IO Virtualisation in a Partitioned System

M. Masmano, S. Peiró, J. Sánchez , J. Simó, A. Crespo
Instituto de Automatica e Informatica Industrial
Universidad Politecnica de Valencia, Spain
{mmasmano,speiro,jsanchez,jsimo,acrespo}@ai2.upv.es

Abstract

Partitioned systems permit to isolate in partitions several applications with different security levels and/or criticality. Hypervisor technology provides virtual machines to execute partitions under two basic principles: space and time isolation. This view is complemented with the ”dedicated devices” technique that assigns devices exclusively to a partition. However in case of shared devices a partition has to provide a device or IO virtualisation to the other partitions, referred as the ”I/O Server” approach. We present a solution for device virtualisation on the XtratuM hypervisor which has been specifically designed for critical embedded systems. The approach is in the scope of the Open Secure Vehicular Platform project. Such system will support different types of partitions, from real time constrained to non-trusted user partitions running general purpose operating systems. This work has been partially suported by Spanish Government COBAMI: DPI2011-28507-C02-02

1. Introduction

Partitioned software architectures can represent the future of secure systems. They have evolved to fulfil security and avionics requirements where predictability is extremely important. The separation kernel proposed in [3] established a combination of hardware and software to allow multiple functions to be performed on a common set of physical resources without interference. It is precisely XtratuM the selected kernel to serve as the base execution environment for the OVERSEE partitioned system.

The Open Vehicular Secure Platform [5] (or OVERSEE, for short) is intended to serve as a single access point to vehicle networks. It will provide a protected, standardized in-vehicle runtime environment and on-board access and communication point. Applications for this platform include, between others, positioning systems, stolen vehicle tracking, traffic information, web browsing, etc. Ultimately, OVERSEE aims to create an open software platform for which everyone can develop applications and download them. Of course, this platform must ensure that the applications cannot harm each other and, especially, internal in-vehicle software applications. The heterogeneous nature of the applications being executed under OVERSEE makes XtratuM system partitioning the best way to isolate and protect applications from each other. Finally, for this system, the chosen target hardware is the Intel Atom architecture, which is the Intel architecture for embedded systems.

XtratuM ensures temporal and spatial isolation of the applications that run over it. This applications may vary from bare-C applications to entire operating systems as, for example, Linux. Each of this applications is referred to as a *partition* or *guest*. On this paper, we will pay special attention to Linux guests. The use of Linux as a guest to run general purpose applications is very interesting; thanks to the development efforts made by the open source community, Linux offers a vast variety of device drivers, as well as many other software mechanisms (i.e. a TCP/IP stack), which reduce implementation burden of software projects.

With the expansion of hypervisor technology, Linux has begun to host extensions for system virtualisation. Linux kernel supports at least 8 distinct virtualisation systems, being Xen [2], KVM [4] or Lguest [7] among them, without taking into account the relatively new XtratuM hypervisor. With such large variety of virtualisation systems, a new standard has appeared to fulfil the needs of device virtualisation, which until now had to be implemented by each hypervisor. This standard is known as Virtio [8]: “a series of efficient well-maintained Linux drivers which can be adapted for various hypervisor implementations”. As will be explained on section , the special features of XtratuM needs Virtio to be modified in a different way than other virtualisation solutions.

Section 2 outlines the design and requirements of the OVERSEE platform. Section 3 presents the architecture and main design criteria of the XtratuM hypervisor. Section 4 introduces the para-virtualisation of the Linux operating system to run on the XtratuM hypervisor. Section 5 presents the approach of the Virtio to achieve IO device virtualisation. Section 6 analyses the Virtio design assumptions and how they are met on the XtratuM hypervisor. Finally some conclusions are enumerated.

2. OVERSEE Platform Overview

The figure 1 presents the architecture of OVERSEE. Starting at the bottom are the devices provided by the hardware platform, running on top of the hardware is the virtualisation layer provided by the XtratuM hypervisor, the XtratuM hypervisor isolates each of the partitions into its own virtual machine. Special focus is placed on the system partitions, these system partitions are an integral part of OVERSEE in charge of offering generic services and facilities for the applications running within the configured partitions and clusters.

