Grant Agreement number: 288574

Project acronym: **vIrtical**

Project title: SW/HW extensions for virtualized heterogeneous multicore platforms

Seventh Framework Programme

Funding Scheme: Collaborative project

FP7 -ICT -2011-7

Objective ICT-2011.3.4 Computing Systems

Start date of project: 15/07/2011 Duration: 36 months

D 3.1 *Hypervisor for ARM A15 and GPPA*

Due date of deliverable: July 2013

Actual submission date: July 2013

Organization name of lead beneficiary and contributors for this deliverable: VOSYS
Work package contributing to the Deliverable: VOSYS, UNIBO, UPV

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

APPROVED BY:

Partners	**Date**
All partners	**30th July 2013**

Table of Contents

1 Introduction

In this document we describe KVM on ARM implementation, targeting the Cortex-A15 implementation of the architecture, and also look into the implementation of GPPA resource sharing for virtual machines.

In the last decade virtualization has been established as a very powerful tool, expanding the capabilities of servers and enabling disruptive technologies, such as cloud computing. At the same time, virtualization has also been proven as a powerful tool for end users, system administrators, security researchers, and system developers. Virtualization has only started to show its capabilities on mobile and embedded platforms, however not unlike the desktop and server world, a very wide range of new use cases can be supported.

The Linux Kernel Virtual Machine (Linux KVM) is one of the most successful and powerful Virtualization solutions available, enabling the Linux kernel to boot guest Operating Systems under a process. Linux KVM has been designed to be portable, and has proven itself in a number of architectures, like Intel VT-x, AMD SVM, PowerPC and IA64, and has been ported on the ARM Cortex-A15 platform in the context of this deliverable.

For KVM, in this document we look into the architecture of the hypervisor, and also look into how KVM handles the cache coherency requirements imposed by the hardware.

2 The Linux Kernel Virtual Machine

In this section we look into the definitions involved with KVM and virtualization, and the architecture of the hypervisor. KVM is implemented as a kernel module, which allows a user space driver, such as QEMU, to implement virtual machine functionality. We will look into the split of functionality between KVM and QEMU and how QEMU takes advantage of KVM to provide a complete virtualization solution.

2.1 Hypervisor Software

Virtualization is a technique where an abstraction of the physical hardware is created in order to run applications and operating systems while hiding the details of the hardware used. The software that manages this abstraction is often called a Hypervisor or a Virtual Machine Monitor (VMM), and the abstractions created are called Virtual Machines (VM). Using a Hypervisor, one can run multiple operating systems on the same machine, at the same time. Each operating system is run under its own Virtual Machine and accesses physical hardware which is abstracted with the help of the Hypervisor.

It is common to classify virtual machines as Native Virtual Machines (Type I) and Hosted Virtual Machines (Type II). In the former case the Hypervisor is run directly on the hardware and can load different virtual machines side by side. In the latter case however, the Hypervisor is run as an application under an existing operating system, which is called Host. Virtual machines are run alongside the regular processes of the host and are called guests.

Not all Hypervisors are alike, since there is more than one way to do virtualization. For example a Virtual Machine may be designed to run software not intended for the hardware architecture used, as is the case with various emulators, or with software such as the Java Virtual Machine. However, we are mostly interested in Virtual Machines that can run the same software as the hardware architecture, unchanged or with minimal changes. Hypervisors that can run complete operating systems intended for the underlying hardware, with no changes to the code, are said to implement Full Virtualization; the software running under the VM is under the illusion it runs on real hardware. At the same time, there are Hypervisors that require the cooperation of the Operating System running under the VM; in this case the operating system needs to be patched to run under a virtual architecture slightly different than the real hardware. This kind of virtualization is called paravirtualization.

One of the most significant barriers to efficiently virtualize a given architecture is the presence of instructions that are sensitive to the current mode of operation of the processor. Typically an operating system running under a VM executes in a lower privilege mode than what it was designed for, and attempts to use instructions that control the state of the hardware. If these do not cause a trap to the Hypervisor and fail silently, or just behave differently in the lower privilege mode, then the Hypervisor would have to implement complicated binary patching techniques to intercept those instructions. Other challenges to efficiently virtualize a system include the way memory management and virtual memory are implemented, which mean that often Hypervisors have to maintain Shadow Page Tables incurring additional overheads.

In order to overcome these performance overheads, one solution is to implement paravirtualization instead of full virtualization. However, running unmodified guest operating systems is desirable, so hardware vendors have started shipping extensions to their processors so they can be efficiently virtualized. In that case, when a Hypervisor may take advantage of the hardware support for efficient virtualization, the system is said to support ***Hardware Assisted Virtualization***.

2.2 The KVM Architecture

KVM works by exposing a simple interface to user space, through which a regular process can request to be turned into a virtual machine. Usually QEMU is used on the user space side to emulate I/O devices, with KVM handling virtual CPUs and memory management.

The Linux Kernel Virtual Machine (KVM) is an established system virtualization solution, implemented as a driver running within Linux, which effectively turns the Linux kernel into a hypervisor. This approach takes advantage of the existing mechanisms within the Linux kernel, such as the scheduler, and memory management. This results in the KVM code base being very small compared to other hypervisors; this has allowed KVM to evolve at an impressive pace and become one of the most well regarded and feature full virtualization solutions.

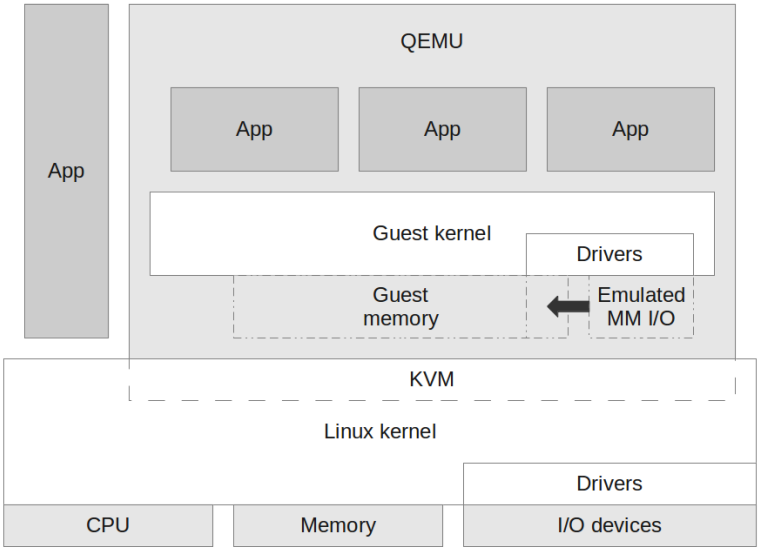


Figure 1. Virtualization using KVM and QEMU.

KVM is designed with a simple architecture in mind (see Figure 1), leveraging existing Linux infrastructure, including process scheduling, and memory management, thread and process creation. This is done by exposing an ioctl interface towards user space which allows a user space application to enable virtualization functionality, turning the Linux kernel itself into a Hypervisor. Through this interface, regular Linux processes are turned into virtual machines, with threads acting as virtual CPUs. KVM itself handles the switching of the context of the

processor when the process that corresponds to a virtual machine gets loaded by Linux, taking advantage of the virtualization extensions supported by the hardware. In this fashion the processor and the memory are virtualized, however to virtualize I/O devices, such as network interfaces and storage, an interface to user space exists, so these can be emulated by the application setting up the virtual machine (usually the QEMU emulator).

2.3 The User Space Driver: QEMU

QEMU functions as a caller from user space for KVM, e.g. setting up the memory of the VM to be launched, the virtual CPUs to be used, etcetera. QEMU (with the help from KVM) configures memory regions that would trap when the guest attempts to read or write to them; the execution workflow will return to QEMU, which emulates the behavior of memory mapped I/O devices (MMI/O), such as network interfaces, graphics controllers, and storage and user interface devices, such as keyboards. Depending on the underlying architecture QEMU may also handle injecting interrupts and emulating an interrupt controller in the same fashion.

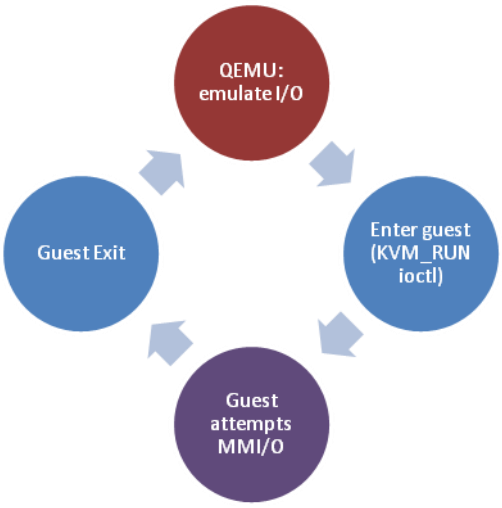


Figure 2. Basic view of KVM and QEMU interactions.

As shown in Figure 2, synergy between QEMU and KVM is based on a standard ioctl system call interface which KVM exposes to user space; QEMU simply issues ioctl commands to setup KVM, and to enter execution inside the guest. In the case of a guest exit, QEMU is able to determine why the guest stopped executing and take appropriate action, e.g. by emulating a MM I/O operation.

