Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration

Tian Xia, Jean-Christophe Prévotet, Fabienne Nouvel

Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration

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

***Abstract*—Today, ARM is becoming the mainstream family of processors in the high-performance embedded systems domain. In this context, adding a run-time reconfigurable FPGA device to the ARM processor into a single chip makes it possible to combine high performance and flexibility. In this paper, we propose a low-complexity design of system virtualization running on the Zynq platform. Virtualization of software and hardware resources are managed by a custom microkernel. The dedicated features to efficiently manage the dynamic partial reconfiguration (DPR) technology are described in details. The performance of the DPR management is evaluated and presented at the end of this paper.**

I. Introduction

The virtualization technology has been under intense research for the last decade. It has already been demonstrated that employing such a technique provides users with increased energy efficiency, shortened development cycles and other advantages. In the embedded computing domain, this technology has also gained a lot of interests and achieved enormous progress. For current devices, such as mobile phones or automotive electronics, applications and peripheral drivers are generally reprogrammed for each operating system (OS). Thanks to virtualization, applications can be added regardless of the OS, therefore reducing the development cycle and the time to market [1]. Moreover, since the real-time characteristic is still critical in a lot of applications, virtualization may also provide a solution to simultaneously host real-time OS (RTOS) and high-level generic OS (e.g. Linux, Android) on a single unified platform. Furthermore, whereas security issues are often met in modern mobile devices, applying virtualization makes it possible to efficiently protect systems from malicious applications and untrusted users.

In parallel, in the embedded computing domain, a new architecture based on the combination of ARM cores and FPGA fabric turned out to be an emerging solution for higher-performance embedded systems. With the improvement of FPGAs and especially with the dynamic partial reconfiguration (DPR) technology, an embedded system can dynamically dispatch and manage hardware accelerators as flexible software functions. The FPGA is then seen as a full-featured coprocessor. This emerging technology is especially suitable for computationally intensive applications in the digital communication field [2]. With an efficient management of both hardware and software tasks, the overall performance can be drastically improved.

In this paper, we present an approach to exploit the potential of CPU-FPGA systems by proposing a virtualization framework taking advantage of both virtualization and DPR techniques. We will describe and examine a virtualized embedded system in a hybrid ARM-FPGA platform Xilinx Zynq-7000. This system is based on the Mini-NOVA microkernel, an adjusted version of the NOVA (x86) microhypervisor [3]. Our version features custom mechanisms that enable the management of DPR shared resources.

The remainder of the paper is organized as follows: Section II presents the background concepts and researches in both reconfigurable computing and virtualization domain for embedded systems. In Section III, we describe the overall architecture and virtualization features of our custom microkernel named Mini-NOVA. Section IV presents the mechanisms of the DPR management in a virtualized environment. In Section V, we verify and examine the proposed system with practical hardware/software applications and analyze the results. Then the conclusion and prospect of our work will be given in Section VI.

II. Backgrounds

A. Concepts

The motivation of microkernel-based systems is to achieve the minimized trust computing base (TCB) size by using a small kernel that implements only a minimal set of abstractions. [4] identifies three key features for microkernels: address space management, threads control and inter-process communication. Other functionalities are available as extensions running at the user level. Microkernels generally reduce the complexity of kernels and increase security and flexibility. The principle of a microkernel-based virtualization is to establish an abstract layer to host multiple OS(es) simultaneously on a single processor. Usually, each guest OS runs in a virtual machine (VM), which consists of an abstraction of a real system. These virtual machines are managed by a virtual machine monitor (VMM) that is a kernel component. The VMM must ensure that each VM

is logically and temporally isolated from the others. While running multiple VMs, the VMM saves and resumes the guests' execution content to share resources among them.

Virtualization systems are built in split privilege-levels. These levels are used to separate the critical code from the less critical one. The higher privilege level space is always of higher authority to resources and forbidden to be accessed from lower privileges. For instance, the x86 Intel CPU has 4 privilege levels (referred to as Rings), where the critical software, normally the kernel, uses the highest privilege level (Ring 0) and the user applications use the lowest level (Ring 3). Using virtualization, the OS kernel is ported to the lower privilege level (de-privileged) and its behavior is limited because the critical resources are kept under the VMM's control. The instructions towards these resources are defined as sensitive instructions and should be detected by the VMM. On the other hand, several instructions can only be run in higher privilege level, which are referred to as privileged instructions. Executing privilege instructions at lower privilege level will cause CPU exceptions. In this case, the privilege instruction is trapped and will then be detected and emulated by the VMM. For sensitive instructions that do not trap in a lower privilege level, extra mechanisms are used to force them to trap into the VMM. [5] Currently there are two main approaches related to the virtualization technology:

(1) Full virtualization: In this configuration, guest OS(es) are hosted in VM(s) without any code modification. They are completely unaware of the virtualized environment in which they are running. This approach relies on the emulation of low-level drivers by the VMM. Note that these operations are usually performed by the guest OS kernel, itself. Unfortunately, to take advantage of this approach, the processor has to propose hardware mechanisms that are able to determine which OS have the rights to perform specific privilege operations, e.g. have access to some hardware peripherals. Today, a small set of processors implement these mechanisms, especially in the embedded domain, e.g. ARM Cortex-A15 and Intel VT-x.

(2) Para-virtualization: in this configuration, guest OS should be revised to get de-privileged i.e their internal code has to be modified. In this approach guest OS kernels generate hypercalls to the VMM for sensitive operations instead of the trap & emulate mechanism. This technology normally results in better performance and a smaller TCB size, but demands higher design complexity.

B. Related works

The approach of developing efficient embedded CPU-FPGA based systems has been studied in a lot of researches. Numerous works have focused on providing OS or VMM support to current reconfigurable FPGA devices. One successful approach in this domain is ReconOS [6], which is based on an open-source RTOS (eCos) and supports multithreaded hardware/software tasks. In [7], another specific OS, CAP-OS, has been proposed, that is based on the RAMPSoC architecture to manage resources in a multiprocessor reconfigurable system.

