
# Interactive Realtime Multimedia Applications on Service Oriented Infrastructures

## ICT FP7-214777

## WP 6 Execution Environment

## D6.4.1 Initial Version of Realtime Architecture of Execution Environment

## IRMOS_WP6_D6_4_1_SSSA_V1_0

**Scheduled Delivery:** 31.01.2009
**Actual Delivery:** 30.01.2009
**Version:** 1.0

| Project co-funded by the European Commission within the 7th Framework Programme | | |
|---|---|---|
| Dissemination Level | | |
| PU | Public | X |
| PP | Restricted to other programme participants (including the Commission) | |
| RE | Restricted to a group specified by the consortium (including the Commission) | |
| CO | Confidential, only for members of the consortium (including the Commission) | |


## Responsible Partner: Scuola Superiore Sant'Anna (SSSA)

## Revision history:

| Date | Editor | Status | Version | Changes |
|---|---|---|---|---|
| 24.06.2008 | Tommaso Cucinotta | Draft | 0.1 | ToC Draft |
| 03.07.2008 | Tommaso Cucinotta | Draft | 0.2 | Refined ToC Draft |
| 10.11.2008 | Tommaso Cucinotta | Draft | 0.3 | Integrated partners first contributions |
| 13.11.2008 | Tommaso Cucinotta | Draft | 0.4 | Integrated first DTO contribution |
| 05.12.2008 | Tommaso Cucinotta | Draft | 0.5 | Integrated partners 2nd contributions |
| 19.12.2008 | Tommaso Cucinotta | Draft | 0.6 | Integrated comments from WP6 Audio Conf |
| 07.01.2009 | Tommaso Cucinotta | Draft | 0.7 | Integrated comments by ALUD and USTUTT and improved formatting |
| 12.01.2009 | Tommaso Cucinotta | Draft | 0.8 | Minor changes |
| 27.01.2009 | Tommaso Cucinotta | Draft | 0.9 | Integrated QA reviewers comments |
| 29.01.2009 | Tommaso Cucinotta | Draft | 0.10 | Style issues solved |
| 30.01.2009 | Tommaso Cucinotta | Final | 1.0 | Style issues notified by QA coordinator solved |

## Authors


Tommaso Cucinotta (SSSA)
Gaetano Anastasi (SSSA)
Fabio Checconi (SSSA)
Kleopatra Kostanteli (NTUA)
Antonio Cuevas (USTUTT)
Dominik Lamp (USTUTT)
Manuel Stein (ALUD)

Lars Fuerst (ALUD)
Thomas Voith (ALUD)

## Internal Reviewers

Eduardo Oliveros (TID)
George Kousiouris (NTUA)
Stefan Wesner (USTUTT)

## Copyright



## Acknowledgements

The research leading to these results has received funding from the EC Seventh Framework Programme FP7/2007-2011 under grant agreement n° 214777.

## More information

The most recent version of this document and all other public deliverables of IRMOS can be found at http://www.irmosproject.eu

# Glossary of Acronyms

| Acronym | Definition |
| --- | --- |
| APIC | Advanced Programmable Interrupt Controller |
| ASC | Application Service Component |
| BIOS | Basic Input/Output System |
| BVT | Borrowed Virtual Time |
| COTS | Common Off-The-Shelf hardware |
| CPU | Central Processing Unit |
| D | Deliverable |
| DAS | Direct-Attached Storage |
| EDF | Earliest Deadline First |
| EE | Execution Environment |
| GPL | General Public License |
| GPU | Graphics Processing Unit |
| IRMOS | Interactive Realtime Multimedia Applications on Service Oriented Infrastructures |
| ISONI | Intelligent Service Oriented Network Infrastructure |
| IXB | ISONI eXchange Box |
| KVM | Kernel-based Virtual Machine |
| NAS | Network-Attached Storage |
| OAM | Operation, Administration and Maintenance |
| OS | Operating System |
| OSD | Object-based Storage Device |
| PCI | Peripheral Component Interconnect Bus |
| PH | Physical Host (ISONI) |
| PIC | Programmable Interrupt Controller |
| PIT | Programmable Interval Timer |
| POSIX | Portable Operating System Interface for uniX |
| QoS | Quality of Service |
| SAN | Storage Area Network |
| SC | Service Component |
| SCSI | Small Computer System Interface |
| SEDF | Simple Earliest Deadline First |
| SLA | Service Level Agreement |
| SOI | Service-Oriented Infrastructure |
| SOA | Service-Oriented Architecture |
| SMP | Symmetric Multi-Processing |
| T-SLA | Technical Service Level Agreement |
| UML | User Mode Linux |
| VCPU | Virtual CPU |
| VM | Virtual Machine (synonym of VMU) |

| Acronym | Definition |
|---|---|
| VMU | Virtual Machine Unit (synonym of VM) |
| VMM | Virtual Machine Monitor, a.k.a. Hypervisor |
| VSN | Virtual Service Network |
| WP | Work Package of the IRMOS Project |