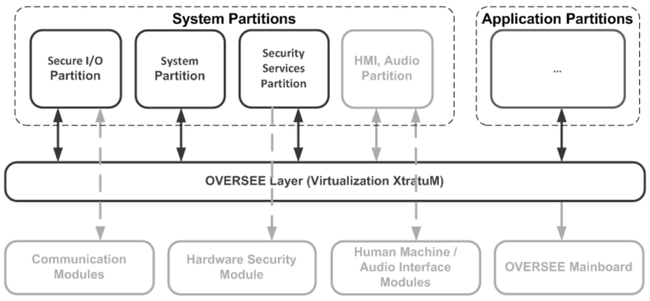

Figure 1. OVERSEE platform architecture.

The system partitions shown in figure 1 reflect the design decision to place additional platform functionality (e.g., communication management and security services) in additional partitions instead of integrating them into the virtualisation subsystem. The main reasons for this decision are:

- The code size of the virtualisation subsystem should be kept very small due to the needed efficiency of frequent context switches.
- The dependability of the platform and the security of the platform will be improved since the device drivers and management components are running in an isolated partition.
- Further functionality like additional communication services could easily be added to the OVERSEE platform since this only means to modify the concerned system partition while the core virtualisation system will stay unchanged.

2.1. Secure I/O Partition

Within the secure I/O partition most of the drivers for the connectivity modules will be handled. Therefore, this partition together with the internal communication resources of XtratuM is responsible for the provision of communication connections for the partitions. Secure means that the access to the different communication means will be restricted inside the secure I/O partition.

2.2. System Partition

The system partition is responsible for the management of runtime environments within XtratuM, so starting, stopping and monitoring of the application partitions and clusters. Furthermore, the components to provide additional services of the OVERSEE platform (e.g., remote diagnosis) will be placed in this partition.

2.3. HMI, Audio Partition

The management of HMI and Audio devices is no integral part of the OVERSEE platform. However for most of the use cases which are applicable for OVERSEE interaction with the driver or other occupants of the vehicle is required is is implemented as proof of concept.

3. XtratuM Overview

XtratuM [3] is a bare-metal hypervisor with extended capabilities for highly critical real-time systems. The design of XtratuM has tried to apply the philosophy of the ARINC-653 standard [1] (despite not being ARINC-653 compliant). This standard is used to achieve strong isolation between running guests, so it has rigid policies about resource management.

On the time domain, XtratuM allocates CPU to partitions following a *plan* which is defined at configuration time, i.e., it uses a cyclic scheduler. Dynamic schedulers are avoided when systems under control are critical real-time. This ensures a predictable behaviour as well as scheduler robustness against system temporary overloads. On the other hand, this scheme narrows maximum bandwidth of non real-time guests, which may need more relaxed scheduling policies. As explained on section 6, implementation of virtual devices have to take into account this fact in order to optimize throughput.

Besides CPU management, XtratuM isolates spatially the partitions by defining a set of accessible physical memory areas for each one. Also, physical memory areas can be defined as shared, thus being accessible by several guests. This is precisely the chosen mechanism for driver virtualisation.

3.1. XtratuM Architecture

XtratuM is in charge of virtualisation services to partitions. It is executed in supervisor processor mode and virtualises the cpu, memory, interrupts and some specific peripherals. The figure 2 shows the complete system architecture. The internal XtratuM architecture includes: memory management, scheduling (fixed cyclic scheduling), interrupt management, clock and timers management, partition communication management (ARINC 653 communication model), health monitoring and tracing facilities. At the hypervisor level three layers can be identified:

- Hardware-dependent layer: It implements the set of drivers required to manage the strictly necessary hardware: processor, interrupts, hardware clocks, hardware timers, paging, etc. This layer is isolated from the rest through the Hardware Abstraction Layer (HAL). Thus, the HAL hides the complexity of the underlying hardware by offering a high-level abstraction.

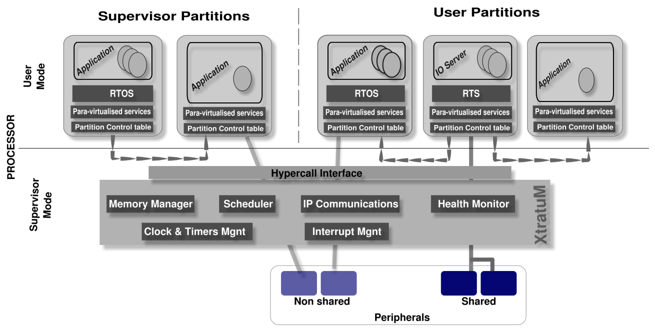

Figure 2. XtratuM architecture.

