed hypervisor. In Section 4 the mechanisms of the DPR management in a virtual environment are presented. Section 5 shows the performance of the micro-kernel and performs some comparisons with existing architectures. In Section 6, we demonstrate the

feasibility of the proposed system with practical hardware/software applications and analyze the results.

2. Related Works

An hybrid CPU-FPGA architecture generally features CPUs that are dedicated to the embedded system domain. For example, Xilinx released the Zynq-7000 series which features an ARM-FPGA SoC. ARM processors have also been introduced in Cyclone-V and Arria-V Altera families. Intel has proposed its Atom processor E600C Series, which consists of an Intel Atom processor SoC and an FPGA within the same chip. Recently, Intel has taken a further step by releasing a Xeon/FPGA platform dedicated to data centers.

In the academic domain, embedded CPU-FPGA based systems have also been massively studied. Numerous works have tried to provide current reconfigurable FPGA devices with OS support ([5],[6][7],[8]). One successful approach in this domain is ReconOS [9], 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. However, virtualization is not fully discussed in these works.

In [10], reconfiguration management is implemented by providing the OS4RS framework in Linux. Virtual hardware allows the same devices and the same logic resources to be simultaneously shared between different applications. However, this approach is proposed for a single OS only, without considering virtualization. Another study is described in [11]. This was one of the earliest researches in this domain. The authors tries to extend the Xen hypervisor to support FPGA accelerator sharing among several virtual machines. However, this research proposes 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 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 [12], authors use partial reconfiguration to split a single FPGA into several reconfigurable regions that are managed as a single Virtual FPGA Resource (VFR). Based on the same principle, the work of RC3E [13] provides several vFPGA models, allowing users to access 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 drastically limited compared to those available in servers or data centers.

Another interesting research [14] proposed 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. In this work, the classical DPR technology is not exploited for hardware reconfiguration. Instead, reconfigurable computing components are quite simple and seem more appropriate to systems with light but frequently-switched computations.

In this paper, we propose an original approach for DPR virtualization on an embedded hypervisor named Ker-ONE, an updated version of a custom microkernel [15]. Efforts have been made to provide efficient DPR resource sharing among virtual machines, while meeting the applications' constraints.

3. The Ker-ONE Architecture

3.1. Overview

In this section, we describe the design and implementation of the Ker-ONE micro-kernel, which lays the foundation of our framework by offering the mandatory virtualization capabilities. Ker-ONE outperforms other approaches since it is very small and fast. Moreover, it provides enhanced real-time support. Currently, the design of Ker-ONE is based on few assumptions:

- In a first research step, we only have considered single-core architectures, leaving multi-core systems to future prospects.

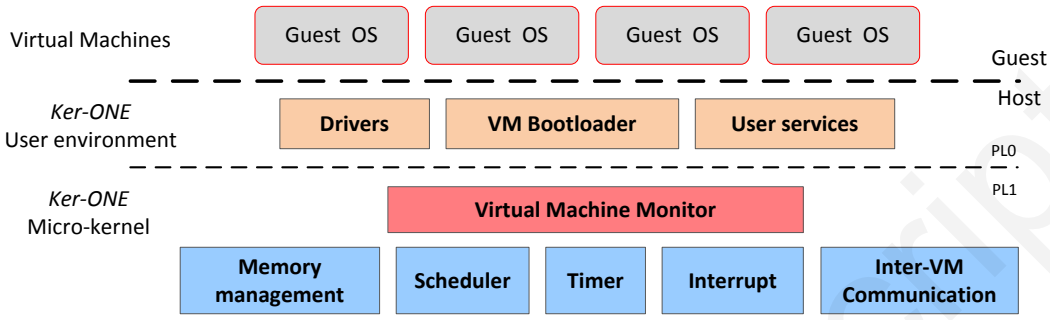

Figure 1: Ker-ONE consists of a micro-kernel and of a Virtual Machine Monitor running at a privileged level. The User environment executes in a non-privileged level

- We mainly deal with virtualization of simple guest OSs such as μC/OS or FreeRTOS, instead of complex systems such as Linux, since para-virtualizing these types of OS would be quite expensive and error prone.
- In order to provide strong protection to critical tasks, we made sure that all critical real-time tasks execute in one specific guest real-time OS (RTOS). The less critical tasks execute in general-purpose OSs (GPOSs). Therefore, Ker-ONE is designed to co-host a single guest RTOS and one or multiple additional guest GPOSs.

The Ker-ONE framework is shown in Figure 1. It consists of a host micro-kernel and a user-level environment. Ker-ONE follows the principle of minimal authority and low complexity. The micro-kernel is the only component that runs at the highest privilege level, in the supervisor mode. Only the basic features that are security-critical have been implemented in the micro-kernel: the scheduler, memory management, the inter-VM communication, etc. All non-mandatory features have been eliminated, so that the micro-kernel's Trust Computing Base (TCB) is reduced. The Trust Computing Based corresponds to pieces of software and hardware on top of which the system security is built. Normally, a smaller TCB size corresponds to higher security since it reduces the system's attack surface. In our case the TCB is kept small, which leads to improved security.

The user environment runs in user mode and is composed of additional system services, such as device drivers, file systems, VM bootloaders, which run as server processes (see Figure 1). Note that this framework is designed to be scalable and easily-adaptable to extension mechanisms.

Multiple virtual machines (VM) run on top of the user environment and Ker-ONE is based on para-virtualization. In this technique, a guest OS is modified to explicitly make calls (i.e. hyper-calls) to the hypervisor or a virtual machine monitor in order to handle privileged operations. Each virtual machine may host a para-virtualized OS (i.e. guest OS) or a software image of a user application, which has its own independent address space and executes on a virtual piece of hardware.

The Ker-ONE framework relies on a virtual machine monitor(VMM) to support the execution of guest OSs in their associated virtual machine. It handles virtual machines hyper-calls, emulates sensitive instructions and provides virtual resources to the virtual machines.

In the following, we briefly introduce our approach to virtualize some basic system resources.

3.1.1. Memory Virtualization

Ker-ONE offers three memory privilege levels: *host*, for the VMM, *guest kernel* for guest OS kernels and *guest user* for guest OS applications. For each VM, an independent page table is created. In this table, the *host* space is configured to be only accessible to the micro-kernel and cannot be directly accessed by VMs.

We then leverage the Domain Access Control functionality in the MMU to forbid the access from pieces of software running in the *guest user* space to the *guest kernel*. This guarantees that a guest OS kernel is protected from guest user applications. We also avoid complex shadow mapping techniques since our targeted guest OSs only have single-domain page tables, e.g. uC/OS-II. According to the para-virtualization concepts, guest OSs may update page table mapping by sending hyper-calls to the VMM.

3.1.2. Interrupt Virtualization

The ARMv7 architecture features a Generic Interrupt Controller (GIC) to control interrupts. For VMs, virtual interrupts are generated by the VMM. In order to maintain the guest OS original interrupts handling routine, a virtual GIC (vGIC) has been designed with virtual registers that are similar to the physical GIC registers.

The vGIC stores the state of virtual interrupts for each VM and emulates the GIC behavior by handling virtual interrupt states. When a physical interrupt is generated, the VMM handler generates a corresponding virtual interrupt in the vGIC, which will insert a virtual interrupt to the VM and forced it to jump to its local exception vector.

Note that the states of virtual interrupts are consistent and independent in each VM. For example, a virtual interrupt can be disabled or masked by one VM, while the corresponding physical interrupt can still be collected by other VMs.

3.2. Ker-ONE Optimzation

One important issue that influences the real-time capability of an OS is the kernel critical path, i.e the kernel code that cannot response to any events or be preempted. Longer and uncertain kernel critical paths will make an OS unsuitable for hard real-time tasks. For example, for a monolithic OS kernel such as Linux, the costly and unpredictable kernel path severely undermined its real-time capability [16]. This can only be solved by patching the kernel to make it fully preemptive or to use the concept of micro-kernel [17].

In micro-kernels, this problem is naturally relieved due to the simplicity of the kernel. In our research, by applying a series of new optimization methods, we improved our micro-kernel to obtain higher performance by shortening the kernel critical paths.

3.2.1. Shared Memory Region

For a guest OS, any access to privileged resources is trapped into the VMM in order to manipulate the corresponding virtual resources. This mechanism can be quite costly if the guest OS accesses such resources frequently and causes considerable overhead due to numerous hyper-calls.

To address this problem, we chose to implement virtual resources that are frequently used in a VM/VMM shared memory region, in which guest OS can directly access the virtual resources without making hyper-calls. In other words, the VMM does not need to be aware of such state updates immediately. Instead, the VMM checks and emulates the virtual resources only when necessary and asynchronously.

One obvious advantage of this policy is that a guest OS can perform operations on these resources without generating hyper-calls, which greatly reduces overheads. Though this approach requires extra coding at both VM and VMM sides, it is still the preferred optimization since it considerably shortens the execution path to access resources.

Here we focus on the two most frequently accessed resources by a guest OS. the first consists of the PSR registers, including CPSR and SPSR, which are used for common OS operations such as context switch, synchronization and mode change. The second is the virtual GIC registers.

In an RTOS, the access to these resources may noticeably influence the IRQ handling overhead. As shown in Figure 2, we use a data structure to store the virtual contents of PSR and vGIC in a shared memory region to optimize the overall performance.