# Table of Contents

# List of Figures

# Executive Summary

This document provides an overview of how real-time service guarantees will be provided within the IRMOS platform and architecture. Even if the overview generally spans across the multiple technological areas of processing, storage and networking, the focus is primarily on the first one. The other two areas will be covered in further deliverables in WP6 and WP7, respectively (precisely, D6.7.1 and D6.7.2 will deal with QoS aware storage, while D7.3.1 and D7.3.2 will deal with QoS support at the network level).

Specifically, the document focuses on issues related to the provisioning, within an IRMOS Execution Environment (EE), of appropriate performance guarantees to concurrently running Service Components (SCs). The discussion spans from the motivations at the base of the need for mechanisms that ensure predictable scheduling of the available computing elements within an EE, as driven by the requirements analysis performed in WP2, to the technical details about how such mechanisms may be provided within the project, along with what are the research challenges arising from the intermixing of virtualization and real-time.

# 1 Introduction

The IRMOS project will provide the necessary infrastructure for allowing hosted services and applications to run with predictable levels of Quality of Service, e.g., in terms of throughput and latency, while allowing providers to share physical resources among the multitude of applications that will be instantiated at run-time.
Contrary to what is increasingly happening in the world of distributed computing, where either only best-effort performance is granted, or service levels are granted only within the very limited scope of a single service or resource (think to what happens nowadays when accessing compile farms or virtual machines), the IRMOS project will enable users and providers to establish well-defined Service-Level Agreements (SLAs) in which an important part will be constituted by the agreed level of performance that will be guaranteed by the platform on an end-to-end basis.
To this purpose, it is essential that all the elements that are glued together constituting an IRMOS platform, like computational nodes, network links and storage boxes, be capable of providing temporal guarantees to individual activities, whenever the underlying physical resources are shared across multiple applications and users.

## 1.1 Objectives

This document mainly reports about activities carried on in Task 6.4 (Schedulability of real-time services and admission control) and Task 6.5 (Real-time enabled Execution Environment) of WP6.
Purpose of Task 6.4 is to design a global resource management and allocation policy for scheduling services enhanced with real-time attributes. Such allocation policy has to take into account the nature of applications hosted in IRMOS, based on composition of service components in a workflow, and the timing requirements they have.
Task 6.5 has the purpose of providing an implementation of selected mechanisms for resource management at the Operating System level that are needed in order to guarantee the timing requirements of IRMOS applications.
Both tasks constitute essential bricks in the construction of an Execution Environment that is enriched with real-time capabilities within the IRMOS platform, and specifically within the ISONI infrastructure, for the purpose of supporting deployment of computation-intensive services and applications that exhibit real-time (or simply highly predictable) performance.

## 1.2 Relationship with other WPs

The relationships between this deliverable and the other ones produced within the IRMOS project may be summarized as follows (see Figure 1):
*Inputs*

- **from D2.1.1** (Initial version of requirements analysis) and **ID2.1.2** (Updated version of requirements analysis): the analysis of the timing and real-time requirements on the IRMOS platform is the basis for understanding what type of mechanisms are needed, which are described in this deliverable; being ID2.1.2 an internal deliverable, the minimum information that is necessary for comprehension of the present document has been briefly summarised in Section 2;

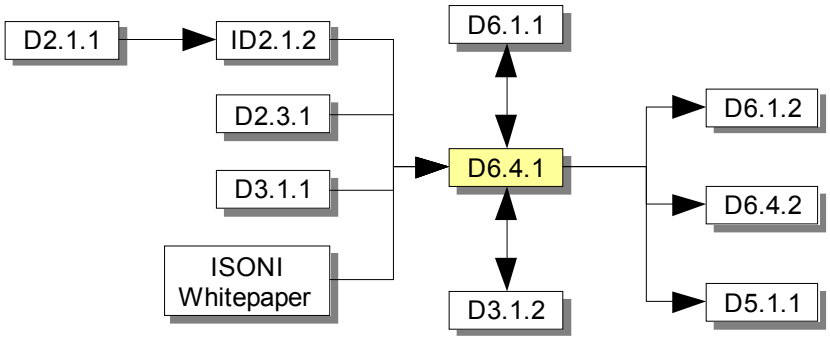


**Figure 1. Relationship between this document and other IRMOS deliverables.**

- **from D3.1.1** (Initial IRMOS Overall Architecture): the general Service Oriented Architecture (SOA) architecture of IRMOS is needed for identifying what activities or SOA-related software components are expected to be concurrently running on each IRMOS physical node;
- **from D2.3.1** (State of the art on IRMOS technologies) and **D9.2.2** (State of the art of standards relevant for IRMOS): the current state of the art in technologies and standards related to IRMOS is appropriately taken into consideration in this document;
- **from ISONI white-paper**: it is important to identify what software components need to be deployed on physical nodes and/or within EEs;