Those approaches, however, are less likely to be appropriate in scenarios where applications with different constraints are running concurrently, ranging from the hard real-time safety system to the less constrained personal entertainment applications. Within a mixed critical system like this, microkernel-based virtualization seems to be a better solution than a conventional generic OS, because of its small trust computing base, software security, flexibility and real-time capabilities [8][9]. However, for embedded systems, especially the ARM processors prior to the ARMv7A architecture, no hardware virtualization extension has been foreseen. Therefore, numerous researches have been led to paravirtualize ARM systems. One well-known work in this domain is the OKL4 Microvisor [9], based on the L4 microkernel. Other researches include the Xen-ARM hypervisor [10] and ARMvisor [11]. Beginning from the Cortex-A15 processor, hardware virtualization extension has been added to the ARM architecture, by introducing a new HYP privileged level. This makes full virtualization possible for ARM systems. Emerging researches in this domain include the revised OKL4 kernel [12], CASL hypervisor[13] and the KVM/ARM[14].

Despite of the wide study for the embedded ARM virtualization, there are still few works considering the combined ARM-FPGA platforms, especially dealing with DPR features. [15] proposed a revised Xen kernel to support DPR in virtualization environment, based on x86 Intel architecture. In [16], another approach to support DPR in virtualization system was made, focusing on the cloud computing on servers, rather than in embedded systems. In [8], a microkernel-based hypervisor is proposed to manage an ARM-FPGA architecture. However, it did not consider the DPR technique. Virtualization features are not fully revealed and discussed in this work, either.

In our former work [17], we proposed a simple ARM based microkernel without virtualization implementation. In this paper, we propose a paravirtualization system based on the Mini-NOVA microkernel, capable of dynamically managing software and DPR hardware tasks. By introducing the virtualization into original works [17], major modifications and novel mechanisms are required for efficient coordination of virtual machines and reconfigurable hardware accelerators, e.g. the paravirtualization extension, the porting of guest operating systems, and the security issues for teh shared FPGA resources. In this paper we also evaluate and analyze in details the virtualization performance of our microkernel, proving that this approach enables to gain better performance and enhances the ARM-FPGA architecture combining higher flexibility, security and reliability.

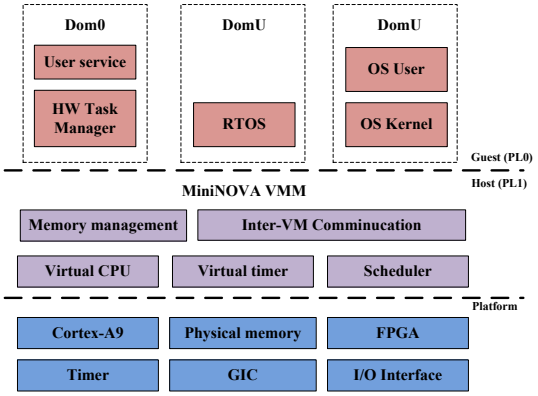

Figure 1. Mini-NOVA Architecture Overview

III. Mini-NOVA Virtualization on Cortex-A9

In this section, we introduce the implementation of Mini-NOVA virtualization on ARM platform. Mini-NOVA is a revised and simplified version of the NOVA micro-hypervisor (x86), and has been ported on the ARM Cortex-A9 architecture, which is the latest ARM version available for contemporary ARM-FPGA platforms. The principle of Mini-NOVA is to reduce complexity to reach lower overhead, smaller TCB size and higher security, making it more flexible and portable for embedded devices. Due to the absence of hardware virtualization support in the Cortex-A9 architecture, paravirtualization is used in Mini-NOVA.

The Cortex-A9 architecture offers 6 main operating modes, which are divided into two privilege levels: non-privileged PL0 (USR mode) and privileged PL1 (SVC, IRQ, FIQ, UND and ABT modes). Mini-NOVA is mainly executing in the supervisor (SVC) mode, occupying the privileged level, while guest OS(es) are running in the user (USR) mode. Other modes are mainly used to respectively trap different types of exceptions: interrupt, Undefined Instruction (UND) and Prefetch/Data Abort (ABT). These exception types are used to build the virtualized environment. Interrupts are trapped into the IRQ and fast IRQ modes (IRQ/FIQ). UND exceptions are mostly caused by unpermitted instructions on the system registers or coprocessors. They are usually used to trap privilege instructions. ABT exceptions are caused by illegal memory access attempts, such as page faults, and are used for the virtualized memory space management. Whenever an exception occurs, the CPU leaves the user mode and enters the corresponding exception mode, which would later give control back to the SVC mode to handle this exception.

The Mini-NOVA kernel is an abstract layer between the physical resources and software users. For each guest OS/Application, a virtual machine is initiated, running in an isolated user domain. A virtual machine monitor is used to create the virtualized environment for VMs. Based on the microkernel features [4], the VMM should provide VMs with four basic properties: CPU virtualization, memory management, communication, and scheduling. To minimize the TCB size of the privileged code, we decompose the microkernel and implemented parts of its properties at user level. The overview of Mini-NOVA is illustrated in Fig. 1.

Table I. Virtual CPU Content in Mini-NOVA

Privilege level	Resources	Switch mechanism
Non-privileged	General-Purpose Registers Platform-specific timer	Active switch
	Vector Floating-Point (VFP)	Lazy switch
Privileged	Coprocessor Registers(CP14/CP15) Generic Interrupt Controller (GIC) Memory Management Unit (MMU)	Active switch
	Vector Floating-Point (VFP) L2 Cache Control Registers	Lazy switch

A. CPU virtualization

For each virtual machine, Mini-NOVA instantiates a specific data structure that holds in kernel memory the states of hardware resources that are used by the virtual machine. This structure acts as a virtual CPU (vCPU). A vCPU includes the registers of necessary resources to build up a virtual environment. Table I shows the hardware resources involved in a vCPU, which are divided into two privilege levels.