Dedicated macros are used to patch the source code of guest OSs and to replace the hyper-calls that are trying to access these registers. Guest OSs perform operations on these registers via macros, which translate them into read/write processes from/to the virtual registers that are in shared memory.

The VMM only manipulates the current state of virtual PSR and vGIC when necessary. For example, when a VM switch is performed or when a physical

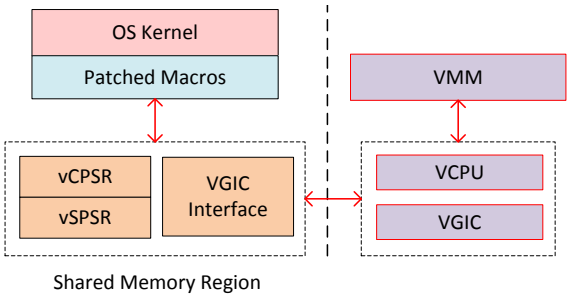

Figure 2: Implementation of the virtual PSR and vGIC interface in a VM/VMM shared memory region.

interrupt arrives, the VMM will check the shared memory region and handles the events according to the state of the virtual registers.

3.2.2. Optimized Inter-VM Communications

The issue of IVC can be interpreted as the classical inter-process communication (IPC) problem in micro-kernels. In Ker-ONE, we use simple and optimized asynchronous communication methods instead of classic synchronous IPC model to achieve lower complexity. An IRQ-based IVC mechanisms is implemented in our system. Ker-ONE leverages the VMM/VM shared memory region to facilitate asynchronous IVC. For each VM, a shared memory page is created that can be accessed from both VMM and VM sides. The sending and receiving processes of IVC mechanisms are performed with only several lines of read/write instructions on the shared memory. Therefore, this approach is shorter and lightweight compared to the simplified fast IPC model in L4 micro-kernels [18].

3.3. Real-Time Capability

Ker-ONE has been designed to host one RTOS and several GPOSs. The RTOS tasks are considered as critical with real-time constraints. We assume here that users are responsible for defining a scheduling strategy for the real-time task set with a suitable scheduler (Rate Monotonic, EDF, server-based, etc). Ker-ONE is responsible for guaranteeing real-time constraints with no or at least minimal modification of the original RTOS scheduling settings.

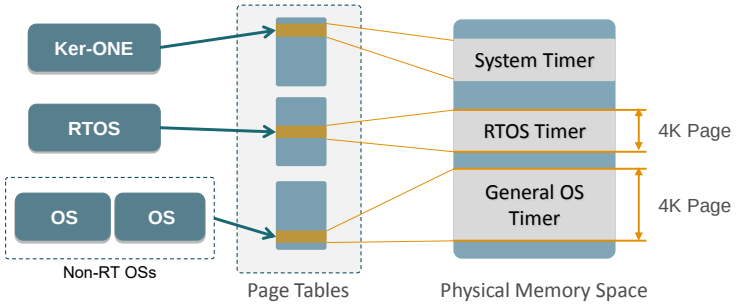

Figure 3: Management of 3 independent physical timers by the VMM, RTOS and non-RTOSs respectively. For a single guest OS, only one timer interface is mapped in the corresponding memory space.

This requires several features: the scheduling accuracy for the RTOSs, the guarantee of efficient CPU bandwidth for these RTOSs and the compliance with the RTOSs' original scheduler. These characteristics will be discussed in the following subsections.

3.3.1. Timer Virtualization

A RTOS scheduler relies on timer ticks to determine if a specific task is ready to execute. In classic virtualization, a physical timer is managed by a VMM, and VMs are provided with virtual timers that may be be accessed by traps or hyper-calls. This method is generally problematic. First, trapping into the hypervisor at each timer operation may imply high performance overhead [19]. Second, the VM timer resolution is bounded by the timer period of the hypervisor. For example, with an hypervisor period of $10ms$, a guest OS with $1ms$ timer accuracy may not work correctly. In Ker-ONE we propose a high accuracy timer virtualization approach to improve the RTOS schedulability.

First, three independent physical timers are provided: a system timer, a RTOS timer and a GPOS timer (see Figure 3). The system timer is dedicated to the host and can only be accessed by the micro-kernel. The RTOS timer is exclusively used by the RTOS VM. The GPOS timer is shared by the other VMs.

Second, we allow VMs to access and program the timer directly without

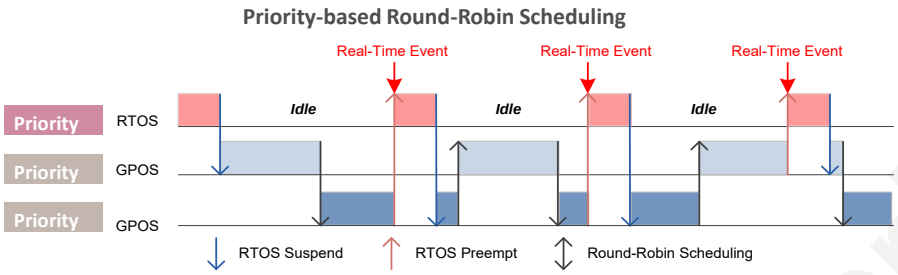

Figure 4: Priority-based Round-Robin Scheduling.

being trapped in the hypervisor. For each VM, only one timer interface is mapped in its memory space, so that it can only access the allocated timer. A guest OS is free to configure its timer, e.g. the clocking period, the interval value and interrupts.

This timer pass-through mechanism is especially advantageous for the RTOS since it fully controls a native physical timer directly. Without virtualization overhead, the performance of the RTOS scheduler is maximized.

Moreover, the GPOS timer has to be virtualized to protect the timer state of each GPOS, which includes saving and restoring the timers' registers values. Although this slightly increases the VM switch overhead, this mechanism is still preferred for GPOSs since it avoids frequent hyper-calls or traps and facilitates the VM timer emulation.

3.3.2. Real-time Scheduling

Several researches on real-time scheduling in virtualization systems have already been led. For example, VMM schedulers based on compositional real-time framework [20] and server-based scheduler [21] have been designed to be used in RT-XEN and other micro-kernels. However, they either require additional model computation [20] or require modifications of the OS original scheduling interface, which is against our intention.

In our work, we assume that users have already designed a workable schedule for a given real-time tasks set executed on a native machine. The purpose of

the VMM scheduler is to host real-time tasks according to the original scheduling settings. This strategy minimizes the additional workload on users, and simplifies the micro-kernel.

The VMM scheduler follows the concept of background scheduling, which is quite simple and reliable. Low priority tasks are only allowed to execute when high priority tasks are idle. Ideally, low priority tasks have no influence on the execution of high priority tasks, since only the idle time is donated.

In Ker-ONE, a priority-based preemptive round-robin strategy is applied (see Figure 4). GPOSs run at an identical low priority level, while the RTOS is assigned a higher priority. Within the same priority level, the CPU is shared according to a time-slice-based round-robin policy.

The RTOS can always preempt the GPOSs as long as it is ready to run. The events evoking RTOS include timer ticks pre-set by the RTOS scheduler and sporadic interrupts for RTOS. In either case, RTOS will be immediately scheduled and start running.Note that, system service threads automatically inherit the priority of the caller VM, so that system services are also preemptable and will not block the RTOS scheduling.

With the proposed scheduling policy, and the accurate pass-through timer introduced earlier, the influence on the original RTOS scheduler is minimized. In section 5, we will demonstrate that the virtualization overhead on the RTOS scheduler is negligible, and that the original scheduling settings are maintained.

4. Dynamic Partial Reconfiguration Management

In this section, the CPU-FPGA architecture is studied, where CPU and FPGA are tightly integrated. FPGA resources are connected to a CPU with dedicated interfaces and can be mapped to its unified memory space. In this context, the role of Ker-ONE is to host several simple guest OSs with different priorities.

In our architecture, we made the assumption that all critical tasks are hosted in a high-priority VM, with high performance. Non-critical tasks run in low-

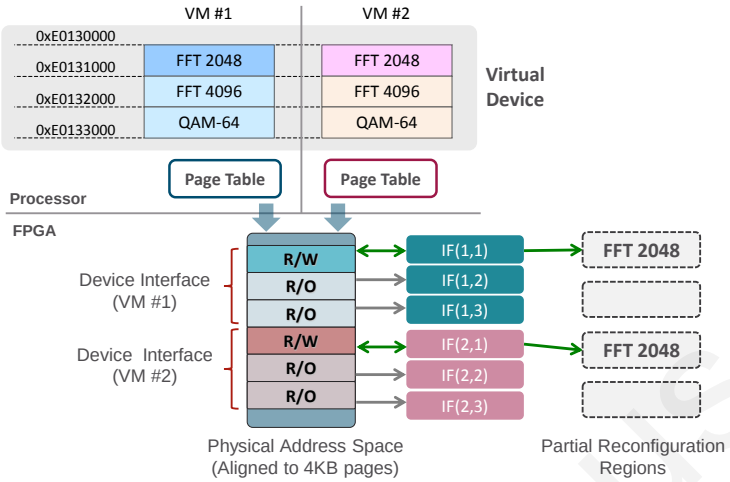

Figure 5: Allocation of virtual devices to virtual machines

priority VMs, for which long latency and resource blocking may be tolerable.

To predict the behavior of critical tasks, we also assume that the FPGA resources are always sufficient for the high-priority VM, whereas they can also be shared and re-used by low-priority VMs. This assumption seems reasonable in practice, since critical tasks are pre-determined in most embedded systems.