- Internal-service layer: These services are not available to the partitions. This layer includes a minimal C library which provides the strictly required set of standard C functions (e.g. strcpy, memcpy, sprintf) and a bundle of data structures. The system boot is also part of the internal services.

- Virtualisation-service layer: It provides the services required to support the para-virtualisation services, which are provided via the hypercall mechanism to partitions. Some of these services are also used from other XtratuM modules.

3.2. XtratuM Design Principles

Bare-metal hypervisor technology is the most promising approach to achieve the best performance which is a major criteria to design and implement critical real-time systems. On the other hand, para-virtualisation technique jointly with dedicated devices permits to reduce drastically the code of the virtualisation layer.

In order to design a hypervisor for safety critical systems, the following design criteria have to be considered:

- Strong spatial isolation: the hypervisor is executed in privilege (supervisor) processor mode whereas partitions are executed in user one. Partitions are allocated in independent physical memory addresses. A partition only can access to its memory areas.

- Strong temporal isolation: the hypervisor enforces the temporal isolation by using a fixed cyclic scheduler to execute partitions.

- Partition management: the partitions are executed in user mode, thus guaranteeing that they have not access to processor control registers. The hypervisor defines a set of services that allow system partitions to start, reset, reboot and stop partitions.

- System partitions: some partitions can use *special* services provided by the hypervisor. These services include: partition management, access to system logs, etc.

- Robust communication mechanisms: the partitions are able to communicate with other partitions by using specific services provided by the hypervisor. The basic mechanism provided to the partitions is the port-based communication. The hypervisor implements the link (channel) between two ports or more ports. Two types of ports are provided: sampling a queuing as defined in the ARINC-653 standard [1].

- Interrupt Model: the hypervisor provides an interrupt model to the partitions. Partitions can not interact with native traps. All the interrupts are detected handled by the hypervisor and propagated to the partitions according to the system configuration file (XM_CF).

- Fault management model: faults are detected and handled by the hypervisor. The detection of a fault can be the occurrence of a system trap or the occurrence of an event generated by the hypervisor code. The health monitor module in the hypervisor implements the fault management model.

- Non-preemtable: in order to reduce the design complexity and increase the reliability of the implementation, the hypervisor is designed to be non-preemtable.

- Resource allocation: fine grain hardware resource allocation is specified in the system configuration file (XM_CF). This configuration permits to assign system resources (memory, I/O registers, devices, memory, etc.) to the partitions.

- Minimal entry points: the hypervisor has to clearly identify the execution paths and the entry points.

- Small: The validation and formal verification complexity increases with the number of lines of code. The hypervisor code shall provide the minimum services in order to be as minimal as possible.

- Deterministic hypercalls: All services (hypercalls) shall be deterministic and fast.

3.3. Interrupt Model

Different manufacturers use terms like exceptions, faults, aborts, traps, and interrupts to describe the processor mechanism to receive a signal indicating the need for attention. Also, different authors adopt different terms to their own use. **In order to define the interrupt model, we provide the definition of the terms used in this work**.

A *trap* is the mechanism provided by the processor to implement the asynchronous transfer of control. When a trap occurs, the processor switches to supervisor mode and unconditionally jumps into a predefined handler.

A *software trap* is raised by a processor instruction and it is commonly used to implement the system call mechanism in the operating systems.

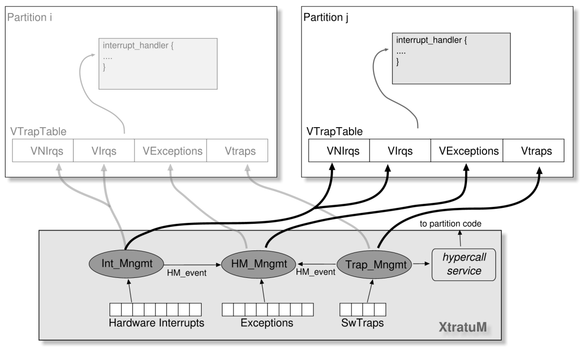

Figure 3. Interrupt Model.