3 KVM on the ARM Cortex-A15

The KVM port on the ARM Cortex-A15 has been developed in this task, and is mature and stable for guest operating systems targeting the Cortex-A15 processors. The port utilizes the hardware virtualization extensions present in the ARM Cortex-A15 in order to efficiently switch between guests.

Since virtualization on ARM did not have to go through the various stages of poor hardware virtualization support as in x86, the port includes support from the beginning for advanced hardware features targeted at virtualization. These include support for two stages of memory translation, and virtualization of the interrupt controller for improved interrupt delivery to virtual machines.

3.1 Second Stage Address Translation

The Virtualization Extensions allow setting the VTTBR register when in HYP mode, which controls the second stage of memory translation. A Hypervisor, such as KVM will set this register before switching to a guest.

This register includes the physical address in memory of the first level of the stage 2 page tables that are used to translate memory accesses by the guest, and also the VMID assigned to it.

Part of the HYP mode HCR register, the HCR.VM bit controls whether a second stage of memory translation will be used. When disabled, the mapping of IPA to PA is flat and the guest's memory operations will not trap in HYP mode.

When the seconds stage of translation is enabled, faults caused by permission settings in the first stage page tables, will trap to the guest OS as usual. However, for the second stage of translation the page tables can also set their own permission controls; these will trap in HYP mode so that memory management features can be implemented for VMs, and also to allow handling of memory mapped I/O emulation.

Most of the code of interest for the guest's MMU support can be found in arch/arm/kvm/mmu.c. The function that is used to set up the initial empty page tables for the second stage is called from the main KVM code in kvm_arch_init_vm().

When exiting from the guest due to a Stage 2 translation fault, KVM will handle the fault using the kvm_handle_guest_abort() function. Stage 2 aborts might also occur because of MMIO accesses that will be emulated; so this code largely decodes the registers with the clues about the fault that caused the exit, and react accordingly.

If the fault is indeed due to the unmapped guest memory, then the user_mem_abort function takes over: However, if the fault was caused by an MMIO access, the io_mem_abort() function takes over instead.

3.2 Virtual Interrupts Support

The ARM Generic Interrupt Controller (GIC) architecture includes a distributor block, where interrupts from devices themselves are delivered. This component will distribute interrupts to the CPU interface blocks, which talk to the CPUs according to the configuration and interrupt prioritization set by the system.

Every CPU in the system interfaces with the GIC architecture through its corresponding CPU interface in the GIC; this is where the CPU can determine which IRQ has interrupted the system's execution and where it can choose what interrupts it wishes to see. Importantly, an interrupt should be marked as being handled here as well.

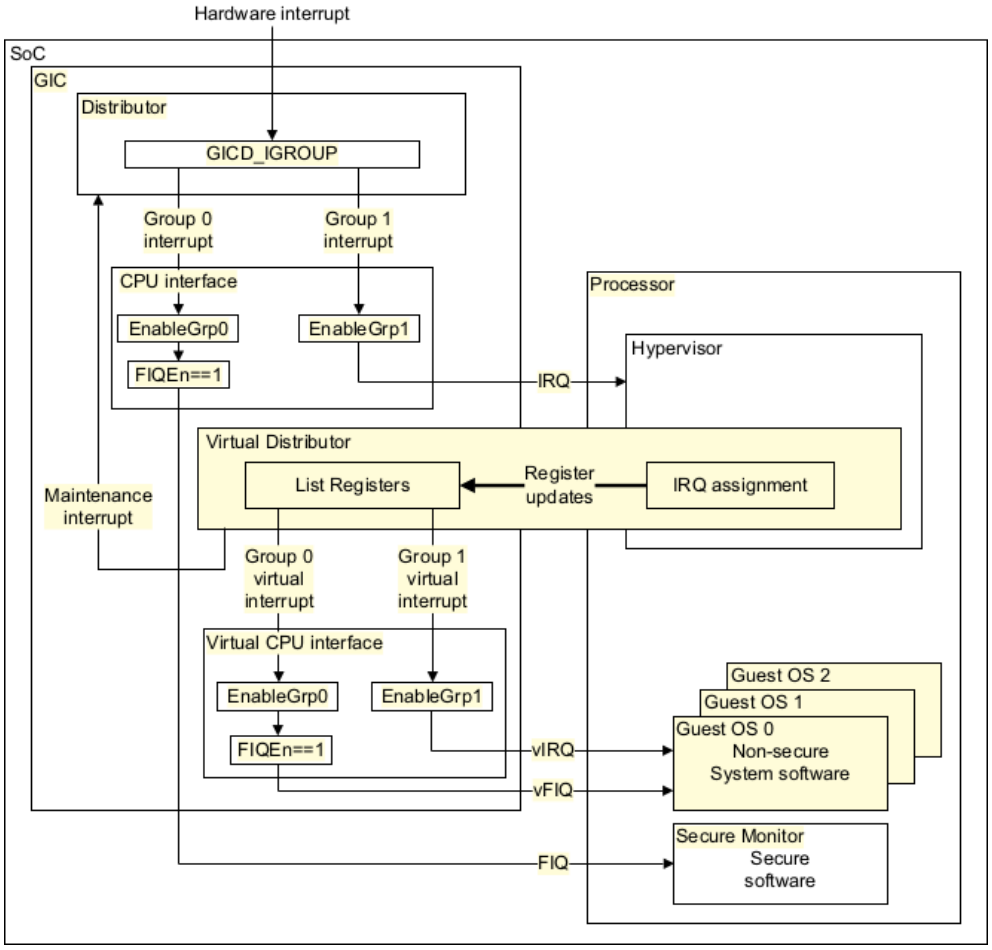


Figure 3: The VGIC Architecture

The ARM Virtualization Extensions also add the possibility for a virtual CPU interface alongside the physical one. These still correspond to physical CPUs, and should be configured according to the VCPU and guest VM that is being executed on the CPU at any point in time. Therefore on a system which includes support for the ARM Virtualization Extensions, there are as many virtual CPU interfaces as there are physical ones.

The virtual CPU interface, includes a virtual interface control block, which allows us from hypervisor mode to control what kind of state the guest will perceive from its end. The most significant of these are the List Registers (LR) where the list of active and pending interrupts to be delivered to the guest can be saved.

Each virtual CPU interface block is physically implemented and will allow a guest to access it, without incurring any exits to the hypervisor, while still having control of the virtualized state of interrupts the guest will receive. The format used is almost identical to the format used in the physical CPU interfaces, and a guest will not be able to deduce that it is running in a virtualized environment, at least from an interrupts perspective.

A hypervisor, such as KVM, will:

- Emulate the distributor's behavior and, its interface to the system using regular traps of MMIO operations.
- Set the virtual CPU interface control registers, when switching to a guest system.
- Map the physical address of the virtual CPU interface in the guest intermediate physical address space, in the area expected by the guest.

KVM emulates a distributor, by trapping MMIO operations as described in the previous subsection.

On a context switch, KVM updates the List Registers, to add any new interrupts to the list of pending interrupts for the guest that will be entered. A guest system will handle an interrupt from the virtual CPU interface in the same way as it would handle a physical interrupt. The Guest OS cannot detect that it is receiving interrupts from a virtual device instead of the physical hardware.

4 Cache coherency support in KVM for the Cortex-A15

There are a number of cases where KVM needs to perform cache and TLB maintenance operations, in order to avoid stale data from being accessed by a virtual machine. We will go over them in this section. The maintenance operations described here have been implemented assuming a system where only the inner shareable domain is used by CPUs, an assumption which is true for all the targets supported by KVM.

4.1 VMID recycling operations

In order to avoid the need to perform TLB and cache flushes each time a new VM executes on a given CPU, the hardware provides the capability to tag VMs with an 8 bit value called VMID.

KVM takes advantage of VMID by assigning a unique value to each VM as it is executed, starting from 1; VMID zero is reserved for the host. However the VMID being an 8 bit value, there is an inherent limit of 255 virtual machines running on the system. KVM gets around this limitation by implementing a VMID recycling scheme.

VMID recycling is implemented by keeping track of a system wide “VMID generation” value. For a given generation, the complete 255 VMID range can be used, assigning one VMID per VM on demand as VMs are loaded for execution. However, at the same time KVM checks for each VM if there is already a VMID assigned to it in the current generation; as long as the generation value remains unchanged, so is the VMID that correspond to a virtual machine.

Eventually, if there are more than 255 virtual machines, the VMIDs will run out. It is at this point that the VMID generation value will be incremented by one, and all previously assigned VMIDs will be considered invalid, assigning them to VMs from scratch.

At this point, the first VM to be loaded within a given VMID generation will also cause KVM to perform a complete ***TLB and instruction cache flush***. Those structures will contain stale data tagged by VMID, but since they are to be reassigned from scratch; KVM ensures that this data will not be accessible by a VM which uses a VMID that previously was assigned to another.

This is implemented in the KVM source code in arch/arm/kvm/arm.c in update_vttbr(), which calls __kvm_flush_vm_context from arch/arm/kvm/interrupts.S in order to perform the flushes.