4.1. Accelerator Mapping

In our system, reconfigurable accelerators are hosted in different partial reconfiguration regions (PRR), which can be seen as containers. These accelerators are denoted as hardware (HW) tasks.

Each HW task is an instance of an accelerator algorithm and can be implemented in different reconfigurable regions by downloading the corresponding bitstream into the targeted area via the PCAP interface [4]. HW tasks are presented as virtual devices (VD) in the VM domain, and completely abstract the implementation details.

Figure 5 describes the way virtual devices are mapped to fixed addresses in all guest OSs. A unified interface, i.e. a standard structure of registers, is provided to users. Like any other peripherals in ARM systems, OSs access these devices by reading/writing from/to the address of the corresponding device interface.

Note that the physical positions of these virtual devices are not determined since they can be implemented in different PRRs.

An interface component (IF) has been implemented on the FPGA side. This interface can be seen as an intermediate layer between the logical virtual devices and the actual accelerators. It is in charge of connecting the virtual machines with accelerators so that software can control their behavior. Each IF is exclusively associated to a specific virtual device in a specific VM. Therefore, mapping of reconfigurable accelerators is performed in two steps.First, the IF is statically mapped to the VM address space as a virtual device interface. Second, the IF is dynamically connected to the target PRR that implements the corresponding device function. The second mapping is performed on the FPGA side.

As shown in Figure 5, IFs are initiated on the FPGA side and are assigned to physical memory addresses on the processor side. Their physical addresses are configured to be aligned to 4KB memory pages. The VMM manipulates the page table of each VM to map IFs to the VM's guest physical address space as independent device interfaces. IFs that offer the same virtual device function are mapped to identical address spaces in different VMs. For example, in Figure 5, though the QAM accelerator is mapped to the same virtual address for all VMs, it has different IFs in the FPGA.

As the mapping between a particular IF and the VM space of a virtual device is fixed, an IF can be identified with two identifiers: *vm_id* and *dev_id* (i.e. referred to as IF(*vm_id*, *dev_id*)), which indicates the VM and the accelerator algorithm to which it is associated. An IF has two states: *connected* to a certain PRR or *unconnected*. When it is *connected*, the corresponding virtual device is implemented in the PRR and 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 the VMs access to IFs. When an IF is *connected*, its registers are mapped as read/write so that a VM can directly control the accelerator. On the other hand, for unavailable devices (with an *unconnected* IF), the registers are set as read-only and whenever a VM configures or commands a virtual device by writing to its IF, a VM exit

is triggered. This mechanism guarantees the unique use of accelerators, and automatically detects any VM's request on unavailable FPGA resources.

See the example in Figure 5. In VM #1, an application is free to program and command Dev #1 (FFT-2048) as the associated IF is currently connected to PRR #1, where the HW task is implemented. Meanwhile, VM #1 cannot give orders to Dev #2 and #3 since these their IFs are currently read-only. Any writing on these IFs will cause a page-fault exception to the VMM. This type of exception will be handled as a VM's request for FPGA resources, which is automatically detected.

4.2. Hardware Task Model

HW tasks are accelerator instances running in PRRs containers. PRRs provide FPGA resources to implement their algorithms. A given PRR may not be compatible if its area (i.e. amount of resources) is insufficient to implement the corresponding accelerator algorithm of virtual device function. 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 PRRs for each virtual device are listed. For each compatible PRR, a *HW Task Descriptor* structure is provided, which stores the information of the corresponding bitstream, including its identifier, memory address and size. This information is used to correctly launch PCAP transfers and perform reconfiguration.

Figure 6 depicts the model of HW tasks and its interaction with VMs. As shown in this figure, VMs access HW tasks via IFs. We proposed a standard register structure to facilitate the multiplexing of PRR resources, denoted as the *partially reconfigurable (PR) accelerator interface*. It is implemented in the IF, and conveys the register values from the IF to HW task. Once the IF is connected to an HW task, a VM can operate on the IF registers to control the HW task behavior.

In Table 1, the structure of the *PR accelerator interface* is listed. VMs start the HW task workload by setting the *START* flag. When the required

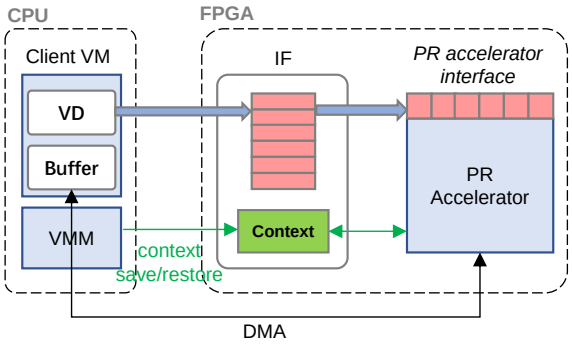

Figure 6: Reconfigurable accelerator model

computation is over, the *OVER* flag is set and the result is returned in the *RESULT* register. Additionally, HW tasks can be programmed to perform a DMA transfer or to generate interrupts.

Note that, since a *PR accelerator interface* structure is implemented in the IF, its register values are persistent for the VM. When an IF is disconnected from a PRR, the state of the corresponding virtual device (e.g. results, status) is still stored in this IF and can be read by the VM. In this way, the consistency of the virtual device interface is guaranteed.

4.3. Hardware Task Preemption

Considering that multiple VMs share FPGA resources, the RTOS tasks may be unexpectedly blocked when resources are occupied by GPOS tasks. To guarantee the timing constraints of real-time tasks, the HW tasks should be preemptible so that resources can be re-assigned to RTOS tasks when necessary. We denote the VM corresponding to a HW task as a *client*.

HW tasks inherit the priorities of their VM *clients*, meaning that virtual devices in RTOS and GPOS have different priorities. In our policy, the execution of low-priority HW tasks can be preempted when RTOS virtual devices require more FPGA resources. Note that HW tasks with the same priority level cannot be preempted.

The preemption mechanism requires to address several issues to make sure

Table 1: Ports Description in the *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

HW tasks can be safely stopped and resumed. First, to protect data integrity, accelerators may only be stopped when they reach some point in their execution, for example, the interval of data frames in communication processing. These points are denoted as *consistency points* where the execution path is safe to be interrupted and can be resumed 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.

Additionally, the context of HW tasks must be properly handled. We define the HW task context as the accelerator logic and the register states in the accelerator. The logic is stored in the bitstream file and is indexed in the *HW Task Index* table. On the other hand, the registers states depend on the design of accelerators.

As shown in Figure 6, in each IF, a 1KB buffer is implemented to store the accelerator context when preempted, which can later be used to resume its execution. Since the format of the saved context depends on the accelerator design, it is the designer's work to implement the save/restore routine of an accelerator. This routine is registered and called back by the VMM when preemption occurs.

Preemption is performed by making a context switch on a PRR i.e. stopping one HW task and reloading another. A complete context switch includes: (1) the reconfiguration of the accelerator logic by downloading a bitstream into the target PRR; (2) the saving and resuming of the corresponding register states, following user-designed routines. In the following we introduce how a PRR is designed to facilitate the preemption policy.

4.3.1. PRR State Machine

As a container, a PRR is allocated to HW tasks to provide FPGA resources and behaves as a state machine. The state determines if a PRR can be allocated to a specific HW task, and how it could be allocated.

Six states exist:

- ***Idle***: The PRR is idle without any ongoing computation and is ready for allocation.
- ***Busy***: The PRR is in the middle of a computation.
- ***Preempt***: The PRR is running, but the computation will be stopped (preempted) once it reaches a consistency point.
- ***Switch***: The PRR is in the middle of a context switch.
- ***Reconfig***: The PRR is in the middle of reconfiguration.
- ***Hold***: The PRR is allocated to a VM and is preserved for a certain amount of time.

The PRRs' behaviour can be described according to the flow chart given in Figure 7. As depicted in this figure, a PRR can be allocated in two states: *Idle* and *Busy*. It can only be directly allocated to VMs when it is in *Idle* state and requires no reconfiguration. In other situations, the PRR changes to the *Preempt* state to stop the current running task and to the *Reconfig.* state when performing PCAP reconfiguration.

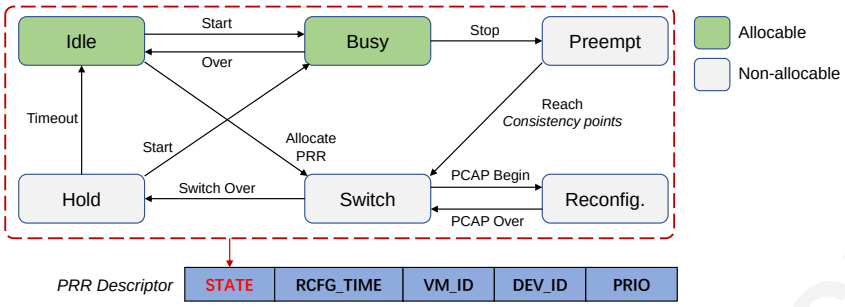

Figure 7: PRRs state machine

We have also introduced the *Hold* intermediate state. PRRs that are allocated to a VM will first enter this state. This indicates that the PRR is reserved for 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 when the preset waiting time *Expire* runs out.

A PRR holds the essential information in a *PRR Descriptor* data structure. This list indicates the PRR state (see Figure 7). 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 pre-defined by the size of the reconfigurable area. Therefore, the reconfiguration time of each PRR can be easily predicted. This factor is also included in the *PRR Descriptor*.