An *exception* is an automatically generated interrupt that occurs in response to some exceptional condition violation. It is raised by the processor to inform about a condition that prevents the continuation of the normal execution sequence. There are basically two kind of exceptions: those caused by the normal operation of the processor and those caused by an abnormal situation (like an memory error).

A *hardware interrupt* is trap raised due to an external hardware event (external to the CPU). These interrupts generally have nothing at all to do with the instructions currently executing and informs the CPU that a device needs some attention.

In a partitioned system, as the one depicted in figure 3 the hypervisor (*XtratuM*) handles these interrupts (*native interrupts*) and generates the appropriated *virtual interrupts* to the partitions (*Partition i, j*). A partition have to deal with the following *virtual interrupts*:

- *virtual traps* are the traps generated by the hypervisor to the partitions as consequence of a *native trap* occurrence.
- *virtual exceptions* are the exceptions propagated by the hypervisor to the partitions as consequence of a *native exception* occurrence. Not all the *native exceptions* are propagated to the partition. For instance, a memory access error that is generated as consequence of a space isolation violation is handled by the hypervisor which can perform a halt partition action or can generate another different virtual exception (like memory isolation fault). On the other hand, a numeric error is propagated directly to the partition. *Virtual exceptions* are a superset of the *native exceptions* which include additional exceptions generated by the hypervisor (virtual processor). Some of them are: memory isolation error, IO isolation error and temporal isolation error.
- *virtual hardware interrupts* are directly generated by the real or the virtual hardware. The real hardware corresponds to external devices (dedicated devices technique) or peripherals and the virtual hardware includes the different virtual devices associated to the virtualisation. Some of these virtual devices are:
  - Virtual hardware and execution clocks
  - Virtual timers based on hardware or execution clocks.
  - New message arrival. The communication mechanism (channel) implemented by XtratuM is seen as a hardware device.
  - Partition slot execution. In a partitioned system the partition is aware of the partition scheduling, this interrupt informs to the partition that a new slot has been scheduled.

Only *virtual hardware interrupts* can be enabled or disabled by partitions.

Four strategies have been used to prevent partitions to jeopardise temporal isolation:

- Partitions have no access to the trap table. Thus, partitions are unable to install their own trap handlers. All traps are directly handled by XtratuM and, when required, propagated to partitions which defines its own *virtual trap table*.
- Partitions cannot interact with native traps. Partitions are executed in user mode, thus guaranteeing that they have not access to control registers.
- A partition can not mask those *virtual hardware interrupts* not allocated to the partition.
- When a partition is scheduled, all the *hardware interrupts* associated to other partitions are disabled. When the partition context switch occurs, the hypervisor detects the hardware interrupts pending for the next partition to be executed and raise them depending on the partition interrupt mask.

3.4. Temporal Isolation

Temporal isolation refers to the system ability to execute several executable entities (threads, processes, partitions, etc.) guaranteeing:

- the timing constraints of the executable entities
- the execution of each entity does not depend on the temporal behaviour of other unrelated entities

The temporal isolation enforcement is achieved at the first scheduling level (partition scheduling). Partitions are scheduled according a static schedule (plan). The plan defines a set of time slots for each partition within a Major Frame (MAF). The MAF is executed in a repetitive way.

XtratuM implements a static (cyclic) scheduling that follows the ARINC 653 specification [1] which defines a general-purpose Application/Executive (APEX) software interface between the operating system and the application software. ARINC 653 defines a cyclic scheduling for the global scheduler and a preemptive fixed priority policy for the local scheduler.

4. Para-virtualisation of Linux Partitions

XtratuM uses the technique of para-virtualisation. Therefore, guests have to be modified (i.e. para-virtualised) in order to not access hardware directly but use the hypercalls services. Here we focus on guest partitions running the Linux operating system, as it provides several advantages like drivers and applications which help to leverage the software implementation burden. Along with the virtualisation technologies, Linux kernel has evolved to offer built-in para-virtualisation mechanisms. This has greatly simplified the task of porting Linux to the XtratuM architecture while enhancing forward compatibility.

4.1. IO Management on a Partitioned System

Intel x86 architecture offers a separate address space for accessing peripherals. This address space is composed by ports and can be mapped so that they appear in the physical memory address space. Partitions can be given permissions to use some of these ports in order to have direct access to some peripheral.