4.2 TLB Maintenance during Stage 2 Table Updates

KVM implements a lazy scheme where stage 2 page tables are being filled on demand as they are being used by the guest. Each time we update these page tables, all TLBs in the system need to be updated. A flush by IPA is implemented in __kvm_tlb_flush_vmid_ipa in arch/arm/kvm/interrupts.S, which is called every time the stage 2 page tables corresponding to a virtual machine are updated in arch/arm/kvm//mmu.c.

4.3 Upgrading uniprocessor data cache flushes to all processors

According to the ARM Architecture Reference Manual, paragraph **B1.14.4**:

Virtualizing a uniprocessor system within an MP system, permitting a virtual machine to move between different physical processors, makes cache maintenance by set/way difficult. This is because a set/way operation might be interrupted part way through its operation, and therefore the hypervisor must reproduce the effect of the maintenance on both physical processors.

In order for a uniprocessor system to safely execute on KVM, when the virtual machine performs a data cache flush on only one vCPU, KVM needs to perform a system wide flush for all physical CPUs. This is because the vCPU might at any time migrate to another physical CPU.

This behavior is implemented in arch/arm/kvm/coproc.c in access_dcsw().

4.4 Upgrading Barriers

When switching to a virtual machine, the HCR.BSU_IS bit will be set by KVM; any memory barrier operations performed by the guest operating system will be then upgraded by the hardware to apply to all CPUs in the inner shareable domain. This is implemented in the configure_hyp_role macro in arch/arm/kvm/interrupts_head.S.

4.5 Instruction Cache Coherency

According to the ARM Architecture Reference Manual, paragraph **B3.11.2**, the way instruction caches are kept coherent depends on the architecture implementation. The required behavior is implemented in arch/arm/include/asm/kvm_mmu.h in coherent_icache_guest_page(), which is called whenever a stage 2 page corresponding to the virtual machine needs to be mapped.

In the case of a Physically Indexed Physically Tagged cache, that page needs to be kept coherent at all times by the host, considering the same page may be mapped by the host or another virtual machine.

Particularly in the case of a Virtually Indexed Physically tagged cache, the entire instruction cache needs to be flushed.

Virtually Indexed Virtually Tagged caches are tagged using the ASID and the VMID, the Address Space and Virtual Machine IDentifiers. These color TLB entries by their respective virtual machine and process, and therefore no additional cache maintenance needs to be performed other than the VMID recycling scheme described previously.

4.6 HYP Mode Data Caches Maintenance

KVM will map a number of structures in HYP mode memory, which are used in the context switches. When the corresponding pages are mapped, the data caches are flushed to the point of coherency by calling kvm_flush_dcache_to_poc() in kvm_mmu_init() and __create_hyp_mappings in arch/arm/kvm/mmu.c.

5 Architecture for Integration of the GPPA within KVM

In the following sections the virtualization infrastructure of the GPPA is described following the logical flow of Figure 4, from the *guest* to the physical GPPA device.
The virtualization of the GPPA amongst different virtual machines is performed using the model imposed by the KVM hypervisor, where I/O is completely emulated.
From the *host* point of view the GPPA is a standard Linux device, appearing in the system as a character device. The *host* has a complete view of the GPPA and is aware of all types of resources available on it (e.g. number of free cluster, memory).
Each *guest* Operating System, instead, needs an isolated view of the GPPA with respect to other *guests* and will not have access to the actual state of the device. *guests* also see the GPPA as a character Linux device, with the difference that there is no actual hardware communicating with the Linux driver. The *guest* Linux driver is instead communicating with an emulated version of the GPPA (*GPPAv*).
Standard I/O virtualization in KVM-based systems is not handled by the Hypervisor itself, but is rather demanded to QEMU (1). QEMU is the machine emulator used by KVM to run each *guest* Virtual Machine. The implementation of virtualization extensions for the GPPA does not involve any modification to the KVM Hypervisor.
When an ARM-based guest system is used, such as in our case, QEMU provides the abstraction of an ARM Versatile Express baseboard. The way QEMU emulates I/O devices is

based on an extension of the ARM Versatile baseboard and is implemented as a virtual device mapped into the memory map of the virtual system, emulating the behavior of real hardware devices.

Each guest Virtual Machine is provided with a (fully-virtualized) Linux driver, which is referred in the following as *gppavdriver* (the virtual GPPA driver). *gppavdriver* exposes an *ioctl* interface that will be used by the OpenMP runtime, on behalf of applications running on guest systems to offload tasks to the GPPA.
As described in Deliverable 2.1, the OpenMP runtime system invokes the virtual GPPA driver's ioctl interface passing as a parameter a task descriptor structured as follows:

```
struct data_desc {
  unsigned int * ptr;
  unsigned int size ;
}


struct mdata {
  unsigned int n_data ;
  struct data_desc data [n_data] ;
}

struct otask {
  char * name ;
  int id;
  int num_clusters;
  int qos_channels;
  int *qos[2];
  struct mdata * shared_data ;
  struct mdata * fprivate_data ;
  struct mdata * lprivate_data ;
  /* Filled in by lower levels */
  unsigned int bin_size;
  void * bin_pointer;
  unsigned int clusters_bitmask;
  unsigned int *noc_conf_bits;
  unsigned int *data_context;
}
```

The task descriptor is passed through virtualization layers (*guest* ➔ *host* virtual mem, *host* virtual mem ➔ *host* physical mem). Explicit copies of binary and data are implied at each layer traversal. New memory is allocated and the pointers in the task descriptor are updated accordingly. The first copy takes place in the GPPA emulation device to resolve the second level of virtualization (*guest* ➔ *host* virtual mem), while the second copy happens inside the GPPA *host* driver (*host* virtual mem ➔ *host* physical mem), moving binary and data to a contiguous memory area managed by the *host* Linux driver and accessible from the GPPA.

Once requests arrive to the *gppavdriver,* the *ioctl* function communicates with the GPPA emulation device using a set of ioread/iowrite calls to the address range at which the emulation device is mapped. It is important to understand that when executing the GPPA emulation device, the control is not anymore on the *guest* system but rather on the *host*. The emulation is in fact part of QEMU, which is in turn a process running on the *host* system.
Each GPPA emulation device communicates with a process running on the *host* system and in charge of handling requests coming from different *guests*. This module is called *GPPA Bridge* and implements the logic to assign GPPA resources to each requesting application, according to a scheduling policy. The *GPPA Bridge* interacts directly with the physical GPPA device, and once the scheduling and resource assignment decisions are taken, it will forward the specific offload request to the GPPA through the physical GPPA Linux driver (*gppadriver*) running on the *host* system.

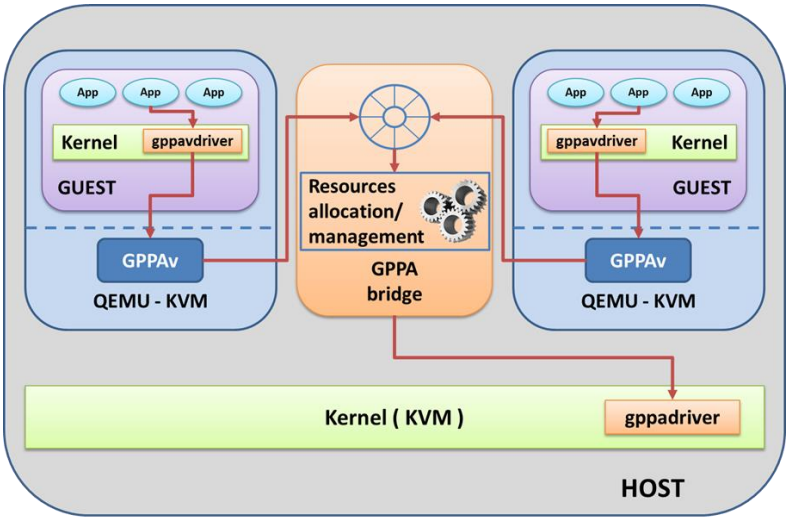


Figure 4: Virtualization Framework overview

5.1 GPPA Virtual Driver (gppavdriver)

The GPPA Virtual driver is located at the very top of the virtualization stack implemented for this system. It is used to give each *guest* Operating System the illusion of a dedicated GPPA device. Applications communicate with the driver using the Linux ioctl system call.

The interface implemented via ioctl defines the following services:

- *Task Offload*: This function is used to offload a task to the GPPA, a pointer to the task descriptor is passed as parameter. This function also returns the id of the specific offload request. This is an asynchronous operation, once called the application can continue executing other code.
- *Wait Task Completion*: This function is used to define a synchronization point between an application and the GPPA. The identifier of the request is passed as a parameter.

Figure 5 depicts the logical flow of a *Task Offload* request. The guest driver will receive via the *ioctl* a pointer to the task descriptor which is then copied into the kernel space and forwarded using iowrites to the GPPA Emulation Device (arrows 1 and 2 in Figure 5). During the offload procedure applications wait until the ID of the task is returned by the ioctl (arrow 10 in Figure 5). -1 is returned in case of an error. In case of error the offload procedure is executed on the *host* processor.