4.4. Management Mechanism

One major characteristic of virtualization is that VMs are totally independent from each other. In our case, however, VMs share reconfigurable resources. This can unfortunately lead to resource sharing issues that are well known in computing systems. In traditional OSs, such a problem can be solved by applying synchronization mechanisms like semaphores or spin-locks.

For Ker-ONE, such mechanisms are not suitable since they may undermine the independence of VMs. Therefore, our system introduces additional management mechanisms to dynamically handle the VMs' request for PRR resources.

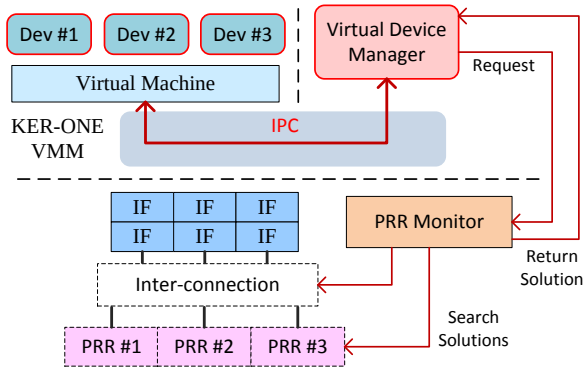

Figure 8: Overview of the DPR management framework in Ker-ONE

Note that such requests may occur randomly and are unpredictable.

In Figure 8, the proposed management mechanism is described, which mainly involves two components: a *Virtual Device Manager* on the software side and a *PRR Monitor* in the FPGA hardware.

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 their virtual devices. This is performed through an Inter-VM Commnunication (IVC) mechanism.

The *PRR Monitor* is running in the static part of the FPGA and is in cooperation with the *Virtual Device Manager* to dynamically monitor reconfigurable accelerators and search proper solutions to the VMs' requests.

4.4.1. PR Resource Requests and Solutions

As described earlier, every time a VM tries to use an unavailable virtual device, a page-fault exception is triggered and then handled by the *Virtual Device Manager* as a partially reconfigurable (PR) resource request: *Request (vm_id, dev_id, prio)*, which is composed of the VM ID, the device ID and a request priority. The device ID identifies the accelerator functionality. The request priority is equal to the priority of the calling VM. Note that, when a running HW task is preempted, the interrupted task is automatically composed as a request, indicating the corresponding virtual device still has unfinished

computation workloads.

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

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

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

- ***Assign****(prr_id)*: this solution directly allocates the returned PRR (i.e. *prr_id*), which is *Idle*, to the request VM. If the requested device *dev_id* is not implemented in this PRR, a *Reconfig* flag is also added.
- ***Preempt****(prr_id)*: all PRRs are *Busy* and none can be directly allocated, but the returned PRR (i.e. *prr_id*) can be preempted and re-allocated. If the requested accelerator (*dev_id*) is not implemented in this PRR, a *Reconfig* flag is also 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 Figure 7). 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. If multiple solutions are found, the best one is chosen according to the selecting policy.

In our algorithm, assigning *Idle* PRRs are considered to be best solutions. Preemption is 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.

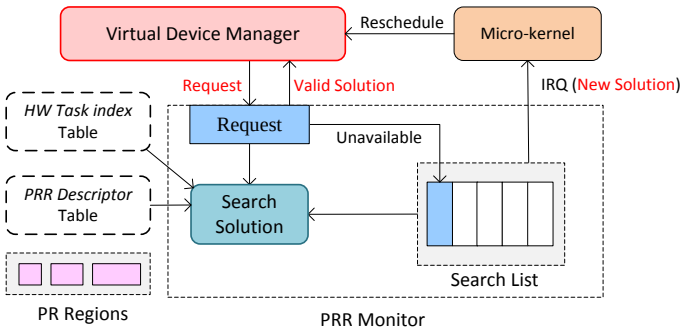

Figure 9: Solution searching in the *PPR monitor*

Figure 9 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 is added to the *Search List*, which is a waiting queue of all unsolved requests.

The *PPR Monitor* keeps searching solutions for requests in this queue on the FPGA side, and acknowledges the *Virtual Device Manager* whenever a new solution is found. The searching runs in parallel with VMs, following a priority-based FIFO principle, so that when a requests conflict occurs, the *PPR Monitor* always chooses the highest priority request.

4.4.2. Virtual Device Manager

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

As already explained, if any VMs try to use an unavailable virtual device, this will automatically be detected by the VMM, and then forwarded to the *Virtual Device Manager*.

In Figure 10, the full flow to allocate an accelerator to a VM is depicted. In this example, after a given $Request(vm01, dev01, prio01)$, a solution $\{Assign$

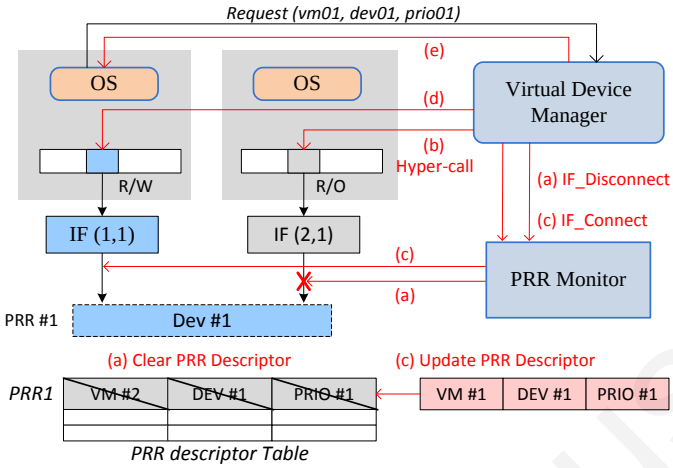

Figure 10: Execution flow for accelerator allocation

(prr01), non-Reconfig} is found. We assume that PRR #1 was previously used by VM #2 and that it is currently in the *Idle* state. In this case, it can then be directly re-allocated following these steps:

a) The *Virtual Device Manager* calls the user-registered context-save routine to save the states of VM #2's dev01. The *IF_Disconnect* command is sent to the *PRR Monitor* to disconnect the IF of VM #2. Meanwhile, the PRR#1 *PRR descriptor* entry is erased.

b) The no-more-available device IF is set as read-only in VM #2's page table (via hyper-call).

c) The *Virtual Device Manager* calls the user-registered context-resume routine to restore the states of VM #1's dev01. *IF_Connect* is used to connect the PRR to the IF of VM #1. The *PRR Monitor* also updates the *PRR descriptor* entry with the new VM #1 client.

d) The VM #1's dev01 IF is changed as read-write (via hyper-call).

e) The VMM suspends the *Virtual Device Manager* and resumes VM #1 to the exception point. VM #1 keeps on using this device.

Regarding guest OSs, the best solution in terms of latency is {*Assign, non-Reconfig*} in which a PRR can be immediately allocated. For other solutions

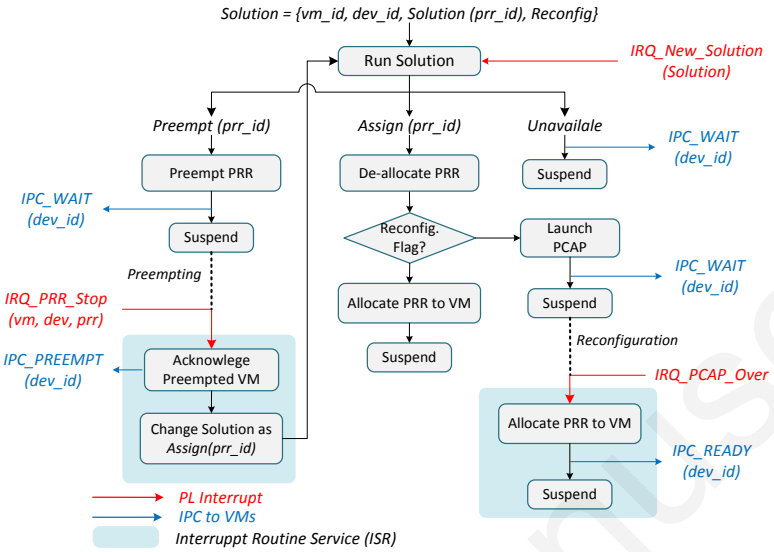

Figure 11: Virtual device manager handling *solutions*

requiring reconfiguration or preemption, the *Virtual Device Manager* informs the requesting VM with IPC messages, and suspends itself to wait for the end of reconfiguration or preemption. Meanwhile, the *PRR Monitor* keeps track of the unfinished solutions on the FPGA, and sends interrupts to the *Virtual Device Manager* whenever further operations are required.

This mechanism is explained in details in Figure 11, which demonstrates the role of the *Virtual Device Manager*. The program is composed of a main function *Run_Solution()* and two interrupts service routines (ISR) for interrupts *IRQ_PRR_Stop* and *IRQ_PCAP_Over*.

Preemption and reconfiguration solutions are performed in two steps: First, the manager launches the reconfiguration or preemption and then enters an idle state, Second, the manager is awakened to complete the unfinished solution in ISR. Note that for the *Preempt* solution, the manager first stops the preempted accelerator, and then handles it as a standard *Assign* solution.

In Figure 11, different signals are used to facilitate the allocation process and help synchronization among components. Some communication signals are destined to requesting VMs, and indicate the state of the required device. Others

Table 2: Communication signals for PRR Allocations, among *Virtual Device Manager* (VDM), *PRR Monitor* (PM) and VM.