When talking about system input and output (I/O) on partitioned systems, there are two paradigms for device driver access. The first approach is to leave device management to the hypervisor. However, this is not a good approach as, for each device, a driver would have to be implemented at hypervisor level and, also, XtratuM complexity would grow too much. Thus, only a few simple drivers (console, UART) have been implemented inside XtratuM. The second approach is to leave device driver implementation to partitions themselves. This approach is more flexible as we can use device drivers already implemented, like those included in the Linux kernel. For this method to work, guests are given permissions to access some I/O ports on the configuration step.

By leaving devices to partition control, new problems arise, related to device management. If a device on the system is dedicated so that it is mapped to at most one partition, there is no problem. Nevertheless, when there is the need for sharing devices between several guests, special management has to be applied. Operating system device drivers control mutual exclusion of the threads accessing the same device, so that input/output transactions are atomic, thus ensuring a correct operation. However, there are no mutual exclusion mechanisms on XtratuM, as this would break partition isolation (partitions may control the way other partitions are executed).

A proper solution to the device virtualisation problem is to create a separate secure I/O partition with exclusive access to devices. This partition virtualises these devices for each of the user partitions. With these model, the virtual devices on the user partition side will send requests to the I/O partition for accessing real devices. Underneath the virtual devices, a software layer will provide the necessary mechanisms for transporting the requests, as the partitions are spatially isolated. Figures 4 and 6 depict the model with different detail level.

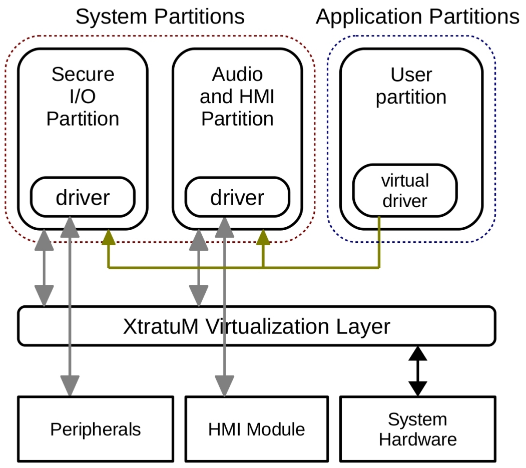

Figure 4. I/O virtualiation model.

5. Virtio Overview

The VIRTIO specification [9] provides an stable and efficient mechanism for device virtualisation. Originally it was designed for providing virtual device I/O to Linux guests running hosted under virtual environments, where the hypervisor (host) also happens to be a Linux system.

The Virtio design is logically composed of the three parts that are required in order to provide a device virtualisation solution; that is: the device model, the driver model and the transport mechanism used to glue devices and drivers together:

1. The Virtio device model in 5.1 is provided by the I/O partition, is in charge of offering a suitable device abstraction closely resembling a typical hardware device.
2. The Virtio device drivers in 5.2 is used by the Linux guest partitions are in charge of managing and accessing the Virtio devices offered by the I/O partition.
3. The Virtio transport mechanism in 5.3 is in charge of providing an efficient mechanism for connecting both the Virtio devices and drivers.

5.1. Virtio Devices

The Virtio device model is designed with PCI [6] devices virtualisation in mind, that is Virtio devices are very similar in several aspects to PCI devices. This similarity can be further explained by examining the typical operations performed on devices:

1. **Device configuration:** is performed on a ”configuration memory space” associated to a Virtio device that contains: IRQ, Status, Device features and Data descriptors information, as it would be found on a PCI device.

2. **Device activity notification:** is performed using the extended IRQ (Interrupts Requests) mechanism provided by the hypervisor, used by the Virtio device to requests attention to the guest.

3. **Device operations:** common device operations as device data transfers are performed on buffers allocated by the guest and provided to the device (I/O Partition) which resembles programming of DMA data transfers.

5.2. Virtio Drivers

The following Virtio drivers are supported on XtratuM which are available on the guest partitions:

- The virtio_net driver implements a virtual network device that provides TCP/IP communication to the guest partitions. virtio_net provides a virtual Ethernet network interface card that provides guest partitions point to point communication with the host partition. This can be used to perform NAT/filtering to provide the guests access to Internet.

  The chosen architecture for the virtual network can be seen on figure 5. Each of the guests has its own *eth* interface, which is virtually connected to *vnetX*, a virtual network interface on the host side. The I/O partition will act as a router, taking packets from the virtual networks and possibly routing them to other guests or the outside world through the real network.