Figure 6 depicts the logical flow of a *Wait Task Completion* request. The guest driver receives, as parameter of the ioctl call, the identifier of the task for which the application wants to wait (arrow 1 in Figure 6). This request is then forwarded to the GPPA emulation device (arrow 2 in Figure 6). The requesting process is inserted in a wait queue of the Linux kernel in which it will stay until a response to the waiting request arrives (arrow 8 in Figure 6). The way responses are notified to the application is based on interrupts. An interrupt is raised by the GPPA Emulation device (using the qemu_irq_pulse built-in function) and the interrupt handler will wake up only the process waiting for that response. Once awake, the application is sure that the computation of the specific task has finished.

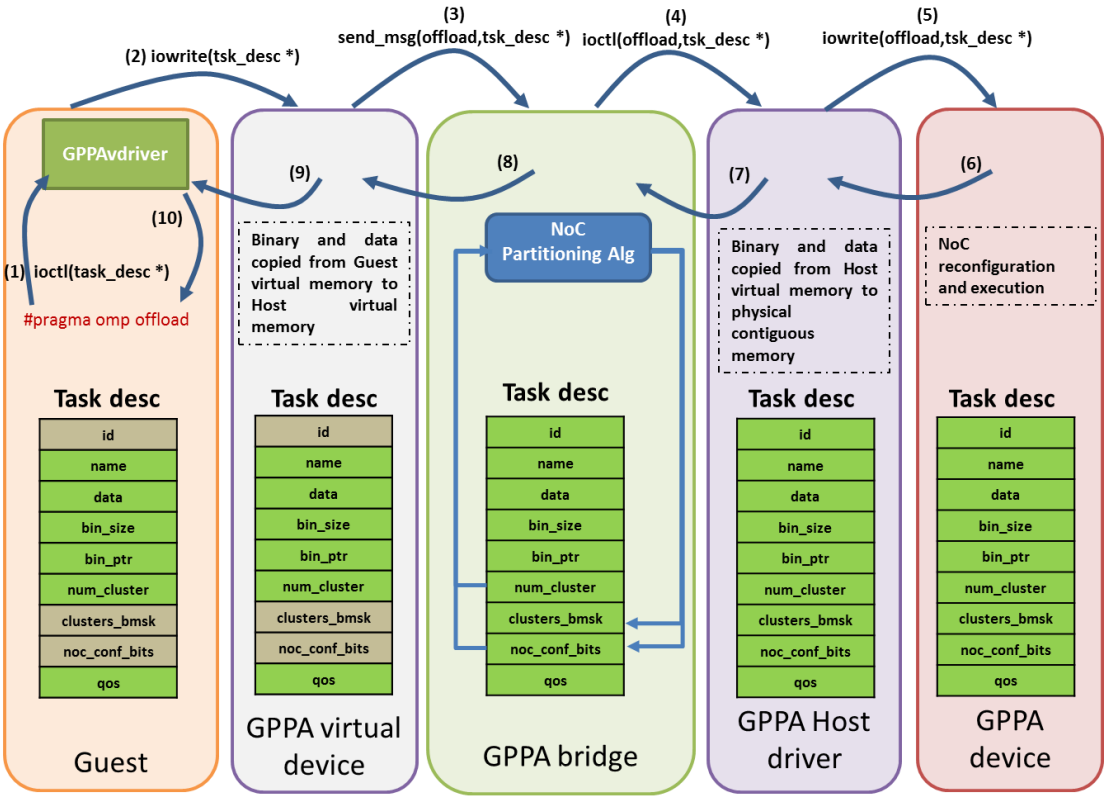


Figure 5: Logical flow of a task offload command

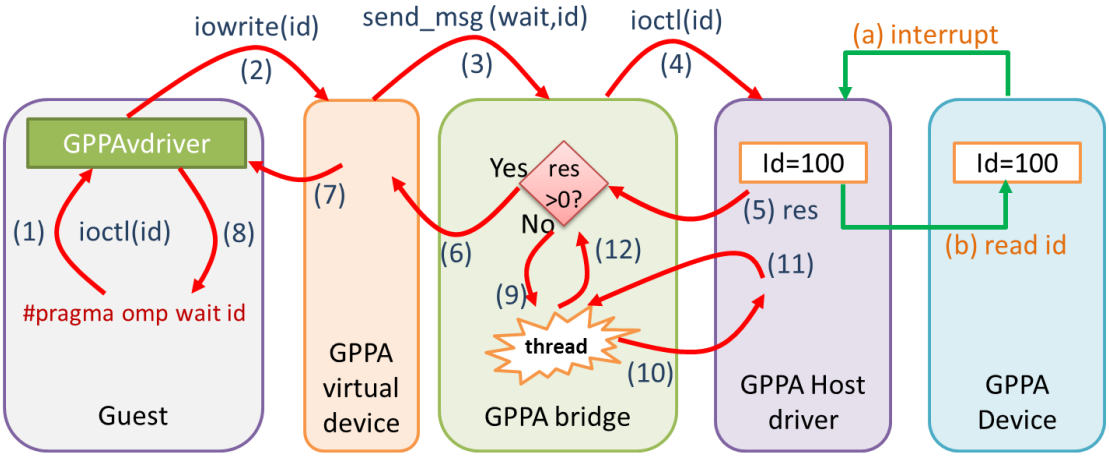


Figure 6: Logical flow of a task wait command

5.2 GPPA Emulation device (GPPAv)

The GPPA emulation device is a software module which is developed as an extension of QEMU. QEMU offers a simple way to enhance its virtual machine model with custom devices. Once designed each virtual device is attached to the bus of the platform modeled by QEMU and mapped at a user-defined address range.
Any ioread/iowrite call made by applications running on a guest operating system, and falling within the address ranges where the custom devices are mapped, is caught by QEMU and redirected towards the virtual device.
This virtual device is the crossing point between the *guest* world and the *host* world in which any scheduling or sharing decision regarding the GPPA is taken by the GPPA bridge.

The *GPPA virtual* device is interfaced with the *GPPA bridge* using POSIX queues, which are an Inter Process Communication mechanism provided by Linux-based systems. We preferred to use POSIX queues instead of Linux Shared Memory or Unix sockets for three main reasons:

1. POSIX queues provide the synchronization mechanisms needed to control possible multiple accesses to the same queue.
2. POSIX queues provide persistence to all messages, in case of an application crash it is possible to re-attach to the same queue and recover all messages present at the moment of the crash.
3. POSIX queues provide the possibility to assign different priorities to different messages. In the future, this will allow for implementing priority based scheduling algorithms for requests coming from different *guests*.

Each time an application running on a *guest* system needs to communicate with the GPPA, its request is first caught by the GPPA virtual device, which in turn redirects it to the GPPA bridge using POSIX queues. In particular, we define a single POSIX queue for messages going from *guests* to the GPPA bridge, while a POSIX queue per virtual machine for messages coming back from the GPPA bridge (Figure 7).

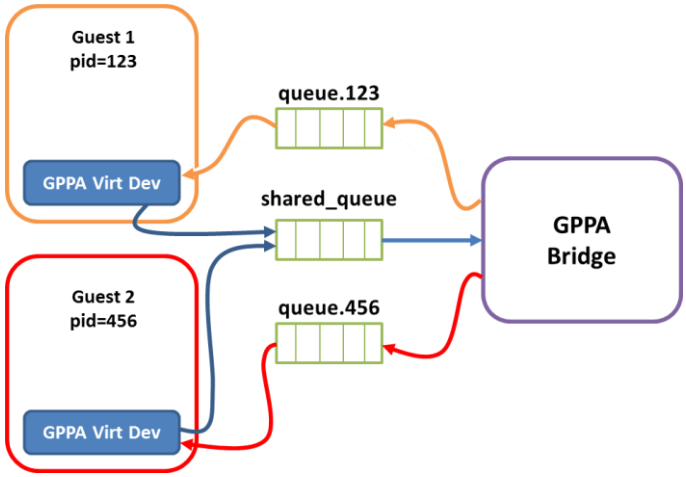


Figure 7: POSIX message queues

When the device is initialized, it first creates its private message queue, whose unique reference in the system is composed by the string "/queue." concatenated with the PID of the QEMU process (Figure 7). Queues have a maximum depth of N messages, the number of physical clusters available in the GPPA (16 in the reference simulation platform). Since we are considering the Cluster as the minimum resource scheduling quantum, there will be no more

than N applications running at the same time on the GPPA. Messages are composed as follows:

```
typedef struct msg{
    unsigned int cmd;
    task_desc payload;
};
```

The virtual device is then attached to the shared POSIX queue (which has already been created by the GPPA bridge). After queue creation, the virtual device sends a first request to the bridge to register itself to the system. This procedure is mandatory and will allow the *guest* to push further requests in the future.

Each GPPA emulation device has a separate thread which is in charge of waiting responses on the dedicated POSIX queue. Whenever a response arrives, the GPPA Virtual Device associated to the destination *guest* will raise an interrupt (arrow 9 in Figure 5 and arrow 7 in Figure 6). The *guest* Linux driver will catch the interrupt and wake-up the application waiting for the response. The way interrupts are raised is based on a helper function provided by QEMU (qemu_irq_pulse) and the interrupt number assigned to the *GPPAv* is defined when the device is first instantiated.
The multi-threaded structure allows the device to simultaneously handle requests coming from applications and responses coming from the *GPPA bridge*.