Mini-NOVA permits the frequently-accessed resources to be directly programmed by the virtual machine, except for the hardware states that may affect the microkernel or other virtual machines. For example, interrupt status registers can only be accessed by the privileged code of the microkernel to prevent malicious users disabling interrupts and monopolizing the CPU. While switching between virtual machines, Mini-NOVA saves the current virtual machine's vCPU state and restores its successor's state. To reduce the switch overhead, the vector floating-point (VFP) coprocessor and cache control registers use the lazy switching, meaning that their contexts are switched passively, instead of actively at every virtual machine switch. The reason is that they are relatively less frequently accessed and quite expensive to save.

To host the vCPU content and organize the virtual machine capabilities in the kernel domain, a kernel object Protection Domain (PD) is applied. A Protection domain acts as a resource container and a capability interface between a virtual machine and the microkernel. It holds the state of a virtual machine (the ID number, the priority level, etc). To handle sensitive operations in the virtual machine domain, PD includes an exception interface, which receives exceptions and hypercalls, and distributes them to different capability portals according to the exception's type. Normally, hypercalls are used to replace frequently-used sensitive operations in order to avoid frequent traps

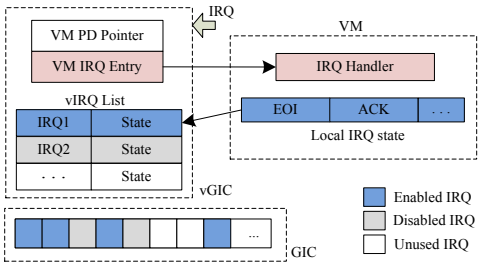

Figure 2. Structure of Virtual Generic Interrupt Controller

during virtual machines' execution. Such operations include: (1) general cache/TLB operations, (2) IRQ operations, (3) memory management: mapping inserting, guest page table creation, etc, (4) access to privileged registers, (5) access to shared devices, such as DMA, FPGA, I/O device, etc, (6) VM inter-communication.

B. Virtual Interrupt

In the ARM architecture, all physical interrupts are managed by the generic interrupt controller (GIC), which receives different types of hardware interrupt resources and generate IRQs to the CPU. For each virtual machine, we create a virtual GIC object. IRQs are trapped to the kernel, and then distributed by the virtual GIC (vGIC) of the current virtual machine. Fig. 2 shows the structure of a vGIC. A vGIC is associated to its host virtual machine's protected domain and keeps a record list of the states of interrupts which the virtual machine is using. In this list, each entry corresponds to the IRQ source number, recording the state of this virtual IRQ. The virtual state of IRQ is programmed locally in the virtual machine, whose values are read back to vGIC when exiting the virtual machine.

On each virtual machine switch, the kernel configures the GIC to mask the interrupts of the previous virtual machine, according to its virtual IRQ list, and unmasks the successor's interrupts (enabled IRQs only). As in Fig.2, the entry address of the virtual machine's IRQ handler is registered in vGIC. Every time a hardware interrupt is received, the system switches to the kernel space to distribute the interrupt. Mini-NOVA writes an End of Interrupt (EOI) value to the GIC interface, then uses the vGIC to inject virtual IRQ into virtual machine by forcing the virtual machine to jump to its IRQ entry and passing the IRQ number to it. Note that because the vGIC only distributes IRQs, it is the guest OS' responsibility to manage its own vIRQ state, for example, if the virtual IRQ has been properly handled.

C. Memory management

In virtualized systems, memory management involves the access control to different privileges: host, guest kernel and guest user. In the ARM architecture, the virtual memory system is controlled by the memory management unit (MMU). The MMU applies a 2-stage page table system for virtual addresses translation. Each page table entry can be marked with the desired access permission ((1) accessible only with a privilege level; (2) full access; (3) no access.). The MMU checks access permission to these pages and generates a page fault exception if the access is not permitted. Obviously, the 2-level privileges are both used for the Mini-NOVA kernel and for the virtual machines respectively.

TABLE II. ACCESS CONTROL IN GUEST USER (GU), GUEST KERNEL (GK) AND HOST KERNEL (HK)

Domain	Permission	DACR (GU)	DACR (GK)	DACR (HK)
Guest user	Full Access	client	client	client
Guest kernel		NA	client	client
Microkernel	Privileged	client		

However, we cannot rely on the privileged access to separate guest kernels and the user space, since both guest kernels and users are executing in ARM's non-privilege mode. In this case, we use the domain access control register (DACR) in MMU, which is divided into 16 sections. Each section can be set to three types: no access (NA), client and manager, meaning forbidden access, permission checked access and check-free access respectively. Guest kernel and guest user are associated to different DACR sections. The guest kernel's DACR section content is set as a client when running in guest kernel space, and changed to NA when running in the guest user space, so that guest kernel spaces are protected from the guest users. Mini-NOVA switches the DACR's value as the guest privilege level changes. Table II shows this mechanism.

Mini-NOVA maintains isolation among VMs' memory spaces. Each VM has its own page table, which only maps the memory space that Mini-NOVA allocates to it. A VM is not allowed to access the microkernel or other guests, otherwise a permission-denied error will occur. Mini-NOVA switches memory space by loading the base address of the page table to the translation table base register (TTBR).

The cache consistency is protected from the switch of memory space, because in Cortex-A9 architecture both instruction and data caches are physically-tagged. Thus, without the duplicate mapping that shares the physical memory, the memory space switch is spared of the expensive cache flush. We utilize the address space identifier (ASID) to simplify the management of TLB. Translations with different ASIDs are respectively labeled in TLB. Each VM is associated with one unique ASID value. The microkernel reloads the ASID register whenever a virtual machine is switched.