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

are sent from the *PRR Monitor* to the *Virtual Device Manager*, to acknowledge the events for unfinished solutions. These signals are listed in Table 2.

4.5. User Programming Model

A major purpose of our framework is to considerably simplify the coding aspects of software applications by making the access to devices as transparent as possible. Ideally, the manipulation of virtual devices is only performed by read/write operations from/into the interface registers, without knowing the resource management at lower level. This is called the native programming model. In this case, the *IPC_WAIT* and *IPC_READY* signals are just ignored.

Alternatively, guest OS may follow the guest programming model, meaning the OS is aware of the true state of virtual devices. For example, when a virtual device is unavailable, the guest OS may be programmed to stop the waiting task, run other tasks and only resumes the suspended task when the *IPC_READY* signal is received.

For the RTOS tasks which are running at higher-priority level, they can always claim more FPGA resources when necessary, meaning that any request from them can be immediately solved by directly allocating or preempting PRRs from lower-level tasks. More importantly, once an RTOS is allocated with an accelerator, it is guaranteed to complete its computation since no preemption

is allowed

Therefore, RTOS tasks are safe to use native programming model, as if running on a native machine. This normally results in a longer execution time since the task can be blocked by the preemption and context switch of HW tasks. But the execution time is still deterministic since we avoid the unpredictable resource blocking caused by other VMs. On the other hand, in some special cases where the context switch involves a very long reconfiguration latency, the extra overhead may become costly. RTOS tasks can then use the guest programming model to avoid long CPU blocking.

For GPOS tasks, using native programming model is also workable, but can be unwise if the FPGA resources are relatively tight and their request for PRRs can always be blocked by RTOS tasks, which will cause unpredictable blocking time. Therefore, using the guest programming model is more appropriate for GPOS if FPGA resources are intensely shared. Additionally, with the *IPC_PREEMPT* signal, a GPOS is aware of its deprivation of FPGA resources. Advanced programming policies can be applied to improve the QoS of its tasks. For example, GPOS tasks can move the computation workload from the accelerator to the CPU, to avoid the long blocking of specific computations.

5. Ker-ONE Virtualization Performance Evaluation

In this part, the performance of our micro-kernel is provided. Several experiments have been led to measure the impact of virtualization and make sure that such a system can be used in very small real-time embedded systems.

The first experiment has focused on measuring the overhead of fundamental virtualization functions, such as VMM scheduling, hyper-calls, interrupt management, etc. Then the impact of virtualization on the RTOS execution has been quantified by measuring the overhead that is due to the VM scheduling. This study has been led using a standard RTOS benchmark. Finally, our platform has been used to implement specific applications taken from standard benchmarks to demonstrate its feasibility.

Our experiments were performed on the ARM Cortex-A9 processor of Xilinx ZedBoard (i.e. the Zynq-7000 SoC), and the frequency has been set as 667 MHz. In order to evaluate the performance of our platform, we have implemented multiple guest OSs (i.e. Mini-μC/OS-II) on top of Ker-ONE. These guest OS had to execute specific applications on a huge number of samples. Two main benchmarks have been considered, Thread-Metric [22] and MiBench [23].

In all our tests, the VMM scheduling period was set to 33 ms. Guest OSs used a 1 ms timer tick for their own schedule. These values are quite common timing configurations in this context and especially for μC/OS-II [24]. Guest OSs were either configured as GPOS or RTOS according to the experimental requirements.

5.1. Basic Virtualization Functions Overhead

The different measurements that have been performed in the experiments allowed us to identify the most critical VMM functions. The platform has been configured to host four similar μC/OS-II at the same priority level. These were considered as GPOS and scheduled according to a round-robin strategy. Software tasks were running in the guest OSs and making hyper-calls. The overheads of the corresponding VMM services that were required to handle these hypercalls have then been recorded by a background monitor during several hours. Figure 12 depicts the experiments results, where minimal, average and maximum overheads are presented in microseconds.

The overhead latency that is required to generate an hyper-call, to process this hyper-call in the VMM and to return back to a virtual machine has been evaluated. This corresponds to the VM entry/exit latency overhead. At this point, it is important to note that hyper-calls are generally performed by the guest OS and rarely by user tasks.

Since Ker-ONE is mapped to the VMs' address space, no switch between VM is required. Hyper-calls entries and exits are relatively low cost processes since they only involve the save/restore of the CPU context.

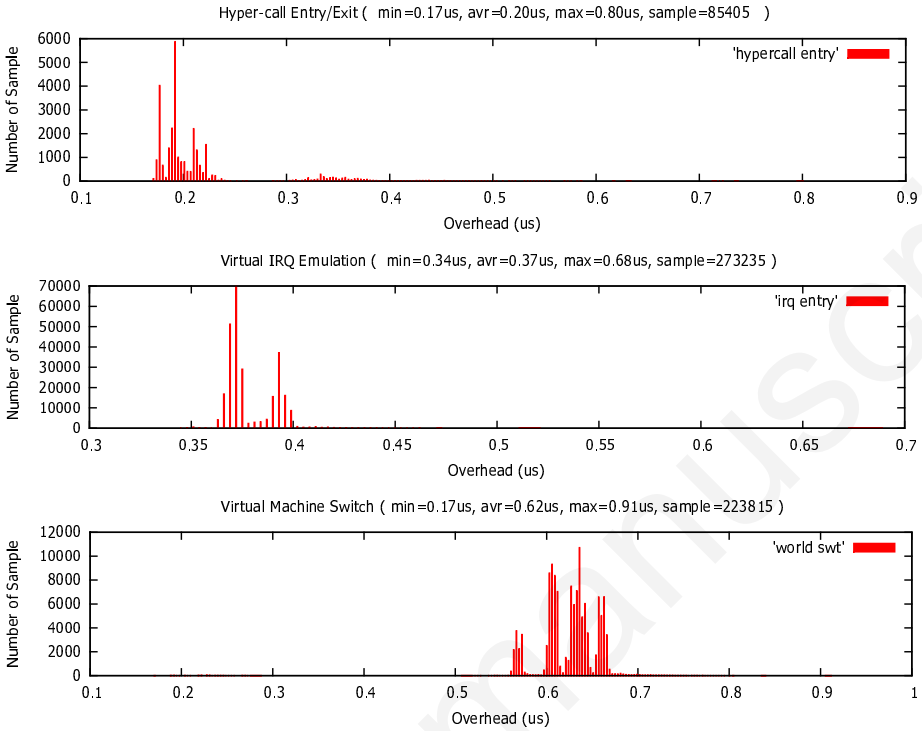

Figure 12: Basic virtualization functions overhead in microseconds (μs) with minimum, average and maximum values.

Another important metric is the virtual IRQ emulation latency that represents the cost of emulating a virtual interrupt for a VM. This functionality is critical for event-driven OSs and this latency has a huge impact on the events' response time. This metric is also closely related to the guest OS' scheduling overhead since a guest OS is driven by a virtual timer tick to handle virtual time. This overhead is measured from the physical event's arrival time until the time at which the VM is forced to its local exception vector. This process involves the handling of physical IRQ and the emulation of the virtual GIC interface registers.

The virtual machine switch latency represents the cost of switching from one VM to another and may be relatively heavy. The overhead of the virtual machine switch is one of the key metric in most virtualization approaches, as

it is usually quite cumbersome, and has a huge impact on the VMM efficiency. In Ker-ONE, this switch is performed when a VM consumes its time quantum and moves to its successor, or when it suspends itself and the VMM schedules another VM or process. This switch includes several major procedures: (1) re-scheduling; (2) vGIC context switch; (3) timer state update; (4) address space (page table) switch; and (5) CP15 registers update. Note that changing the address space causes a higher TLB/cache miss rate and thus increases the switch latency.

Usually, the VMM uses these functions for management and emulation purposes and they are of great importance. Virtualization efficiency is closely related to the performance of these functions. In our case, we can note that these functions exhibit low overheads. As shown in the results, frequently-called functions, i.e. hyper-calls and vIRQ emulation can be handled in less than 1 μs. Furthermore, the virtual machine switch overhead, which constitutes the most expensive process, could be limited to 1 μs.

5.2. RTOS Virtualization Evaluation

In this section, we quantify the impact of virtualization mechanisms on the performance of guest RTOSs. This includes the OS kernel services as well as scheduling overhead. In order to estimate the impact of virtualization, a controlled experiment has been performed. The control group implements a native RTOS on an ARM Cortex-A9 processor, while in the experimental group, the same RTOS is implemented on top of Ker-ONE. Differences in terms of performance have been measured.

In our experiments, Mini-μC/OS-II has been implemented as an RTOS in a VM. Three other VMs were used to host another instance of Mini-μC/OS-II, which plays the role of a GPOS. Benchmarks have run as applications in the RTOS and a comparison between native execution and execution on a VM has been performed for each test.

5.2.1. Benchmarking

We chose the Thread-Metric benchmark suite for the RTOS performance measurement [25]. Thread-Metric has been developed by Express Logic in 2007 and has been applied in several works to measure and compare the performance of multiple RTOSs [26].

In our experiment one RTOS and three GPOSs (all Mini-μC/OS-II) run on top of Ker-ONE. The Thread-Metric suite is executed on the RTOS. In order to obtain the performance loss due to virtualization, the benchmarks results on the native μC/OS-II are also collected and used as reference.