Whenever a request from a *guest* arrives to the *GPPAv*, it is immediately forwarded to the *GPPA Bridge* using the shared POSIX queue (arrow 3 in Figure 5 and Figure 6). At this point, the first virtualization layer is crossed. A copy takes place and before the offload request is actually forwarded to the bridge all data buffers and binary are replicated into the *host* memory space (Figure 8).
The virtual device defines a shared memory segment for each data structure to be copied (i.e. one for the binary and one for each data element). Using shared memory is the simplest and most efficient way of sharing data between different Linux processes, a QEMU instance and the *GPPA bridge* in this case. Binary and data elements are then copied into the shared memory using a helper function provided by QEMU to access the memory of the guest.
The task descriptor is updated by replacing the pointer to each data structure with the identifier of the shared memory segment and is used as payload of the message to be forwarded to the *GPPA bridge*, the `cmd` field of the message is also updated depending on the type of request (*Task Offload* or *Wait Task Completion*).

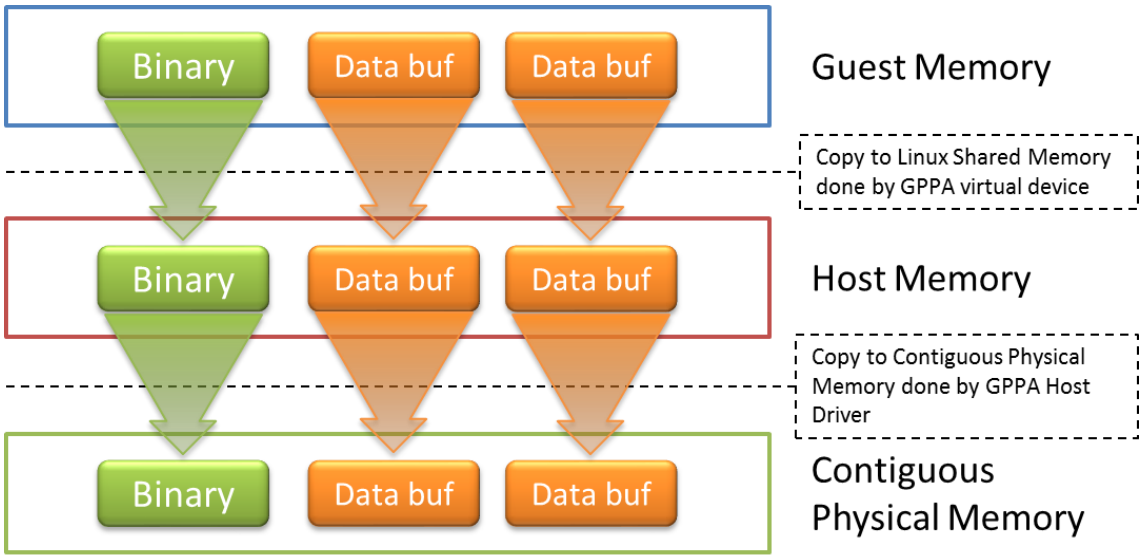


Figure 8: Binary and data buffers copy scheme

5.3 GPPA bridge

The *GPPA bridge* is the heart of the proposed virtualization infrastructure. In this module all decisions regarding the scheduling/sharing of the GPPA are taken.
This module is a server process composed by two POSIX threads, in charge of forwarding requests to the real GPPA and providing responses to the various Guests, respectively.
At startup this process creates the shared POSIX message queue used by all *guests* to push offload requests to the GPPA. This queue has a unique name inside the system which is known to all *guests*. After creation of the shared queue, the bridge starts waiting for incoming requests; it accepts three different commands from *guests*:

- Virtual Machine registration (GPPA_REGISTER_VM)
- Task Offload request (GPPA_TSK_OFFLOAD)
- Task completion check (GPPA_TSK_END)

This module maintains two status tables: in the first (Figure 9) all Virtual Machines are registered at boot time using the GPPA_REGISTER_VM command. In the second table (Figure 10) all information regarding applications already running on the GPPA is stored. It also resembles the actual state of the GPPA (i.e. number of free clusters).

ID	VM PID

Figure 9: Virtual Machine table

ID	VM ID	APP PID	Clusters bmsk	NOC cfg

Figure 10: GPPA/applications status table

The thread in charge of accepting requests from *guests* will extract and serve them in a *First Come First Serve* (FCFS) order from the shared message queue. The scheduling policy for offload requests is described in the following section.

5.3.1 Offload request scheduling and GPPA-NoC partitioning

In this section we describe the scheduler algorithm to support efficient and contiguous resources allocation in the GPPA. The algorithm is in charge of identifying the clusters to be assigned to the different requests for GPPA resources.

The scheduler algorithm is implemented inside the bridge (see Figure 4), which enables the GPPA-NoC partitioning. In an attempt to reduce the fragmentation of the GPPA resources, a weight strategy is implemented. Moreover, we enable a novel strategy to reduce the complexity (computation time) of the algorithm.

We face two important challenges when receiving a new request. The first one is determining the clusters to be assigned to the request in a contiguous way (a partition). This is quite challenging, as it is not straightforward to obtain a contiguous partition compatible with the routing algorithm to be used in the NoC of the GPPA. The second challenge is how to perform such operation in a fast manner, minimizing processing time and required resources amount.

Based on the number of clusters required by a set of requests, the partitioning decision is computed, the selected clusters for each partition must be obtained in a contiguous way, and then, the configuration of the NoC (mainly routing bits for the partition) must be computed accordingly.

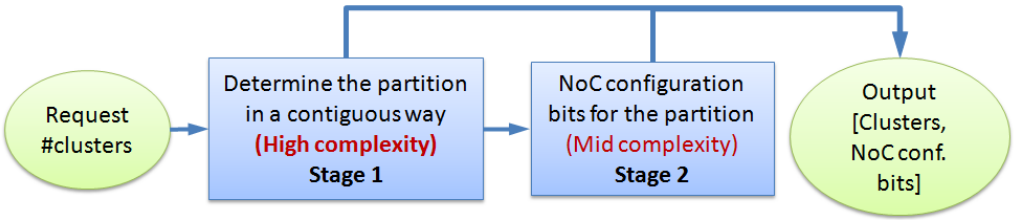


Figure 11: Complete process to compute the scheduling decision from a request.

Figure 11 depicts the stages needed to effectively apply both NoC partitioning and NoC configuration in the bridge. In Stage 1 the scheduler algorithm is computed and the clusters that form the contiguous partition are identified. The complexity of this stage depends on both the number of clusters and mainly on the size of the NoC topology. Depending on the previous parameters the number of combinations can be high. Obviously, since the partition selection is a recursive process we consider this task as complex. In order to cut down the complexity and due to performance reasons, we limit the shape of the partitions by considering some kinds of shapes (Figure 12), while other more sophisticated and irregular shapes are excluded (Figure 13). Also, we fix the maximum size of a partition to eight. This prevents a request to take all the GPPA resources. Notice that preemption is not implemented.

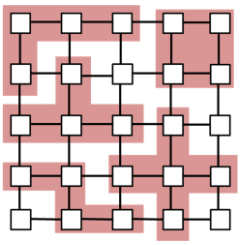


Figure 12: Allowed partitions.

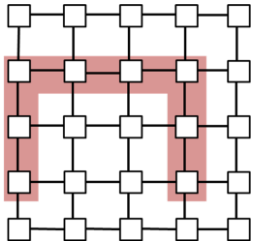


Figure 13: Not allowed partitions.

In the second stage (Figure 11) the NoC configuration bits must be determined to effectively apply a routing algorithm on a partition. The complexity of this stage depends on the routing algorithm but mainly on the irregularity of the partition. An irregular partition requires a topology-agnostic routing algorithm (e.g. up*/down* or SR). For instance, in up*/down* a spanning tree is built and all the unidirectional links are labeled as up or down. Routing restrictions are placed in down → up transitions. We can consider the complexity of this approach as medium since it is not a recursive process but neither is simple. Moreover, the NoC configuration bits (routing and the connectivity bits) are obtained. This is a straightforward process since there is a direct relationship between the routing restrictions and the routing bits.

In order to face both challenges we will overcome the complexity of the previous process with a fast method to compute the partition and the NoC configuration bits of most of the frequent partitions. The two stages will be embedded in a compact database (DB). The DB will be computed at design time and will provide the shape of the partition as well as the set of NoC configuration bits that need to be used to satisfy a new application allocation. Therefore, the bridge will embed the DB and upon reception of a new request will compute the proper set of clusters (the partition) and the NoC configuration bits that need to be modified to support the partition. The most important thing is that the time required for the scheduler algorithm will be reduced, as it will only access the DB.

As Figure 14 shows, the method consists in a pre-computed DB containing all the partitions derived from requests from 1 to 8 clusters in size. For each partition, the DB also indicates the NoC configuration bits to be configured which set-up a correct routing function for the partition. Also, each entry of the DB will be labeled as available depending if all the clusters that form the partition are available or not.

DATABASE				
#clusters	Shape	Weight	NoC Conf. bits	Available
1	[0]	2	010110...	Yes
1	[1]	1	011000...	No
...	...	...	...	...
4	[0,1,4,5]	15	101100...	Yes
4	[2,3,6,7]	23	011010...	Yes
4	[4,5,10,11]	25	111010...	No
...	...	...	...	...