D. Scheduling

Mini-NOVA implements a preemptive priority-based round-robin scheduler. All guest OSes/applications are organized into two execution groups: the run queue and the suspend queue. The run queue is composed of the guest

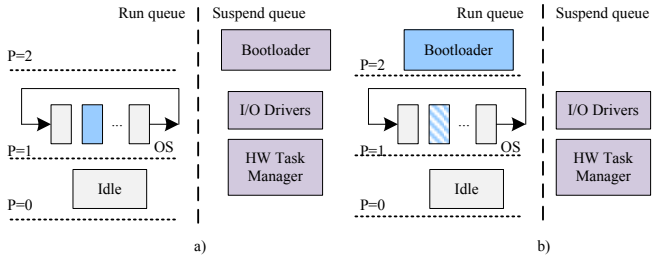

Figure 3. Scheduling Process. a) Guest OSes round-robin sharing CPU. b) User service bootloader is added to the run queue and preempts lower-priority VMs

OSes/applications which are ready to execute. The suspend queue, on the other hand, contains the ones that are not necessarily schedulable to avoid wasting the CPU resource. By default, some user service applications of Mini-NOVA are in the suspend queue because they are only invoked when necessary. Each virtual machine is created with a fixed priority level. The applications which have a tighter time constraint or real-time characteristics are given higher priority level, so that they can preempt general-purpose guest OSes and execute immediately. The VM's priority is held in its protected domain. When invoked, the scheduler selects the highest-priority PD in the run queue and dispatches the vCPU attached to it.

Fig. 3 shows the scheduling process of Mini-NOVA. In the run queue, VMs at the same priority level are organized in double-link circles. As in Fig. 8, all guest OSes are linked in circles at priority level 1, and scheduled in round-robin by default. Normally guest OSes are sharing the CPU equally. The same time quantum is provided to guest OSes. Once activated, a guest OS can run until its time quantum is consumed, or until it is preempted by a higher priority virtual machine. At the preemption point, the microkernel saves the remaining time quantum of the interrupted virtual machine. When this VM is resumed, its time quantum is also resumed so that its total execution time slice is constant.

IV. DPR Technology Support

Mini-NOVA is based on the Zynq-7000 platform, which includes a dual-core ARM processor and a FPGA logic programmable cell allowing DPR. Zynq-7000 is the first and most commonly used SoC platform that integrates ARM with DPR FPGA device, and offers dedicated features and peripherals to facilitate the cooperation of CPU and FPGA logic.

Being designed to manage the coexistence of software and hardware computing resources, it is necessary for Mini-NOVA to provide essential support to the DPR hardware tasks. One major challenge for this approach is to efficiently coordinate the resources of hardware tasks and the separate virtual machines. Security is another major concern because the hardware tasks are shared among guests, and therefore, attention should be paid to protect the guest OS from

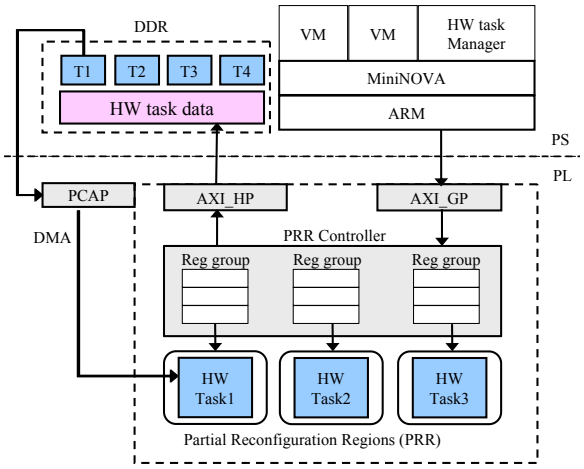

Figure 4. Block Diagram of the proposed PS/PL Architecture

a malicious access of hardware tasks. In this section we introduce Mini-NOVA's support to deal with hardware tasks.

A. Programmable Logic Overview

Fig.4 depicts the cooperation of the processing system (PS) and the programmable logic (PL). While the software system is running within PS, PL is used to host DPR hardware tasks. PL is composed of a Xilinx 7-series FPGA fabric and the interfaces attached to it. As shown in Fig.4, this fabric is divided into static logic and multiple partially reconfiguration regions (PRR). Reconfigurable accelerator modules, or hardware tasks in this paper, are hosted and run in separate PRRs. Each PRR is run-time reconfigurable by downloading configuration data through the Processor Configuration Access Port (PCAP). Static logic is pre-defined and configured at the initialization. It remains unchanged during the system's processing. The main part of static logic consists of the PRR controller, which is in charge of the hardware tasks' execution states, the DMA access and events synchronization. On the PS point of view, the on-chip memory stores the hardware tasks and the data to be processed. On top of Mini-NOVA, a specific microkernel service called the Hardware Task Manager runs in parallel with the guest OSes. Reconfiguration and allocation of hardware tasks are separated from guest users and are handled by the Hardware Task Manager service.

Between PS and PL, different types of AXI interfaces are employed. The General-purpose (AXI_GP) port offers the universally-addressed access of PL. Through the GP port, registers of different PRRs are mapped to different physical addresses in PS, which can be directly accessed by software. Therefore, the GP port is used as a main method to configure and control hardware tasks. The High performance (AXI_HP) port is a buffered AXI high performance interface, and is used by hardware tasks to access and exchange data directly with on-chip memory at high speed. Note that, the AXI Accelerator Coherency Port (AXI_ACP) is also available to provide cache-coherent access from the PL to

ARM. However, since there is only one AXI_HP port and its usage may starve accesses from other AXI masters [18], it is inappropriate and thus aborted in our system, where the AXI_ACP access interferes other simultaneous tasks.

B. Hardware Tasks Organization

The configuration information of hardware tasks is stored in memory as bitstreams files (.bit). Mini-NOVA exclusively maps these .bit files to the memory space of the Hardware Task Manager, which is separated from other VMs. Each hardware task is associated to one or several PRRs that are predefined containers, managed by the PRR controller block. As shown in Fig. 4, the PRR controller provides each PRR with a group of registers, that configures and controls the behavior of the hardware task that is located inside the region. Each PRR's register group is mapped into the universal physical address space and software can directly program the task's configurations by accessing its PRR's physical address.