- The virtio_block driver implements a virtual block device that provides storage to the guest partitions. virtio_block allows the guest partitions to have a virtual storage device where a standard Linux distribution can be installed and used as the root file-system.

- The virtio_console driver implements a virtual console device to access the console of the guest partitions. virtio_console is probably not targeted for the end user, but to developers to perform configuration, debugging and development tasks through the system console.

- The virtio_rng driver implements a virtual RNG (Random Number Generator) for the guest partitions. virtio_rng provides a fast RNG to speed-up the security operations like: key-generation, authentication and encryption operations where a fast RNG is required.

5.3. Virtio Transport

The Virtio Transport Ring is the mechanism used for exchanging shared buffers between the guest and the host.

The Virtio drivers generate lists of scatter/gather buffers (or *scatterlists*). Such lists are the mechanism used by Linux to deal with the virtual/physical memory maps. Due to this memory model, even if a buffer looks contiguous in the virtual memory map, it may be scattered through several pages of physical memory. Thus, the Linux kernel offers a mechanism to get all these scattered buffers from a pointer in virtual memory. Those buffers are then recovered from the virtio transport ring and sent to the I/O partition.

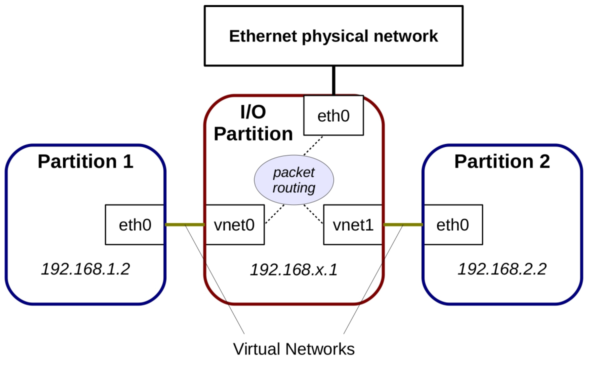

Figure 5. Virtual network architecture.

6. Virtio on XtratuM Partitions

After providing an overview of Virtio, here we focus on the task on being able to use Virtio on partitions running the XtratuM hypervisor. More specifically we consider the case where the partitions are running the para-virtualised Linux OS.

In first place the Virtio design assumptions are analysed in subsection 6.1. Next the implementation and modifications performed to meet the preceding assumptions are presented in subsection 6.2.

6.1. Virtio Design Assumptions

Virtio introduces some assumptions which hold for hosted hypervisors (like the Lguest [7] and KVM [4] hypervisors) where the guests are run *hosted* as user space processes on top of a general purpose operating system (Linux).

Instead XtratuM is a native (or bare machine) hypervisor thus some of the assumptions that Virtio places on the underlying hypervisor do not hold in the case of the XtratuM hypervisor. The following points summarise the Virtio assumptions that are relevant for the implementation of Virtio on the XtratuM hypervisor:

Assumption 1 ***(Hardware device support)****: The host supports (drivers) and has direct access to the hardware devices.*

This assumption is not met by the XtratuM hypervisor as it does not provide drivers for all the hardware supported by the Linux kernel.

This assumption is solved by providing the I/O partition as a Linux host partition which has both access to the physical hardware and drivers support.

Assumption 2 ***(Shared memory)****: The host partition has access to all the memory of the guest partition.*

This assumption presents issues with this design, as XtratuM partitions have strong spatial isolation, there is no way to access the guest memory, unless explicitly allowed in the XM_CF configuration file.

To overcome this problem the guest defines a memory area for the Virtio device virtualisation in the XM_CF configuration file which is shared exclusively with the host partition.

This provides a level of security by ensuring that no other guests have access to the data exchanged between the guest and the host using a Virtio device.

Assumption 3 ***(Host scheduling)****: The host operating system controls scheduling of the guest ("hosted" user process).*

This assumption presents issues with this design, as XtratuM partitions have strong temporal isolation provided by the XtratuM fixed cyclic scheduling policy. To overcome this problem the scheduling of the partitions must be carefully chosen to achieve a compromise between the host and the guest performance.

6.2. Virtio Implementation

In order to support Virtio for Linux partitions on the XtratuM hypervisor, the Linux partitions need to be modified to make them aware of Virtio devices, this is achieved by adding a low level virtio back-end.