To provide an extensive evaluation, the XEN-ARM hypervisor has been evaluated to achieve a comparison with our micro-kernel. The XEN-ARM hypervisor Version-3.0 [27] has been ported to our platform that is based on a Zyng-7000 device. A para-virtualized μC/OS-II (denoted as xeno-μC/OS), that is available on the XEN website as been used as reference. The Thread-Metric benchmark has been executed on this kernel. Note that, since μC/OS-II runs on a single protection domain, no multiple page tables are necessary. Although XEN-ARM and Ker-ONE have different memory virtualization techniques, both virtualization contexts of μC/OS-II are similar in this case. The XEN's support of user-level multiple protection-domains has not been used to provide fair comparison.

The role of the Thread-Metric benchmark is to provide a set of common kernel services to compare different RTOS in terms of performance. These services mainly deal with context switch, interrupts handling, message passing, memory management, etc. For each OS service to be tested, the corresponding function as well as its dual function have been executed in pairs, e.g. allocating/ de-allocating memory, or sending/receiving messages. These functions were executed continually and the number of iterations has been evaluated. Finally, the number of iterations has been recorded every 30 seconds and denoted as test score. A high score means a low overhead in the OS kernel function and obviously better performance. The tests provided by Thread-Metric are:

- *Calibration Test*: A basic single-task rolling counter function to set up a performance baseline for comparisons.

- *Preemptive Context Switching*: Five tasks of different priorities are created. Staring from the lowest priority task, each task resumes the next higher priority task and suspends itself. The sequence of OS scheduling (i.e. OSTaskSuspend, OSTaskResume, OSSched in μC/OS-II) is evaluated.

- *Message Processing*: One task is created to repeatedly send and receive message through the OS message queue (i.e. OSMessagePost, OSMessagePend).

- *Memory Allocation*: One task that allocates and releases memory through the OS memory block (i.e. OSMemGet, OSMemPut).

- *Synchronization Processing*: One task that pends and posts semaphores (i.e. OSSemPost, OSSemPend).

- *Interrupt Handling*: One task is created to generate software IRQ. The semaphore mechanism is used in the IRQ handler routine to guarantee the handling completion.

- *Interrupt Preemption*: Two different priorities tasks are created. The lower priority task generates a software IRQ and while it is executing its IRQ handler routine, the other task is resumed and preempts the low priority one.

Based on the experiments above, the metric *Performance Ratio* has been defined and denoted as R_P, which is computed as:

$$R_P = \frac{S_{vm}}{S_{native}} \times 100\%, \tag{2}$$

where S_{vm} is the benchmark score obtained by the guest OS, and S_{native} concerns the native OS. R_P measures the influence caused by virtualization. A better virtualization technology means less performance loss and thus a higher

R_P value. Table 3 presents the experimental results of the Thread-Metric benchmarks running on both Ker-ONE and native environments, and the corresponding performance ratios.

Table 3: Thread-Metric benchmarks results for both native and virtual μC/OS-II

Test Object	**Native μC/OS-II**	**VM μC/OS-II**	**Performance Ratio (%)**
Calibration Test	764458	753879	98.6
Preemptive Context Switching	32113328	28927171	90.1
Message Processing	18431136	16748720	90.9
Memory Allocation	104601611	85091278	81.3
Synchronization Processing	108589466	90893213	83.7
Interrupt Handling	32541832	25768399	79.2
Interrupt Preemption	19089282	16425610	86.0

As shown in Table 3, μC/OS-II has lower performance when virtualized. This is predictable since the benchmark tests include intense executions of sensitive instructions and privilege operations on protected system resources. One typical operation is the context switch in a guest OS. Originally, this is performed with only a few lines of assembly code. However, in a para-virtualized implementation, this operation is normally re-directed to a bunch of assembly lines of code and involves multiple hyper-calls and VMM handling. In our test, when a context switch is performed frequently, a noticeable extra overhead is caused compared to the original code.

Timer and interrupt virtualization also degrade the performance. The emulation of such mechanisms is particularly expensive if guest OS executes very fast or/and require frequent interrupts. This may be noticed in the *Interrupt Handling* and *Interrupt Preemption* benchmarks that are presented in Table 3. In these benchmarks, a huge number of interrupts are generated and handled. This results in a significant performance degradation and explains the relatively low performance that is obtained in these tests.

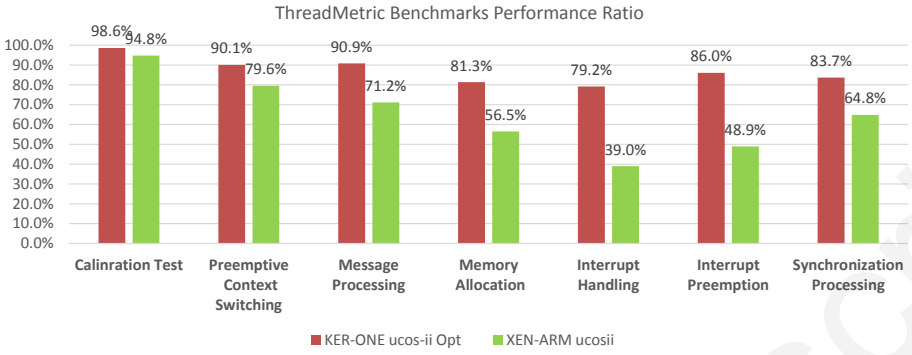

Figure 13: Comparison of Thread-Metric *Performance Ratio* (R_P) for para-virtualized μC/OS-II on Ker-ONE and XEN-ARM.

In Table 3, we can also notice that our micro-kernel performs well when hosting RTOS. Regarding the services that are evaluated in the benchmarks, most losses are under 20%. For some functions such as task scheduling and message processing, the performance is even better and close to those obtained with native OS: only 10% of performance loss. This is due to the fact that, in the Ker-ONE design, virtualization of resources have been optimized using a shared memory region (see Section 3.2.1), which reduces the number of hyper-calls and provides significant improvement in terms of performance.

Figure 13 provides a comparison between two different systems that are implemented in the same platform. The first is the Ker-ONE kernel. The second is the Xen-ARM hypervisor. We may also note that Ker-ONE performs better than XEN-ARM when hosting the μC/OS-II guest OS.

At this point, it is important to notice that both kernels make use of a share memory region. The difference in terms of performance is due to the fact that Ker-ONE provides a simpler virtualization interface. All virtual resources are implemented with smaller structures of smaller size. Additionally, Ker-ONE provides a dedicated physical timer pass-through for guest RTOS (see Section 3.3.1), which largely simplifies the timer virtualization compared to XEN system.

Regarding the *Interrupt Handling* and *Interrupt Preemption* benchmarks, we

may also note that XEN-ARM performs obviously worse compared to Ker-ONE. This may be explained because of the virtual interrupts that are handled differently in XEN-ARM. In this hypervisor, these are manipulated as *event_channels* that separate physical IRQs from VM event ports. This strategy is efficient to ensure isolation between virtual machines but is also more complex.In our approach, Ker-ONE implements a simple virtual IRQ management that is oriented towards the GIC emulation. A simple function forwards the physical interrupts to the VMs. Moreover, the different VMs keep on using their own IRQ handlers, which simplifies the system.

5.2.2. RTOS Virtualization Overhead

Whereas the previous analysis has evaluated the performance of specific OS functions with the Thread-Metric benchmark suite, we also created our own custom benchmarks to estimate the scheduling and context switching overhead. With these benchmarks, schedulability studies may be performed as described in [28].During these tests, we carefully evaluated the worst-case RTOS task response time. We have noticed that this occurs when the RTOS preempts the GPOS to get scheduled.

We define this response time as $Response^{VM}$, which is composed of: delays caused by the VMM critical execution ($\Delta VMMcritical$), by VMM scheduling ($\Delta VMMsched$) and by RTOS task release ($relEv^{VM}$). These three types of overhead have an impact on the release delay of RTOS tasks as demonstrated in [29]. In our experiment, these types of overhead have been measured respectively and recorded during hours of execution.

A total number of 1,048,576 samples have been obtained during long time experiments. For each measurement, we have evaluated the minimal, average and maximal overhead. In the following discussion, we consider the maximal experimental measurement as the worst-case execution time (WCET).The results of the evaluation are shown in Figure 14.