On the other hand, hardware tasks are allowed to launch DMA access to the on-chip memory to fetch data and put the processed data back, through the AXI_HP port. Guest OSes can open up a special memory section for the hardware task to exchange data. This section is shown in Fig.4 and denoted as the hardware task data section. Note that each guest OS can define its own hardware task data section within its own memory space.

Hardware tasks are organized by the Hardware Task Manager in a look-up table that is indexed with unique ID numbers. For each task, the address and size of its .bit file, the reconfiguration latency and the list of predefined PRRs are stored. When a specific hardware task is required by a virtual machine, the Hardware Task Manager reads its PRR list and selects the appropriate PRR to host the target hardware task. Then the caller guest is allowed to use this hardware task as its client.

C. Security mechanism

The application of hardware tasks brings up new security issues. As in classic virtualization systems, the separate execution environment relies on the MMU, which automatically controls accesses from different privilege levels and blocks illegal access. Separating the memory spaces are ensured by switching the page tables. In the Zynq-7000 platform, however, the FPGA accesses directly the physical memory space, without using the MMU. This makes it impossible to monitor and control the hardware tasks' access via the page table access permission. In this case, an extra security mechanism must be introduced, following two principles: first, one hardware task can be shared by any VM, but should be exclusively used once it is dispatched to a specific guest OS. Second, the hardware task should only access the hardware task data section of the VM which is currently using it, and accessing a memory space outside the specific section is forbidden.

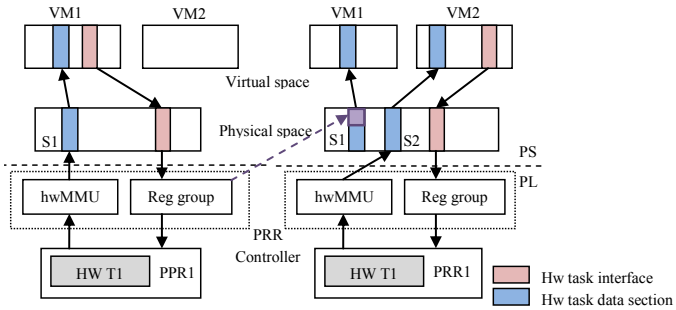

Figure 5. Hardware task allocation between VM1/VM2

As a solution, the security mechanism is shown in Fig.5. To deal with the first consideration, we manage the VM's access to hardware task via updating the page tables. As previously described, the hardware task's behavior is controlled by its PRR register group. To exclusively allocate one hardware task, the PRR register group should only be mapped to its current VM client as the hardware task interface. During the creation of PRRs, we configure each PRR register group to be mapped to the edge of separate physical small-size pages (4KB), so that each PRR can be mapped to a virtual 4KB page independently in a 2-stage page table. As shown in Fig.5, when a hardware task is allocated to one VM client, the corresponding page table is updated to map the hardware task interface, so that the hardware task can be controlled by its client and remain invisible to other guests.

Focusing on the second consideration, we apply a custom component which is called the hardware memory management unit (hwMMU) to control the FPGA's access to the system memory. hwMMU is implemented in the PRR controller and is in charge of monitoring any access to the PS side. When a hardware task is allocated to one VM, the hwMMU is loaded with the physical address of the VM's hardware task data section. So, any access from this hardware task is checked by the hwMMU, which forbids the access outside the determined section. Following this mechanism, the rest of the memory space is protected from the hardware tasks.

As shown in Fig.5, when a hardware task is not used by a VM anymore, it can be dispatched to another VM. In this case, the page table of the former VM should be updated to eliminate the mapping of the hardware task, and the hwMMU should also be reloaded with the successor's address. This guarantees that a hardware task can only be accessed by no more than one VM at a time.

Besides the access security, the consistency of hardware tasks should also be considered. While PRR can be detached from one client and claimed by another when necessary, VMs should be acknowledged whether its hardware task is consistent or not. In the hardware task data section, we

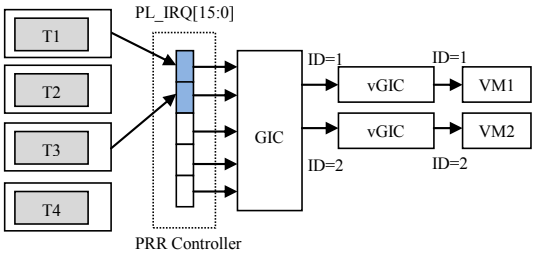

Figure 6. PL interrupt management

allocate a reserved data structure to hold the state of a hardware task, the state flag and the hardware task interface registers. As in Fig.5, when T1 is reclaimed to VM2, the register group content of T1 is saved to the VM1 hardware task data section, with a state flag indicating to VM1 that T1 has been used by other clients.

D. Interrupt Organization

Normally, hardware and software tasks are executing simultaneously, and their synchronization can be achieved by checking the status of hardware tasks, or by allowing the hardware task to generate IRQs to acknowledge certain events. In our system, FPGA IRQ sources are reserved for the PL side, and are governed by the PRR controller. The interrupt sources (PL_IRQ) are organized by the General Interrupt Controller, and support up to 16 different IRQ sources generated from the FPGA side. To efficiently manage the IRQs, the PRR controller needs to allocate those resources to hardware tasks and also to register them in the corresponding VM's vGIC table. When a VM requires an IRQ from its hardware task, the Hardware Task Manager asks the PRR controller to allocate an available IRQ source to the hardware task, and updates the VM's vGIC table to register the IRQ source.

The process of handling PL interrupts is shown in Fig.6. Once a PL interrupt is registered in vGIC, the VM is ready to receive hardware task IRQs. If the IRQ occurs during the VM's execution, then it is handled immediately. If the IRQ occurs when the VM is not active, then the IRQ state remains the same until the next time the VM is scheduled and the IRQ is properly processed.