The Virtio back-end is in charge of the tasks of device setup and discovery, of Virtio devices and the publishing and activity notification of Virtio buffers. This operations rely on the specific mechanisms provided by the hypervisor, which makes the back-end hypervisor dependent and required on both the host and guest partitions:

Host Virtio back-end:

Provides Virtio `virtqueues` support to the host partition, is also in charge of providing devices to the guest partitions. This is done by setting up the device configuration space (Virtio device descriptor page).

After providing the virtio device description, the guests starts using the device and the host is in charge of attending and serving all the Virtio requests made by the guest partitions.

Guest Virtio back-end:

Provides Virtio `virtqueues` support to the guest partition, is also in charge of the discovery/removal of the Virtio devices.

The device discovery and configuration is done as usual on a bus (e.g. the PCI bus), by reading and writing the device configuration space (Virtio descriptor page) which has been previously set by the host partition.

Additionally the management of notifications of Virtio device activity are performed by means of XtratuM IPVI (Inter Partition Virtual Interrupts).

Virtio Transport modifications:

As stated before, Virtio initial assumptions do not match XtratuM features, so the transport mechanism has to be modified in order to share buffers.

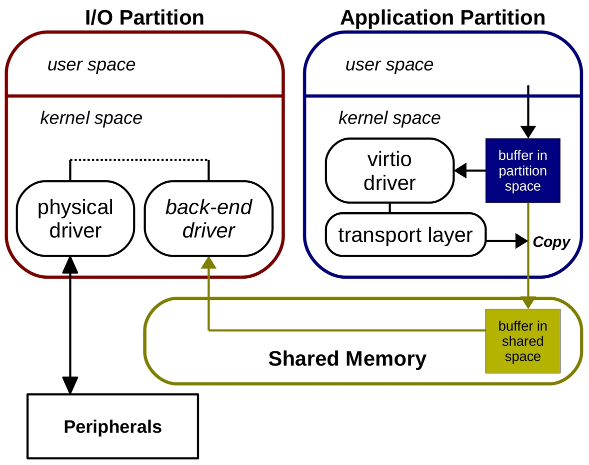

Figure 6. Modified Virtio transport ring for XtratuM.

In the case of XtratuM, partitions are granted spatial isolation. Therefore, the buffers passed to the Virtio ring belong to the guest partition memory map, and are not accessible by the I/O partition. Thus, the Virtio transport ring has been modified in order to copy the scatterlists to a memory area shared between the guest and the I/O partition. The resulting model has been depicted on figure 6.

The shared memory area has to host buffers of unpredictable but bounded sizes. This memory area is managed by a dynamic memory allocator. To avoid the fragmentation problem, the allocator only gives blocks of pre-defined sizes, much like a slab allocator. Therefore, besides avoiding fragmentation problem, its simplicity allows giving buffers in constant time.

7. Conclusions

In this paper we have presented the work done to provide device virtualisation to Linux guests running on the XtratuM [3] hypervisor in the scope of the OVERSEE project [5].

The presented approach is based on a I/O partition that exclusively owns the hardware devices and performs device virtualisation. The device virtualisation itself is solved by using the virtual I/O device standard known as Virtio [8] which provides "a series of efficient well-maintained Linux drivers which can be adapted for various hypervisor implementations".

The assumptions that Virtio performs about how the underlying hypervisor works are reviewed in order to design and achieve an efficient Virtio implementation on the XtratuM hypervisor.

References

[1] *Avionics Application Software Standard Interface (ARINC-653)*, March 1996. Airlines Electronic Eng. Committee.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[3] A. Crespo, I. Ripoll, M. Masmano, and S. Peiró. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *European Dependable Computing Conference (EDCC)*, pages 67–72, 2010.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[5] OVERSEE Project Consortium. OVERSEE Project: Open Vehicular Secure Platform, December 2010. FP7-ICT-2009-4. Project Id: 248333.

[6] PCI Special Interest Group. *PCI Local Bus Specification: Revision 3.0*, March 29 2002.

[7] R. Russell. Lguest: A simple virtualization platform for Linux, February 2008.

[8] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42:95–103, July 2008.

[9] R. Russell. Virtio PCI Card Specification v0.8.10 DRAFT, October 2010.