The *Critical Execution* (i.e.$\Delta VMMcritical$) measures the overhead of the VMM critical execution when IRQs are masked. Any events occurring in this

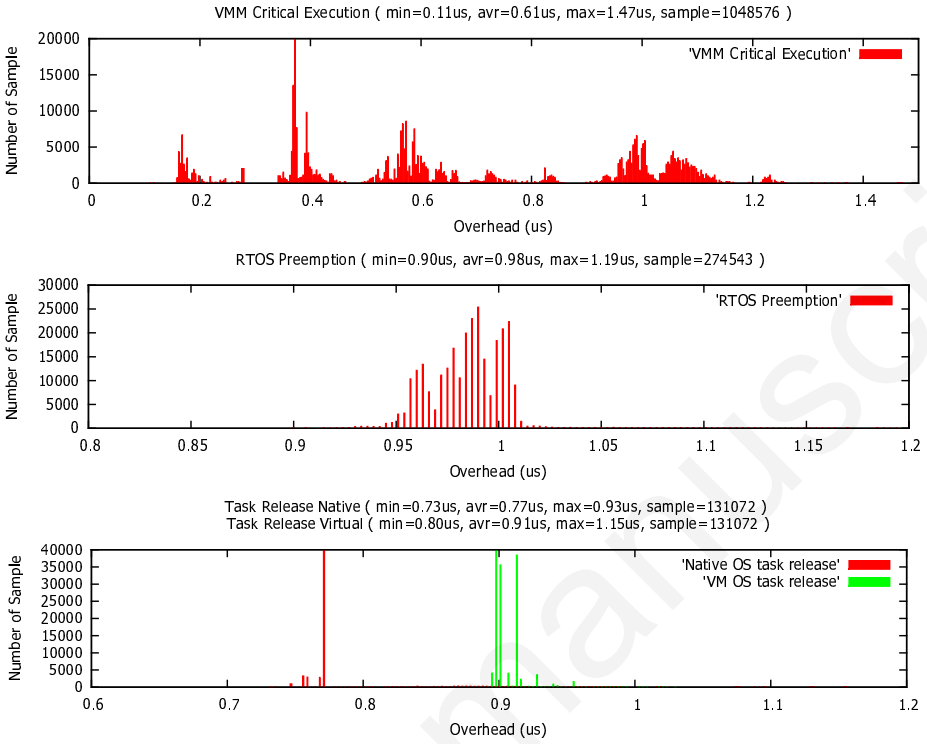

Figure 14: VM RTOS task response overhead in microseconds (μs) with minimum, average and maximum values

period are delayed until the critical execution is over. When VMs run, the VMM performs critical execution for various reasons, i.e. hyper-calls, IRQs, exceptions or VM switches. In order to cover all possible critical execution overheads, we have executed dedicated test software which helped trigger all possible hyper-calls, IRQs and exceptions. As shown in Figure 14, the worst-case VMM critical execution has been estimated at 1.47 μs.

The *RTOS Preemption* (i.e.$\Delta VMMsched$) refers to the cost of an RTOS preempting the current GPOS. This process is performed by the VMM and includes several steps: (1) real-time event handling (timer tick interrupts in our test), (2) rescheduling, (3) VM switch, (4) forwarding the timer interrupts to the RTOS. As described in Figure 14, RTOS preemption is completed after an average delay of 0.98 μs, whereas the WCET is 1.19 μs.

In Figure 14, the *Task Release* latency represents the time that is required to handle a virtual timer tick in an RTOS and to schedule a new task. Two latencies have been measured for native and VM corresponding to $relEv^{Native}$ and $relEv^{VM}$, respectively. A loss of performance is to be expected in virtualization in the Task Release latency. This is mainly due to the emulation of sensitive instructions that are required to handle interrupts and to perform a context switch. It follows that the worst-case extra *Release Event* overhead can be estimated as:

$$\Delta_{VM}^{relEv} = relEv_{\boldsymbol{(WCET)}}^{VM} - relEv_{\boldsymbol{(BCET)}}^{Native}, \tag{3}$$

where $relEv_{(BCET)}^{Native}$ is the best-case execution time of the native latency.

In this equation, Δ_{VM}^{relEv} has been estimated at 0.42 μs. Therefore, the equation that gives the total influence that virtualization causes on the RTOS response time $\Delta_{VM}^{Response}$ may be written as:

$$\Delta_{VM}^{Response} = \Delta VMMcritical + \Delta VMMsched + \Delta_{VM}^{relEv}. \tag{4}$$

According to the experiment results, $\Delta_{VM}^{Response}$ has been estimated at 3.08 μs. Considering that, the scheduling tick is usually set as 1ms or 10ms in RTOS, the virtualization overhead can be neglected in terms of the real-time task response time. Therefore its influence on real-time schedulability can be ignored.

6. Reconfigurable Accelerator Management Evaluation

In this section we evaluate how the tasks' execution time is influenced by the FPGA resources sharing and determine the tasks WCET. To this purpose, we define the allocation latency L_{alloc}, which corresponds to the delay that is required before an accelerator (i.e. FPGA resources) is properly allocated and ready to start. This latency can be seen as the response time of a virtual device and can be used to represent the increase of a task execution time when running

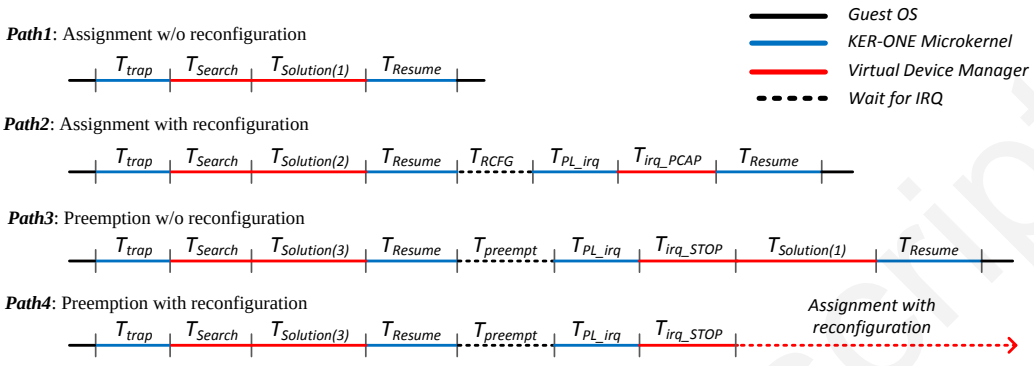

Figure 15: Execution paths of DPR resources allocation

on Ker-ONE. Note that only RTOS tasks are considered here. In this configuration, virtual devices can always demand resources and cannot be preempted, meaning that the allocation latency can be determined via static analysis.

The allocation latency has two main sources: the allocation mechanism itself and the Ker-ONE micro-kernel functions. Additional overhead is to be deplored 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 Figure 15.

In these models, the allocations consist of four different solution paths that can be decomposed into the following list of 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 due to the preemption of the current accelerator.

Based on this model,the worst-case allocation latency can be determined as follows:

$$L_{alloc(\mathbf{WCET})} = \max\left\{T_{Path1}, T_{Path2}, T_{Path3}, T_{Path4}\right\}. \quad (5)$$

In order to estimate and analyze the impact of L_{alloc}, an experiment has been led and described in section 6.1.

6.1. Experimental Description

As mentioned earlier, our experiments were performed on the Xilinx Zed-Board (Zynq-7000 SoC). This SoC consists of two parts: the processing system (PS) which provides a dual-core ARM Cortex-A9 processor, and the programmable logic (PL) which includes a partially reconfigurable FPGA fabric. The CPU operating frequency has been set 667 MHz and the FPGA logic was driven by a 100 MHz clock.

The proposed experiment is shown in Figure 16. The FPGA fabric on PL side has been initially implemented with three PRRs of different sizes. Four hardware accelerators, i.e. *QAM16, QAM64, FFT512, FFT1024,* have been implemented and stored into bitstream files. During the initialization stage of Ker-ONE, these files have been loaded into the RAM memory and are only accessible by 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 conditions. QAM blocks aim 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.

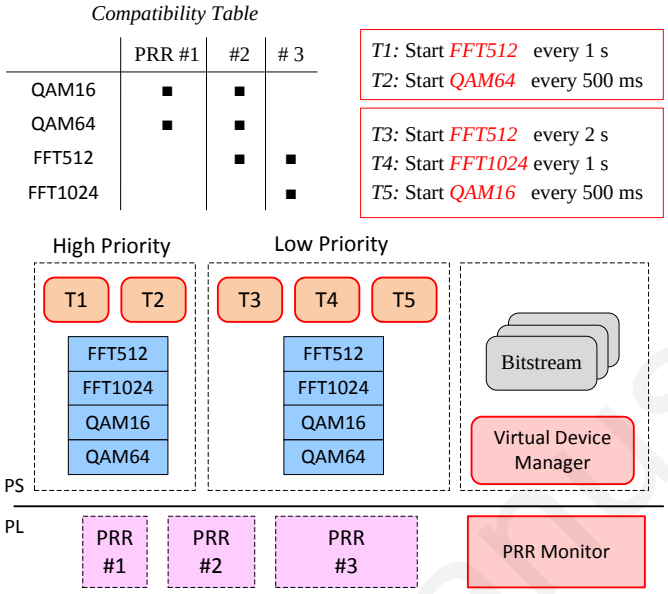

Figure 16: Experimental architecture for performance evaluation

To simplify the experiment, we assume that the FFT always works in sequence with QAM-16 algorithm. The data frame is set to be 18,800 bits, according to the 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 data buffer keeps transferring frames from VM memory space to the accelerators.

Note that the compatibility table shows that PRRs are unsuitable to certain accelerators. For example, PRR #1 cannot host an FFT module because of lack of resources. PRR #3 is incompatible with QAM since PRR #3 is of large bulk of resources while QAM modules require little. In this case, hosting a QAM modulator will cause a waste of resources.

Regarding the guest OSs running in virtual machines, we still use the modified μC/OS-II to execute on top of Ker-ONE. Two μC/OS-II guests are hosted with different priority levels as RTOS and GPOS.

For each guest OS, four available virtual devices have been implemented. Two and three tasks run respectively in both guest OSs to periodically com-

Table 4: Overhead measurement during DPR allocation

Micro-kernel		**Virtual Device Manager**	
Operation	**Overhead (μs)**	**Operation**	**Overhead (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

mand 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. In order to respect the integrity of the OFDM process, both QAM and FFT modules may be preempted only when their data frame is completely processed.

The experiment ran for several hours continuously. A custom monitor has been built to measure and record the various costs of allocation mechanisms on the RTOS tasks.