Besides, the hardware task management also involves the PCAP completion IRQ, which notifies the completion of the bitstream download of the PCAP interface. This signal can be used to acknowledge the VM that the target hardware task is successfully configured and ready to be used. The PCAP interrupt is always connected to the VM which launches the current transfer. The related VM can be configured to receive the PCAP interrupt if required. Such an usage is explained in the next section.

E. Implementation

The allocation mechanism of hardware tasks is implemented in the Hardware Task Manager service. This service runs in an independent memory space and has authority to dispatch and reconfigure hardware tasks. Whenever a virtual machine requires hardware tasks, it passes its request to the service via an hypercall. Then the Hardware Task Manager service is scheduled to properly handle the requests. Normally, we consider the hardware task request as a tighter timing constrained demand that should be given a response as soon as possible. In this case, the Hardware Task Manager service is created with a higher priority level than general guests, so that this service can preempt guests and execute immediately once it is invoked by the microkernel. After processing the request, the manager service will remove itself from the running queue list, resuming the interrupted guest OS with a return status.

Guest OSes invoke a special hypercall to require hardware task resources. Three arguments are passed via this hypercall: the target hardware task ID number, the virtual address of the task interface, and the virtual address of the hardware task data section. To successfully dispatch the target task, the Hardware Task Manager needs to implement the task into the PRR and properly set the mapping. In the Manager service domain, a PRR table is built to record the states of the PRRs. Its contents include the PRR's current client, the hardware task, the execution state (idle or busy), etc. This table cooperates with the hardware task table to rationally allocate tasks. In Fig. 7, an example of the Hardware Task Manager's processing is demonstrated in six stages:

(1) Because the guest process P1 requires the hardware task T1, the guest OS generates a hypercall to invoke the Hradware Task Manager service.

(2) Receiving the VM's request for hardware task T1, the Manager service first allocates an appropriate PRR for T1, by checking the states of T1's suitable PRRs. PRR1 is chosen to implement T1 since it is currently in Idle state. Note that if no idle PRR is available, the manager service would return to the applicant guest OS with a Busy status.

(3) To make PRR1 accessible to the guest OS, the Manager service updates the guest OS' page table by mapping the PRR hardware task interface to the desired virtual address space.

(4) A new hardware task data section address is loaded to the hwMMU to control T1's access to the memory space.

(5) If T1 is currently not implemented in PRR1, then the manager would launch a PCAP transfer to download the T1 bitstream into PRR1.

(6) The Manager Service returns to the calling guest OS with a status value. If a PCAP reconfiguration is made, a reconfig. flag is returned, otherwise a success flag is returned. Note that to overlap the significant reconfiguration overhead, the manager service does not check the completion of the PCAP transfer. The guest OS may choose to acknowledge the PCAP completion by two methods: by polling the completion signal or by receiving the PCAP completion IRQ.

Note that, as described in Section 4.3, if PRR1 was

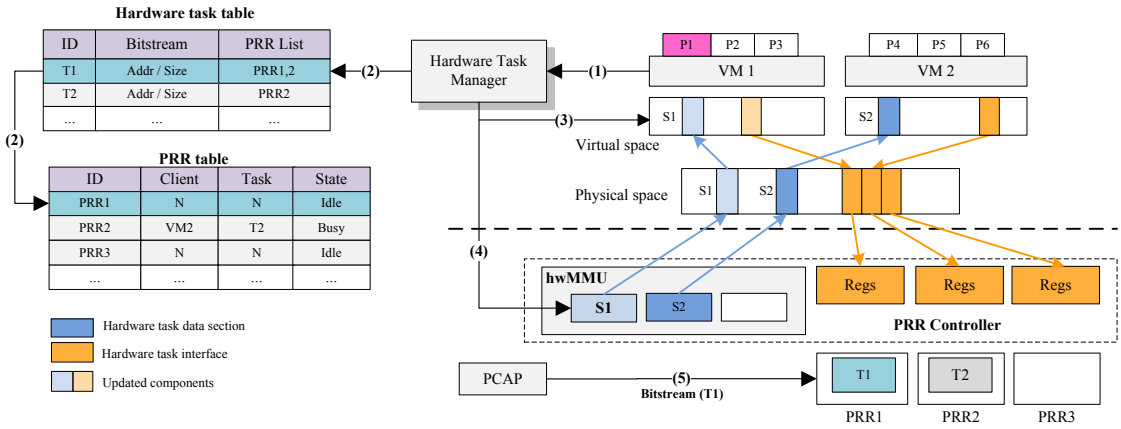

Figure 7. Hardware Task Manager Processing Routine

previously used by other clients, like VM2, then consistency must have been maintained in stage (3) and (4). The VM2's page table must be updated to demap the PRR1 interface section. The interface register contents should be saved into VM2's hardware task data section, and the state flag should be marked as inconsistent. The inconsistency state can be acknowledged by two methods. First, VM2 can automatically check the state flag in hardware task data section whenever it uses the T1. Second, if T1 is demapped from VM2's memory space, then any access to its interface is trapped in a page fault exception and handled by the guest OS' interrupt service.

V. Verification & Evaluation

In this section, we present the methodology and the experimental results of the Mini-NOVA microkernel. The measurements were obtained on the 660MHz ARM Cortex-A9 processor, along with 32KB separate L1 instruction and data cache, 512KB L2 unified cache, 512MB DDR memory, and external 4GB SD card memory, on the Zynq-7000 all programmable SoC. We verified the functionality of Mini-NOVA, especially the performance of the DPR management by executing on-time requests and allocations of hardware tasks. In order to demonstrate the virtualization overhead, we also measured and compared the performance when executing the application in the VM and directly on the native hardware.