*Outputs*

- **to D6.1.2 (EE Prototype)**: the initial prototype of the IRMOS Execution Environment will include some of the mechanisms identified in this deliverable;
- **to D6.4.2 (Final version of Real-time architecture of EE)**: this will constitute a refinement of this document contents;
- **to D5.1.1 (Models of real time applications on service oriented infrastructures)**, or subsequent deliverables: the way framework services in WP5 will perform benchmarking and monitoring activities may be affected by the discussion of the real-time platform capabilities made in this deliverable;

*Interactions*

- **from/to D6.1.1 (Formal description language for application requirements of EE):** the way real-time capabilities and requirements may be formalized at the service ontology level is affected by what parameters are needed by the underlying mechanisms for QoS provisioning and temporal isolation;
- **From/to D3.1.2 (Initial version of IRMOS Overall Architecture)**: the overall IRMOS architecture description will take into account real-time capabilities of the Execution Environment, as outlined in this document.

## 1.3 Document organization

This document is organized as follows. Section 2 recalls fundamental requirements that deal with timing issues, as identified during the requirements analysis tasks performed in WP2, highlighting how the need for real-time support arises within IRMOS. Section 3 performs a preliminary overview of virtualization in IRMOS, because the way virtualization is done is crucial to designing appropriate real-time mechanisms within

the Execution Environment. Section 4 briefly summarizes what are the activities that are envisioned to be running within the Execution Environment, on an IRMOS Physical Host. These must be properly considered while designing the overall real-time and QoS support to Service Components (SCs) running on the platform, in order to ensure that timing guarantees of SCs are not negatively or unpredictably affected by the presence of the other activities that are essential for running the IRMOS platform. Section 5 deals with scheduling mechanisms that must be embedded in the IRMOS platform in order to provide appropriate guarantees to the hosted services. This is analysed both from the high-level workflow-aware perspective, and from the low-level, workflow-unaware perspective typical of the Operating System. Finally, conclusions are drawn in Section 6.

# 2 Real-Time Requirements

The IRMOS general user requirements have been analysed in the WP2 D2.1.1 deliverable [3], while technical platform requirements have been defined in the WP2 ID2.1.2 deliverable [4]. The latter is an internal deliverable, however, for the purpose of clarity, the requirements related to real-time performance of the platform are briefly summarised in what follows. Afterwards, the document will present an overview of the mechanisms that may be used in order to fulfil these requirements.

In particular, real-time requirements mainly belong to the technological areas of the Execution Environment (EE), Network and storage (NET), as summarised below. Please, note that QoS guarantees for storage and networking will be covered in detail by later deliverables of WP6 for storage (for example, the ones associated to Task 6.7) and WP7 ones for networking. Also, note that some of the requirements shown below mainly affect the platform-user interactions, which in turn imply requirements on the platform itself, of course.

- **EE-1 Real-time deployment**: the IRMOS platform needs not only to ensure that the services deployed in it are running under real-time constrains, but also that the mechanism that deploys these services is real-time aware and therefore the deployment, instantiation and availability of the required underlying resources comply with possible real-time constraints;
- **EE-9 Dynamic Context**: the IRMOS platform should provide the needed functionality to run applications and services in a dynamic context, where multiple application work-flows with interactive timeline requirements may be started and stopped dynamically at will, provided that the system is capable to guarantee the necessary QoS levels;
- **EE-12 QoS in Execution Environment and storage**: the IRMOS platform provides means to request and guarantee the real-time requirements imposed by the deployed services; this is expressed in the form of QoS in terms of CPU usage, minimum memory access times, storage access times, storage access bandwidth, and so forth;
- **NET-6 Provide edge-to-edge QoS for soft real-time services**: the IRMOS platform should provide QoS in the transport network, which connects the service components of a virtual service network located at potentially different locations within the borders of the physical network(s) under control; QoS is defined by parameters like bandwidth, delay, jitter and others;
- **NET-7 Maximum jitter specification**: an IRMOS user may specify the maximum acceptable jitter of a specific streaming. It should be a target for the SLA established at the platform-user interface (implying requirements on the platform);
- **NET-8 Maximum delay specification**: an IRMOS user may specify the maximum acceptable delay of a specific streaming. It should be a target for the SLA established at the platform-user interface (implying requirements on the platform);
- **NET-9 Maximum latency specification**: an IRMOS user may specify the maximum acceptable latency of a specific streaming. It should be a target for the SLA established at the platform-user interface (implying requirements on the platform);
- **NET-10 Minimum throughput specification**: an IRMOS user may specify the minimum acceptable throughput of a specific streaming. It should be a target for the

established SLA at the platform-user interface (implying requirements on the platform);