6.2. Overhead Analysis

The measurement results of atomic execution overheads are provided in Table 4. According to this table, it may be seen that VM scheduling as well as virtual interrupt emulation are performed with a low overhead that is less than $1\mu s$. The highest overhead is obtained in $T_{Solution(2)}$, which occurs when a PRR is assigned with reconfiguration. In this case, this process requires a PCAP transfer which is time consuming since it consists of complex operations to organize the download of bitstream files.

According to the performed measurements, the allocation latency of different

Table 5: Reconfiguration and preemption delays

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

[1] The worst-case waiting time when a running accelerator is forcibly stopped.

solution paths, as modeled in Figure 15, can be estimated as:

$$\begin{aligned} T_{Path1} &= 3.03\mu s, \\ T_{Path2} &= 6.76\mu s + T_{RFCG}, \\ T_{Path3} &= 5.10\mu s + T_{preempt}, \\ T_{Path4} &= 9.96\mu s + T_{preempt} + T_{RFCG}. \end{aligned} \tag{6}$$

We may notice that a 3 μs latency is obtained for a direct allocation. Other solutions have additional latencies due to preemption ($T_{preempt}$) or reconfiguration time. The costs of $T_{preempt}$ and T_{RCFG} are mostly depending on the implementation and application of accelerators.

In Table 5, these costs are evaluated for all available accelerators. T_{RCFG} is determined by the size of the bitstream, and therefore corresponds to three PRR areas. The preemption time $T_{preempt}$ is determined by the δ_{pre} of the accelerator to be preempted. δ_{pre} corresponds to the worst-case waiting time when preempted, and depends on the *consistency points* which are set as the interval of data frames.

In terms of WCET analysis (i.e. $T_{preempt(\mathbf{WCET})}$ and $T_{RCFG(\mathbf{WCET})}$), it is important to note that they not only depend on the implementation, but also on the accelerators are being globally designed and used.

For example, considering the compatibility shown in Table 5, a QAM-16

accelerator cannot preempt a FFT1024 since there is no resource competition between them, so $T_{preempt(\mathbf{WCET})}(QAM16)$ is calculated as:

$$T_{preempt(\mathbf{WCET})}(QAM16) = \max\left\{\delta_{pre}^{QAM16}, \delta_{pre}^{QAM64}, \delta_{pre}^{FFT512}\right\}. \quad (7)$$

Meanwhile, since QAM-16 can only be implemented in PRR #1 and #2, the value of $T_{RCFG(\mathbf{WCET})}(QAM16)$ is determined as:

$$T_{RCFG(\mathbf{WCET})}(QAM16) = \max\left\{T_{RCFG}^{PRR1}, T_{RCFG}^{PRR2}\right\}. \quad (8)$$

Therefore, for each accelerator, its worst-case allocation latency $L_{alloc(\mathbf{WCET})}$ can be calculated by obtaining $T_{preempt(\mathbf{WCET})}$ and $T_{RCFG(\mathbf{WCET})}$ according to the system design, and then following the equation 5.

Note that, the implementation of the PRRs and accelerators are set beforehand and the RTOS tasks' access to accelerators are also known. For each RTOS task, the impact of $L_{alloc(\mathbf{WCET})}$ can be predicted and be added to its WCET value for the RTOS schedulability analysis.

6.3. Discussions

From Table 5 we can notice that for the accelerators used in our experiment, $T_{preempt}$ is significantly lower than T_{RCFG}. Therefore, from the RTOS point of view, preemption is always the best solution since it encourages to benefit from existing accelerators of low priority tasks, and reduces the need for reconfiguration. However, for GPOS tasks, being preempted will block their execution.

For a system in which preemptions may occur frequently, it is possible that a GPOS may never get access to hardware resources. Hence, a trade-off should be made regarding the allocation policy.

In our work, we made the assumption that allocating *Idle* PRRs is always better than preempting them. The reason is that we want to make sure that low priority tasks will not be infinitely blocked by FPGA resources. The *PRR Monitor* has been designed accordingly. In a system that manages critical tasks of tight timing constraints, a new policy may be followed that gives more importance to preemption.

Table 6: Comparisons between SW and HW implementation

Algorithm	$T_{HW}(\mu s)$ (per frame)	$T_{SW}(\mu s)$ (per frame)	FPGA Resource Usage[1]
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%

[1] For purpose of simplicity, we present the resource usage portion on the total FPGA fabric, instead of the detailed amount of LUT, FF, etc.

In Table 6, we compare the HW acceleration approach with software. The results show that the accelerator performance of heavy computation (i.e. FFT512/1024) significantly surpasses software implementation. 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(e.g QAM), although hardware accelerators are still faster, this advantage gets undermined when taking T_{RCFG} into account. These results indicate that DPR technology is more suitable for large complex computation algorithms.

Furthermore, in this example, the FPGA is capable of simultaneously providing total 8 virtual devices with only 3 PRR areas, whose total cost is around 23% of the available resources (2%, 8% and 13% respectively). More importantly, from the above analysis it can be concluded that the real-time schedulability of RTOS VM is not undermined. Considering traditional FPGA design, to support both VMs, all 8 accelerators need to be implemented as static circuits, which may take up to 50% resources. Therefore, in our approach, the usage of FPGA is greatly reduced while the real-time safety can be preserved.

7. Conclusion

In this paper we have introduced an hypervisor which facilitates the DPR resource management in a system composed of several virtual machines. 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 overhead 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 and guaranteed real-time schedulability. 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] J. Becker, M. Huebner, G. Hettich, R. Constapel, J. Eisenmann, J. Luka, Dynamic and partial fpga exploitation, Proceedings of the IEEE 95 (2) (2007) 438–452.

[2] G. Heiser, The role of virtualization in embedded systems, in: Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems, IIES '08, ACM, New York, NY, USA, 2008, pp. 11–16. doi:10.1145/1435458.1435461.
URL http://doi.acm.org/10.1145/1435458.1435461

[3] L. Xu, Z. Wang, W. Chen, The study and evaluation of arm-based mobile virtualization, International Journal of Distributed Sensor Networks 2015

(2015) 1:1–1:1. doi:10.1155/2015/310308.
URL https://doi.org/10.1155/2015/310308

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

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

[6] D. Göhringer, M. Hübner, E. N. Zeutebouo, 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.

[7] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, 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.

[8] D. V. Vu, O. Sander, T. Sandmann, J. Heidelberger, S. Baehr, J. Becker, On-demand reconfiguration for coprocessors in mixed criticality multicore systems, in: High Performance Computing Simulation (HPCS), 2015 International Conference on, 2015, pp. 569–576. doi:10.1109/HPCSim.2015.7237094.

[9] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, C. Plessl, Reconos: An operating system approach for reconfigurable computing, Micro, IEEE 34 (1) (2014) 60–71.

[10] C.-H. Huang, P.-A. Hsiung, Hardware resource virtualization for dynam-

ically partially reconfigurable systems, Embedded Systems Letters, IEEE 1 (1) (2009) 19–23.

[11] W. Wang, M. Bolic, 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.

[12] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, 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.

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

[14] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, D. L. Maskell, Virtualized execution and management of hardware tasks on a hybrid arm-fpga platform, Journal of Signal Processing Systems 77 (1-2) (2014) 61–76.

[15] T. Xia, J.-C. Prévotet, 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.

[16] M. Marchesotti, M. Migliardi, R. Podestà, A measurement-based analysis of the responsiveness of the linux kernel, in: Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on, IEEE, 2006, pp. 10–pp.

[17] P. Regnier, G. Lima, L. Barreto, Evaluation of interrupt handling timeliness in real-time linux operating systems, ACM SIGOPS Operating Systems Review 42 (6) (2008) 52–63.

[18] G. Heiser, B. Leslie, The okl4 microvisor: convergence point of microkernels and hypervisors, in: Proceedings of the first ACM asia-pacific workshop on Workshop on systems, ACM, 2010, pp. 19–24.

[19] C. Dall, J. Nieh, Kvm/arm: the design and implementation of the linux arm hypervisor, ACM SIGARCH Computer Architecture News 42 (1) (2014) 333–348.

[20] I. Shin, I. Lee, Compositional real-time scheduling framework, in: Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International, IEEE, 2004, pp. 57–67.

[21] S. Xi, J. Wilson, C. Lu, C. Gill, Rt-xen: Towards real-time hypervisor scheduling in xen, in: Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on, IEEE, 2011, pp. 39–48.

[22] `https://rtos.com/support/extra-tools/`.

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, IEEE, 2001, pp. 3–14.

[24] S. Yoo, C. Yoo, Real-time scheduling for xen-arm virtual machines, IEEE Transactions on Mobile Computing 13 (8) (2014) 1857–1867.

[25] ExpressLogic, Measuring Real-Time Performance Of An RTOS (2007). URL `https://rtos.com/support/extra-tools/`

[26] J. Best, Real-time operating system hardware extension core for system-on-chip designs, Ph.D. thesis, School of Engineering,University of Guelph (2013).

[27] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park, C. R. Kim, Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones, in: 2008 5th IEEE Consumer Communications

and Networking Conference, 2008, pp. 257–261. doi:10.1109/ccnc08.2007.64.

[28] D. B. Stewart, Measuring execution time and real-time performance, in: Embedded Systems Conference (ESC), 2002, pp. 1–15.

[29] M. Aichouch, J.-C. Prevotet, F. Nouvel, Evaluation of an rtos on top of a hosted virtual machine system, in: Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on, IEEE, 2013, pp. 290–297.