A. RTOS implementation

Focusing on the real-time applications in the communication domain, we found that the real-time OS is more interesting to our system. Compared to the general purpose OS like Linux, the RTOS can be virtualized with less modification and lower virtualization latency. It is also an appropriate object to verify and evaluate our system's performance in the real-time domain. In this paper, we paravirtualized uCOS-II real time kernel as a guest OS. Since uCOS-II was originally executing in the supervisor mode and cannot be directly hosted, several modifications have been made to the initial OS code:

- The boot up sequence of uCOS was modified to execute in the user mode. The configuration of privileged system registers should be performed by the microkernel, since uCOS-II has no longer the full authority. During the execution, all sensitive operations were manually replaced by hypercalls.
- The guest timer is implemented by a virtual timer allocated by Mini-NOVA. Thus the original timer initialization is modified to register the virtual timer state to the microkernel, which then configures the virtual timer according to this timer state.
- The interrupt handling process is changed. A local table is built to record the virtual IRQs states. uCOS-II can only access to the local table to handle the interrupts.
- The utilization of shared I/O devices, such as UART and SD card, were added with the microkernel's supervision.
- Functionalities supporting hardware task access were added as application program interfaces (API), which facilitate the guest uCOS-II to require and utilize the hardware tasks.
- Mini-NOVA provides dedicated hypercalls (a total number of 17) for the guest uCOS-II to fulfill its sensitive operations, such as cache flush, page table management, and inter-VM communication.

To minimize the modification to the original source code, all paravirtualization porting codes are organized as a patch package, including additional functions and hypercalls. The size of patch counts to around 200 lines of code (LOC). The uCOS-II source code were rewritten by only replacing the involved functions and having them re-implemented in the porting patch, therefore were slightly modified.

B. Performance Evaluation

Mini-NOVA follows a lightweight implementation, with a small footprint of 20MB. The complexity of Mini-NOVA

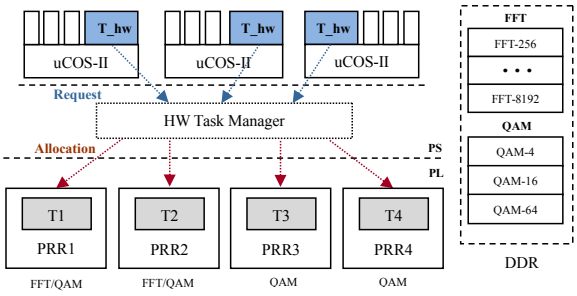

Figure 8. Diagram of Performance Evaluation

TABLE III. OVERHEAD OF HARDWARE TASK MANAGEMENT (US)

Guest OS number	Native	1	2	3	4
HW Manager entry	0	0.87	1.11	1.26	1.29
HW Manager exit	0	0.72	0.91	0.96	0.99
PL IRQ entry	0	0.23	0.46	0.50	0.51
HW Manager execution	15.01	15.46	15.83	16.11	16.31
Total overhead	15.01	17.06	17.84	18.33	18.57

is 5,363 lines of code (LOC) for all kernel code and user services, including the Zynq platform-specific resources, and compiles to about 40KB size as Executable and Linkable Format executable (ELF). A total number of 25 hypercalls are provided to paravirtualized operating systems.

To make our evaluation more realistic and convincing, we selected a series of communication and data processing specific software/hardware tasks. The hardware tasks are computation-intensive IP cores such as FFT and QAM modules. We ran multiple guest VMs. Each VM is assigned with a virtualized uC/OS-II, which is executing heavy workload tasks, for example, GSM encoding, or Adaptive differential pulse-code modulation (ADPCM) compression. Mini-NOVA provides each guest OS with a time slice of 33 ms. Since guest OSes equally share the CPU, when the time slice of a guest is over, the next guest OS is scheduled. Fig. 8 shows the performance evaluation. Two hardware task sets, FFT (ranging from 256 points to 8192 points) and QAM (4, 16, and 64 constellation sizes), are provided. They are stored in the DDR memory as .bit files. Four PRRs were defined in the FPGA fabric to host the hardware tasks. PRRs are allocated with different FPGA resources. Since FFT blocks are quite large, only PRR1 and PRR2 are large enough to contain the FFT tasks. The size and reconfiguration delay of these tasks are directly related and were described in [17]. On the other hand, QAM modules have a small size and can be hosted in all four PRRs. Each guest OS is running multiple tasks, and particularly a special task (T_hw) programmed to invoke hardware task requests. This task is used to evaluate the overhead of the Mini-NOVA's hardware task management. Each time it executes, it randomly selects a hardware task from the hardware task set and generates a hardware task hypercall for this task. After a sufficient number of iterations, the average execution time can be calculated.

Table III presents the overhead of the Mini-NOVA hardware task management according to the guest OS number. The native execution is measured by implementing the uCOS-II natively on the ARM processor, and implementing the hardware task management service as a uCOS-II function. Measurement results with different number of guest OSes are listed to demonstrate how the performance is influenced according to the number of parallel VMs. Typically, the HW Manager entry/exit measurements are the cost of entry and exit of Hardware Task Manager when VM's request. Memory space switches are involved in this process. The PL IRQ entry latency refers to the cost of distributing an hardware task IRQ event to its VM client. This process begins from the exception vector table and ends when the vGIC injects the virtual interrupt to the VM. The HW manager execution overhead measures the average execution time of the Hardware Task Manager to handle the requests. The overall response delay is also listed, which is the sum of overheads from the Hardware Task Manager's entry to its exit.

Table III native execution represents the shortest latency since the hardware manager works as a system function and thus can be directly dispatched without any latency. The IRQ is directly triggered to uCOS-II. Besides, in native uCOS-II, the hardware task manager service does not need to update the page tables since all tasks execute in a unified memory space. The virtualization evaluation starts from one guest OS to four OSes. Generally, as the number of OSes increases, the performance is degraded, which is mainly due to an increase of miss rate of cache and TLB table, and the complexity to allocate hardware tasks.

The influence of cache/TLB can be clearly seen in the overheads of HW Manager entry/exit and PL IRQ entry. As the number of VM increases, the cache and TLB are updated more frequently. Thus, the related cache and TLB list of the Hardware Task Manager hypercall and entry code can be easily flushed when multiple OSes exist. This is why the overhead of the HW Manager entry grows significantly with the OS number. On the other hand, the HW Manager exit latency is changed much less, since it always executes right after the Hardware Task Manager's execution, and thus the cache/TLB is only modified during the manager's process, which results in a higher hit rate. With the same reason, the PL IRQ entry latency is also less influenced by the increasing OS number since the IRQ handling process is frequently called during execution and less affected by the cache/TLB miss. For example, each timer tick is handled as a IRQ, thus the IRQ related content is often cached, or recorded by the TLB, then causing a higher hit rate.