Figure 14: Database that contains all the partitions.

The algorithm interfaces with the task descriptor shown before in this deliverable. In essence, a new request comes for a given number of clusters reflected in the field `num_clusters` in the `otask` struct. The algorithm attempts to provide a partition with the required number of clusters, but it can override the original request and provide a smaller number of clusters (e.g., if the original request exceeds the actual availability). In case the number of clusters is changed with respect to the original request, the algorithm modifies its value in the `num_clusters` field. The IDs of the granted clusters are annotated in the `clusters_bitmask` field. Thus, the application knows which clusters can use. Also, the algorithm provides the configuration of the NoC by writing it into the `noc_conf_bits` field.

As we have previously mentioned, in an attempt to reduce the fragmentation issue, a weight strategy is implemented. The weight will define the selection function. Since there are a huge number of combinations for a particular request, the selection function is in charge of selecting a proper entry of the DB table. Initially, we assign a weight to each cluster depending on the available neighbors. Figure 15 shows the case when all the clusters are available. As we can see, the corner clusters are labeled with a weight of two since they have two available

neighbors. Also, the weight for a particular partition is the sum of the cluster weights that form the partition. For instance, the red partition in the figure is set up as 11, which is the result of the sum of the cluster weights. Then, the selection function consists in selecting the partition with the minimum weight. In this way, the partitions located near the corners and in the boundary of the mesh have higher probabilities to be selected by the algorithm. Moreover, both the cluster weights and the partition weights are updated by setting the proper free neighbors each time a partition is assigned. In the previous example, if the red partition is finally assigned then the weight configuration is updated (Figure 16). With this method, the fragmentation of the resources is considerably minimized. Later, the performance of this method is evaluated.

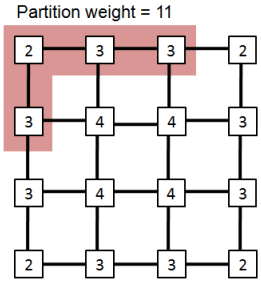


Figure 15: The clusters are marked with weights. The partition weight is the sum of all the cluster weights.

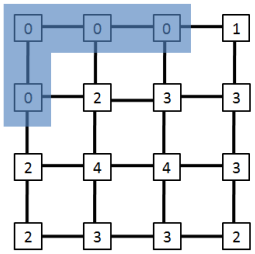


Figure 16: The weights are updated when a partition is assigned.

Finally, the DB should be also updated when a partition is assigned. This fact will invalidate many others partitions in the DB and also will update the weight of the partitions. This process could require higher computational requirements. For this reason, we perform this task off-line by running it in background (and with low priority).

Figure 17 depicts the final scheme. A Fast-Table is implemented in order to be ready for a particular request in a very fast manner. In this way, the scheduling decision will assign clusters to requests over a fixed and very short period of time. The Fast-Table format is depicted in Figure 18. As we can see, the selection function (off-line computed) fills the fast-table with the selected partitions ready for a particular request.

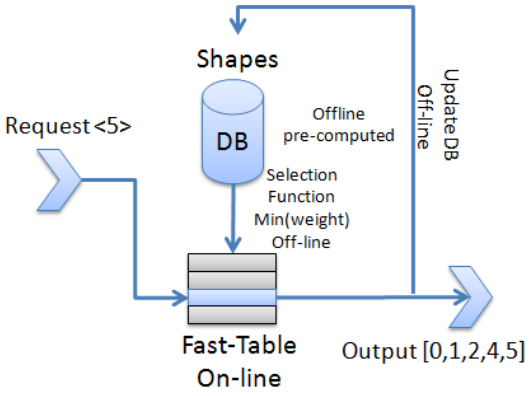


Figure 17: Final scheme of the scheduler algorithm.

FAST TABLE						
2	3	4	5	6	7	8
[0,1]	[0,1,4]	[0,1,4,5]	[0,1,2,4,5]	[0,1,2,4,5,6]	[0,1,2,3,4,5,6]	[0,1,2,3,4,5,6,7]

Figure 18: The Fast-Table contains one partition for each kind of request.

In the next figures we show the performance evaluation of the scheduler algorithm in terms of execution time and required memory footprint. Figure 19 shows the computation time for the different actions performed by the scheduler algorithm when a 5-cluster request arrives to the bridge. The algorithm was coded in C and executed on the instruction set simulator of an ARM7 processor core.

As we can see, for the online access to the Fast-Table the number of processor cycles is significantly low (474 cycles). With a 700MHz operating frequency assumed for the processor core running the bridge, this means that the scheduling decision is ready in less than 0.7 microseconds, which is quite low. For updating the DB and the Fast-Table by the selection function (off-line computed), we can see a logical increment, however, the required time is always lower than 66 microseconds.

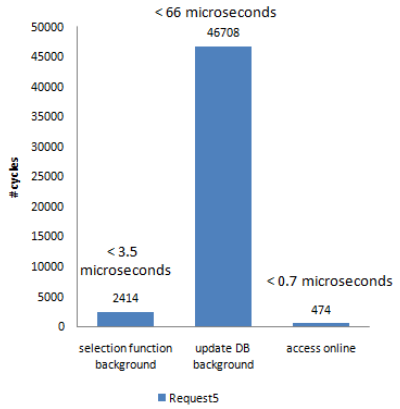


Figure 19: Computation time of the different actions performed by the scheduler algorithm.

Regarding memory footprint, Figure 20 shows the number of bytes needed in order to store the DB. As can be seen, the total memory depends on the number of combinations that will be considered (see Figure 21).

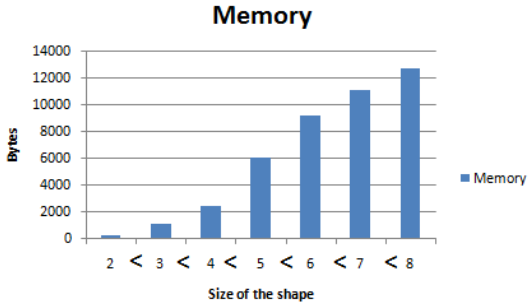


Figure 20: Bytes needed in order to store the DB in the bridge (accumulated bars).

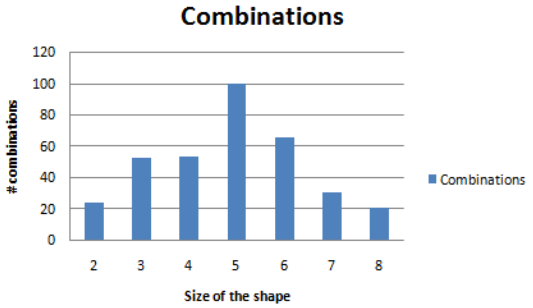


Figure 21: Number of combinations for each kind of request (from 2 to 8 clusters).

Finally, in order to analyze the fragmentation issue we compare two different scheduling policies. Then, using the same method previously described, we will compare our weight scheduler against a random scheduler. The only difference between both schedulers is the way

the selection function fills the fast-table, therefore, the way the partitions are selected when new requests arrive to the bridge.

For this comparison we analyze 100 random requests, which arrive to the bridge at different time. Moreover, since the level of the NoC fragmentation depends on the system load we analyze three different system loads. The LOW load defines a set of requests where the applications have a low duration, the MEDIUM load defines a set of requests with medium duration and the HIGH load defines a set of requests with high duration.

Additionally, we distinguish among three different cases. The blue bar (FullySatisfied) represents all the requests that were fully satisfied, that is, this bar depicts all the grants in which the assigned resources are equal to the required resources. The orange bar (PartiallyOptimal) shows the number of grants that were partially satisfied due to the lack of resources in the system. For instance, a 6-cluster request was satisfied with the last 4 free contiguous cluster resources. And finally, the yellow bar (PartiallySubOptimal) depicts the same information than the orange bar but in this case there are more free resources than the assigned ones but they are fragmented and this is the reason why the scheduler assigns less resources than the ones requested.

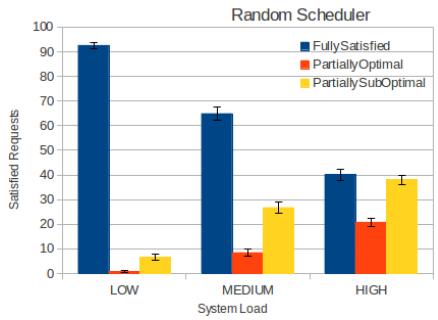


Figure 22: Random scheduler.

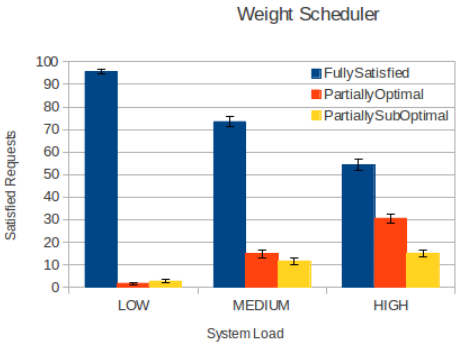
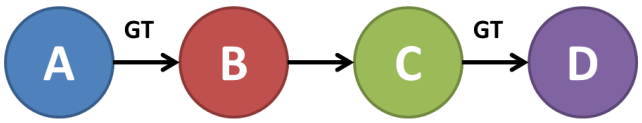


Figure 23: Weight scheduler.

As we can see, the weight scheduler (Figure 22) is significantly better than the random one (Figure 23). The weight strategy is able to reduce the fragmentation issue to reasonable values. For instance, for the worst case (HIGH load), the weight scheduler is able to optimally satisfy up to 85 grants without fragmentation (blue and orange bars) and only 15 grants with fragmentation (yellow bar). With the same system load, we can see how the random scheduler is highly inefficient and 40 grants (almost the half part) were assigned in a fragmented way.

5.3.2 QoS Support

As explained in Deliverable 2.1, the extended OpenMP programming interface developed for the vIrtical project allows to specify soft-QoS requirements as guaranteed-throughput communication channels between tasks mapped on different clusters of the GPPA. For the sake of clarity we show below the same example provided in D2.1. This example shows an application *task graph* composed of four tasks, mapped onto as many clusters and with two QoS channel requests: one between tasks 1 and 2 and one between tasks 3 and 4.



Our programming model extensions provide means to specify these QoS requirements and propagate them all the way down to the GPPA *bridge* through the **`otask`** data structure. Specifically, the `qos_channels` field stores the overall number of QoS dependencies (GT channels). The `qos` field is an array (with `qos_channels` elements) that stores integer pairs

representing the IDs of source and sink nodes for each QoS dependency. In the example above, the `qos` array has two elements, initialized as follows:

```
qos[0][0] = 1;
qos[0][1] = 2;
qos[1][0] = 3;
qos[1][1] = 4;
```

Deliverable 4.1 describes the HW support to set up a circuit between two clusters in order to guarantee exclusive access to the full NoC bandwidth provided by the links between the two clusters. This is achieved by setting a link inside the NoC (the link connecting the two switches the two clusters are connected two) as being faulty, thus not being used by the rest of clusters. However, this link is exclusively used by the two clusters attached to the link, modeling a circuit. This requires an extra addition to the routing logic at each switch input port by adding 11 extra bits to properly steer packets between the two clusters along the reserved circuit.

However, although the previous approach works and allows a straightforward solution to the establishment of QoS circuits, the previous partitions defined by the scheduling algorithm can also guarantee exclusive use of links by pairs of clusters, thus modeling a QoS circuit.

The new method consists in a special configuration of the LBDR bits of the partitions by adding new routing restrictions. The added restrictions allow the involved link to be only used by the two neighbor clusters, thus forming a one-hop circuit. Figure 24 shows the case where a 2x3 rectangular partition is configured using the LBDR extra bits (11 bits @ D4.1) while Figure 25 shows the same case but using only additional restrictions (the bidirectional arrows). Notice that the green dotted-link is disconnected (Cx bits are set to zero) and the 11 additional bits are used to set up the circuit. On the other hand, the circuit (green solid line) is modeled without the LBDR extension and, therefore, the link is not disconnected. The traffic flow traversing the QoS link is totally isolated from the other traffic flows.

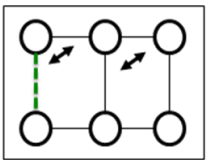


Figure 24: One circuit modeled with LBDR extensions (11 bits @ D4.1).

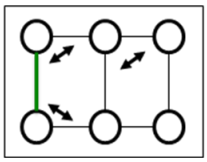


Figure 25: One circuit modeled without LBDR extensions.

Therefore the circuits can be modeled inside the partitions by proper configuring the LBDR bits. However, there are partitions that cannot guarantee any QoS circuit by definition. Figure 26 shows the case where there is no possibility to isolate the QoS traffic flow from other traffic flows. In this case, none of the solutions can set up a valid circuit.

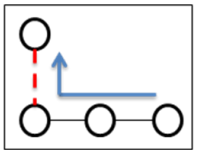


Figure 26: Non-supported QoS circuit in a 4-cluster partition.

However, let us take into account the rest of shapes that can be formed by the scheduler algorithm. Figure 27 shows the case for all the partitions that allow a link to be treated as a QoS circuit (shown in green). By properly setting the routing bits of the partition (and the deroute bits shown in dotted green arrows), the link is exclusively used by the two attached clusters. Moreover, it is also possible to include two circuits per partition as shown in Figure 28. All these combinations have been tested for connectivity and deadlock-freedom properties.

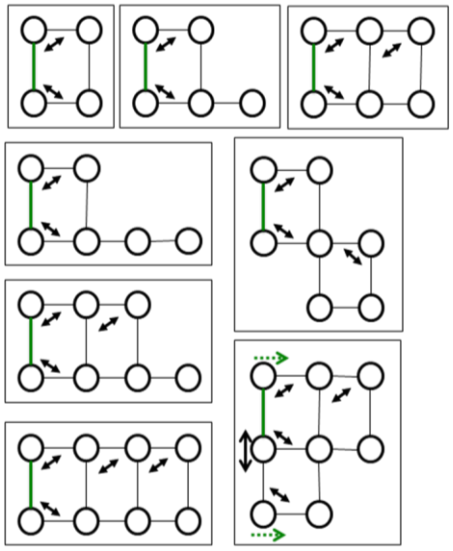


Figure 27: Valid configurations with one circuit per partition.

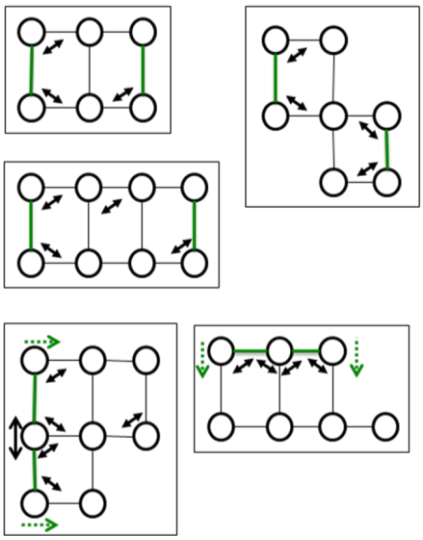


Figure 28: Valid configurations with two circuits per partition.

However, the new scheme suffers from lack of flexibility compared with the previous scheme presented in D4.1. There are few cases for particular circuit configurations where the new scheme is not able to find a valid solution. Figure 29 shows the case where the bold links must be configured as QoS circuits. In this particular case, the new scheme is unable to find a valid solution. Figure 30 shows the case where the red restriction is mandatory by definition, but at the same time impedes some traffic flows (the blue arrow). On the other hand, for this specific case, the scheme presented in D4.1 is able to find a valid solution by using the extra LBDR configuration bits (Figure 31). Therefore, the new scheme is more restrictive and not all the circuit combinations can be achieved.

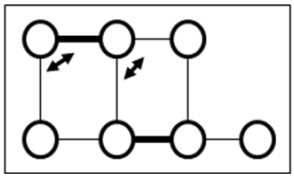


Figure 31: A particular partition.

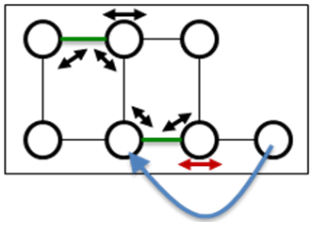


Figure 30: Unsupported by the new scheme.

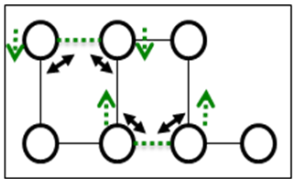


Figure 29: Supported by using the LBDR extra bits.

The algorithm, thus, can be easily extended to support the definition of QoS circuits. The LBDR bits in the DB need to be modified and the partitions that allow the establishment of circuits need to be properly flagged. By adding a new field in the DB this can be easily achieved. Also, upon a request for a partition with a QoS circuit, the algorithm needs to discriminate and to prioritize the partitions with such support. In case, no partition with the number of QoS circuits requested are available, the algorithm needs to notify it through the task descriptor. In particular, the field `qos` will be needed to both, indicate the number of required circuits and the number of granted circuits. In case of a circuit being granted, the IDs of the clusters with the QoS circuit will be named first in the `clusters_bitmask` field.

5.3.3 Task Offload Request

When a message extracted from the message queue is marked with the GPPA_OFFLOAD command different operations take place in the bridge. At first the virtual machine PID is looked-up in the Virtual Machine table to check whether the requesting virtual machine is allowed (registered) to use the GPPA. If not, an error is sent back to the *guest* using its private message queue and the request is ignored. If the offload request can be processed, a resource availability check is executed on the GPPA/Applications status table to check if there are enough resources (clusters) available to fulfill the demand of the application.

Once the requesting virtual machine is recognized by the *GPPABridge* the partition for the application is computed and, the *clusters_bitmask* and *noc_conf_bits* fields of the task descriptor are filled. After the creation of the partition the entire context of the OpenMP application is reconstructed. The binary is copied into the L2 memory of the GPPA, the bridge is in charge of triggeing the allocation and copy of the binary. Both copy and allocation are triggered via an ioctl call to the *host* physical driver. The pointer to the binary into the task descriptor is updated with the new address result of the ioctl call. After the binary, data is moved into the contiguous L3 memory space, even in this case, the bridge using ioctl calls will allocate and copy the data into L3 memory. OpenMP applications use a data structure (called *data_context*), containing the pointers to all data buffers involved in the application (private, first-private and shared data). The *data_context* is allocated in L2, and filled with the addresses of all buffers into the L3 contiguous memory. After the *data_context* is created a task ID is also generated and its value is stored in the *id* field of the task descriptor.