On the other hand, the HW Manger execution cost's growth is caused mainly by an increased allocation complexity. In virtualized systems, the manager needs to change the guest OS' mapping to guarantee the security. This operation is performed by switching to the kernel space to update the target VM's page table, introducing extra hypercalls.

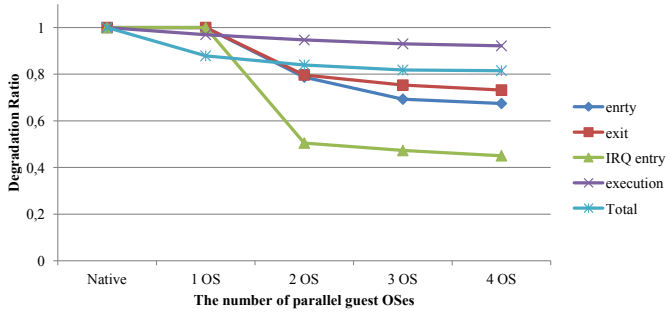

Figure 9. Performance degradation ratio of Hardware Task Manager

Second, as more guest OSes are sharing the hardware tasks, the manager should switch the task's client more frequently, which involves declaiming the task from the previous VM, and reclaiming it to the new one. Also, more PCAP transfers are required to download different hardware tasks.

To evaluate the influence of performance caused by virtualization, we introduce the degradation ratio R_D:

$$R_D = \frac{t_{Virtualization}}{t_{Native}} \quad (1)$$

, where t_{Native} is the overhead measured when OS running on native machine, and $t_{Virtualization}$ is the overhead with virtualization implementation. Fig. 9 gives the degradation ratios of the characteristic overheads of Hardware Task Manager listed in Table III, with different number of parallel virtual machines. For HW Manager entry/exit and PL IRQ entry overheads, which are measured as zero when running natively, the performances with one virtual machine are used instead of t_{Native} in (1), to present the tendency of overhead along with increasing virtual machines. As Fig. 9 shows, it is obvious that the ratios are declining with the OS number, while the trend is slowing down, indicating that the system is getting a constant overhead, when the worst case is approached. Compared to the total execution time, the Mini-NOVA virtualization brings acceptable limited impact to the overall performance of the hardware task management, and is a suitable solution to support the dynamic partial reconfiguration technology in the virtualization domain. This is due to the lightweight implementation of Mini-NOVA.

VI. Conclusion

In this paper, we have proposed the Mini-NOVA microkernel, which is designed to provide a virtualization approach for the ARM-FPGA platform. Mini-NOVA is built to host paravirtualized operating systems with lower complexity and has the ability to dispatch hardware tasks to virtual machines by supporting the dynamic partial reconfiguration technology. Our evaluations have demonstrated that the hardware tasks are efficiently managed with a short response latency, and that the virtualization overhead only has a slight impact on the overall performance.

References

[1] G. Heiser, "The role of virtualization in embedded systems," in *IIES*. ACM, 2008, pp. 11–16.

[2] M. Liu, M. Crussiere, and J.-F. Helard, "A novel data-aided channel estimation with reduced complexity for tds-ofdm systems," *Broadcasting, IEEE Transactions on*, vol. 58, no. 2, pp. 247–260, 2012.

[3] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Eurosys*. ACM, 2010, pp. 209–222.

[4] J. Liedtke, *On micro-kernel construction*. ACM, 1995, vol. 29, no. 5.

[5] M. Aichouch, J. C. Prevotet, and F. Nouvel, "Evaluation of the overheads and latencies of a virtualized rtos," in *SIES*. IEEE, 2013, pp. 81–84.

[6] E. Lubbers and M. Platzner, "Reconos: An rtos supporting hard-and software threads," in *FPL*. IEEE, 2007, pp. 441–446.

[7] D. Gohringer, M. Hubner, E. N. Zeutebouo, and J. Becker, "Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in *IPDPSW*. IEEE, 2010, pp. 1–8.

[8] 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.

[9] G. Heiser and B. Leslie, "The okl4 microvisor: Convergence point of microkernels and hypervisors," in *APSYS*. ACM, 2010, pp. 19–24.

[10] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park *et al.*, "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones," in *CCNC*. IEEE, 2008, pp. 257–261.

[11] J. H. Ding, C. J. Lin, P. H. Chang, C. H. Tsang, W. C. Hsu, and Y. C. Chung, "Armvisor: System virtualization for arm," in *OLS*, 2012, pp. 93–107.

[12] P. Varanasi and G. Heiser, "Hardware-supported virtualization on arm," in *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 2011, p. 11.

[13] C. T. Liu, K. C. Chen, and C. H. Chen, "Casl hypervisor and its virtualization platform," in *ISCAS*. IEEE, 2013, pp. 1224–1227.

[14] C. Dall and J. Nieh, "Kvm/arm: The design and implementation of the linux arm hypervisor," in *ASPLOS*. ACM, 2014, pp. 333–348.

[15] W. Wang, M. Bolic, and J. Parri, "pvfpga: accessing an fpga-based hardware accelerator in a paravirtualized environment," in *CODES+ ISSS*. IEEE, 2013, pp. 1–9.

[16] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *FCCM*. IEEE, 2014, pp. 109–116.

[17] T. Xia, J. C. Prevotet, and F. Nouvel, "Microkernel dedicated for dynamic partial reconfiguration on arm-fpga platform," *SIGBED Rev.(EWiLi 2014)*, vol. 11, no. 4, pp. 31–36, 2015.

[18] *Ug585: Zynq-7000 all programmable soc technical reference manual*, Xilinx Inc., 2013.