The last step of the offload procedure is an ioctl call (Arrow 4 in Figure 5), passing the task descriptor as parameter and the command GPPA_OFFLOAD_TASK. This last call will trigger the execution of the task on the Fabric Controller, starting from the reconfiguration of the NoC to finish starting the actual computation on the clusters indicated by the *clusters_bitmask* field of the task descriptor. The GPPA bridge waits until a confirmation value is received to notify that the offload was successful. This value is propagated back towards the application running on the *guest* (Arrows from 7 to 10 in Figure 5).

The offload procedure just mentioned is shown in the following portion of code.

```
struct gppa_data{
    void * src;
    void * dst;
    unsigned int size;

}

int GPPA_task_offload(struct otask * task){
    struct data_desc data;
    struct gppa_data d;
    int err;

    // Run algorithm to determine partition and fill-in field
    clusters_bitmask and noc_conf_bits
    run_partitioning_alg(task->num_clusters, task->qos, &task->
    clusters_bitmask, &task->noc_conf_bits);

    // Allocate memory for binary
    data.size = task->bin_size;
    err = ioctl(gppa_dev,GPPA_L2_ALLOC,(unsigned int)&data);
    if(err)
          return -1;

    //copy binary in L2 memory
```

```
d.src = task->bin_ptr;
d.dst = data->ptr;
d.size = task->bin_size;
err = ioctl(gppa_dev,GPPA_L2_COPY, (unsigned int)&d);
if(err)
       return -1;

//update pointer tp the binary in the task descriptor
task->bin_ptr = d.dst;

unsigned int * host_ctx;

int count = 0;

//allocate L2 memory for the data context
unsigned int ctx_size = task->shared_data->n_data + task->
private_data-> n_data + task->first_private->n_data;

host_ctx = (unsigned int *) malloc(ctx_size);

data.size = ctx_size;
err = ioctl(gppa_dev,GPPA_L2_ALLOC,(unsigned int)&data);
if(err)
       return -1;

//context pointer update
task->data_context = d.ptr;

//allocate and copy all the data buffers to contiguous physical
memory
for (i=0;i<task->shared_data->n_data;i++){
   data.size = task->shared_data->data[i]->size;
   err = ioctl(gppa_dev,GPPA_L3_ALLOC,(unsigned int)&data);
   if(err)
       return -1;

   //update context
   host_context[count] = data.ptr;

   d.src = task->shared_data->data[i]->ptr;
   d.dst = data->ptr;
   d.size = data->size;
   err = ioctl(gppa_dev,GPPA_L3_COPY, (unsigned int)&d);
   if(err)
       return -1;

   count++;

}

//same procedure for first_private and private data

//copy OpenMP data context to L2 memory
d.src = host_context;
d.dst = task->data_context;
d.size = ctx_size;
err = ioctl(gppa_dev,GPPA_L2_COPY, (unsigned int)&d);
if(err)
       return -1;
```

```
    //generate id for the task
    task->id = new_task_id();

    //offload the task to the gppa, this phase comprises the NoC
    reconfiguration
    int err = ioctl (gppa_dev, GPPA_OFFLOAD, (unsigned int)task);
    if (err)
           return -1;

    return 1;

}
```

5.3.4 Task Completion Check

When a message extracted from the shared POSIX queue is marked with the GPPA_TSK_END, an ioctl is done towards the physical Host GPPA driver (Arrow 4 in Figure 6) passing the ID of the application as parameter.
If the result of the ioctl is not successful, -1 is returned to the virtual machine, a positive value otherwise. In case the result is negative a new thread is spawned (Arrows from 5 to 12 in Figure 6) which at regular time intervals will check for application completion. When the application terminates a message is sent though the private message queue of the application.

The notification of completion arrives asynchronously from the fabric controller, that when a task on the GPPA completes, it notifies the *host* with the ID of the application using an interrupt (Arrow *a* in Figure 6). The interrupt handler inside the GPPA physical driver reads an internal register of the GPPA to know the ID of the task which has just completed its execution (Arrow *b* in Figure 6).

5.4 GPPA Host Driver (gppadriver)

This is the Linux driver which actually communicates with the real GPPA. An ioctl interface is defined, exposing the following functions:

- *Task Offload* (GPPA_TSK_OFFLOAD),
- *Task Completion check* (GPPA_TSK_END).

The GPPA is only capable of accessing contiguous physical memory, thus it is necessary to define a contiguous physical memory region of the External memory to be shared between the Host and the GPPA. This physical memory region is used to store binary and data of applications.
During the Host system boot, a subset of the entire external memory is reserved and will not be used by the Host Linux kernel under virtual memory. During GPPA driver initialization the reserved memory area is then remapped into the kernel space using the ioremap system call.

Task Offload Request

When a task offload request arrives through the ioctl interface (arrow 4 Figure 5), the pointer to a task descriptor is passed as a parameter. The driver then uses the copy_from_user system call to copy the descriptor into the kernel space.
All information needed for the offload operation is already present into the task descriptor, binary and data of the task are already allocated in the L2/L3 memory space of the GPPA. Copies are triggered by the *GPPA bridge* via an ioctl call and executed by the *Host Linux driver.* Each copy procedure is implemented using copy_from_user to read the data from user-space and iowrite to write them into the destination memory space (L2/L3).

The real offload is executed by copying the task descriptor into the local memory of the fabric controller via iowrite calls. The fabric controller is then in charge of reconfiguring the GPPA NoC and scheduling the computation on the cluster specified in the field *clusters_bitmask* of the task descriptor. The *GPPA Host driver* waits for an acknowledgement from the Fabric controller to be sure that the offload is successful. A negative value is propagated back in case of error (arrow 7 Figure 5).

The following portion of code shows an abstract implementation of the ioctl method of the GPPA Host Driver, focusing on the task offload and GPPA memory management.

```
int gppadriver_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg) {

..

case GPPA_L2_ALLOC:
    //allocate GPPA L2 memory
    struct data_desc * data = (struct data_desc otask *)arg;
    task->ptr = gppa_L2_malloc(task-> size);

    if(!task->ptr)
          return -1;
    break;

case GPPA_L2_COPY:
    //copy data from Host Virtual Memory to GPPA L2 memory
    struct gppa_data * data = (struct gppa_data *)arg;
    int err = gppa_copyto_L2(data->src,data->dst,data->size);
    if(err)
```

```
            return -1;
    break;

case GPPA_L3_ALLOC:
    //allocate GPPA L3 memory
    struct data_desc * data = (struct data_desc otask *)arg;
    task->ptr = gppa_L3_malloc(task-> size);

    if(!task->ptr)
            return -1;
    break;



case GPPA_L3_COPY:
    //copy data from Host Virtual Memory to GPPA L3 memory
    struct gppa_data * data = (struct gppa_data *)arg;
    int err = gppa_copyto_L3(data->src,data->dst,data->size);
    if(err)
            return -1;
    break;









case GPPA_OFFLOAD;
    //offload a task to the gppa
    struct otast * task = (struct otast *) arg;
    int err = gppa_offload (task);

if (err)
             return -1;
    break;
...
}
```

Task Completion Check

In case of task completion check request, the ID of the application to be checked is passed as parameter of the ioctl call (Arrows 1 and 8 in Figure 6). The driver uses this value to lookup into its internal tables if the application is finished.
A positive value is returned in case the task has already finished and a negative value indicates otherwise.

6 Conclusion

An implementation of KVM for the ARM Cortex-A15 architecture has been implemented and has also been included in the official Linux kernel releases since version 3.9. The open source hypervisor implementation includes support for the hardware virtualization extensions present in the Cortex-A15, as well as other architectural improvements for interrupt virtualization in the ARM Generic Interrupt Controller.

The KVM on ARM implementation is mature and stable, and can be used with user space drivers such as QEMU, which couple the processor virtualization of KVM with virtual device emulation, or device paravirtualization features, i.e. Virtio. In the future KVM may be extended with more advanced features, such as device pass through utilizing IOMMU hardware.

A vertically integrated stack has been implemented for the virtualization of the GPPA device. The proposed stack is based on a bridge process (GPPA bridge) residing in the user-space of the *host*, and collecting offload requests coming from different *guests*. GPPA resource sharing is based on NoC partitioning to create distinct, isolated subsets of computation clusters with local memory. A virtual device introduced into each QEMU Guest allows each virtual machine to have its independent view of the GPPA. Data sharing between GPPA and the *host* system is enabled thanks to a set of copies from the *guests* virtual memory to a contiguous physical memory space in main DRAM memory.

The GPPA NoC partitioning algorithm implemented in the *GPPA bridge* allows multiple tasks to run in parallel in isolated partitions of the GPPA. The algorithm is designed to minimize the complexity of the algorithm itself and the fragmentation in the GPPA NoC. Also, one-hop circuits can be established and selected by the scheduling algorithm.

7 Bibliography

1. *QEMU, a fast and portable dynamic translator.* **Bellard, Fabrice.** 2005. USENIX.