- **NET-12 Collaborative Work**: the platform should provide the appropriate API and interfaces to support collaborative work to be executed through IRMOS; This includes collaborative tools shared by all the participants (e.g., shared whiteboard) and must be real-time enabled;
- **NET-14 Other Real-Time related Requirements**: this requirement points out to general real-time requirements that include interactivity, audio/video quality, short response-time and high throughput;
- **VS-3 Video and Audio Streaming/Multi video-conference**: besides of the previous requirements, which address explicitly real-time aspects, this other requirement indirectly refers to timing constraints, in fact it states that "video quality is guaranteed".

In order to satisfy the above summarised requirements, proper mechanisms have to be adopted in the heterogeneous areas of networking, processing and storage. These mechanisms must be capable of providing precise QoS guarantees to the individual provisioned service components. The rest of this document discusses what kind of mechanisms might be adopted for this purpose.
In the following sections, this document focuses mainly on the area of QoS guarantees in the domain of EE computing, remanding to the future deliverables mentioned above for QoS support at the storage and networking level.

# 3 Real-time and virtualization

An important feature of the envisioned architecture for IRMOS is the capability to seamlessly migrate running services across physical nodes. This will be achieved by using appropriate virtualization techniques.
Virtualization basically refers to the technology that allows a system to host one or more emulated other systems, called Virtual Machines (VMs), which may also be seamlessly migrated across physical hosts. Note that, within this document, the terms Virtual Machine (VM) and Virtual Machine Unit (VMU) will be used interchangeably.
Such feature is useful not only for purposes related to deployment of services, but also for allowing for a dynamic reconfiguration of the system in case of failures (at the hardware, OS, middleware or software component levels) or in case an SLA violation is predicted, in a way that is transparent to running applications. This requires some activities to be performed continuously during a service execution, for the purpose of transmitting to a backup node information that is sufficient to restore the service execution from the backup location, in case the first instance undergoes an unrecoverable failure.
Therefore on each IRMOS physical node, the presence of additional software components that might have timing interferences with the running services is required. This aspect needs from a real-time perspective particular attention. The following brief overview outlines how virtualization is foreseen to be used within the IRMOS platform, giving special attention to which features are required in order to support real-time performance of the hosted services.

## 3.1 General requirements on the virtualization layer

The IRMOS project aims for fulfilling partly contradictory requirements simultaneously.
From a customer's perspective, the canonical way to gain highly deterministic performance is to deploy the application components to dedicated hardware. Unfortunately, this turns out to be quite expensive, especially in situations where actual resource usage is low and/or the service is used for only a short period of time.
Another drawback of this approach is that the customer has to install and maintain the server Operating System (OS) and also has to perform the deployment of the actual software. The IRMOS infrastructure will provide a more convenient framework for such tasks to the user.
On the other hand, due to the typical presence of workloads that lead to a low saturation of the resources, service providers would be interested in selling the same resources, e.g., servers, simultaneously to multiple customers. Virtualization technologies fulfil perfectly such a need by allowing shared use of the same physical resources across multiple (even heterogeneous) concurrently running Operating Systems (OSes). Unfortunately, this causes *temporal interferences* across OSes running on behalf of different users and applications, making SLAs with guaranteed service levels more difficult to be established/negotiated (in terms of technical SLA parameters, service and costs and violation penalties) and managed. Consequently, appropriate *temporal isolation* mechanisms are needed in order to guarantee a deterministic and predictable behaviour of individual virtualized environments.

A first set of basic requirements that need to be fulfilled by the virtualization layer have been identified as follows:

- **Deterministic Processing**: IRMOS applications are used to deliver to the final user real-time services, i.e., services with well-defined interactivity or performance constraints as defined in the established SLAs. Therefore, the timing behaviour of each application must be deterministic at a certain extent. This requires deterministic behaviour of the underlying virtualization layer when scheduling each Virtual Machine Unit (VMU).
- **Service Isolation**: SLAs between the provider and the customer must also cover guarantees e.g., about data confidentiality. This means for example that crosstalk between VMUs provided to different customers must be prevented, unless explicitly requested by them.
- **Temporal Isolation:** The virtualization layer must guarantee that failures (including system level failures) in one virtualized environment do not affect other virtualized environments. Isolation must be guaranteed not only at a functional level, but *also at a non-functional one,* i.e., in relation to timing behaviour and performance of hosted services. This means that, if one of the VMUs exhibits temporary unforeseen overloads (comprising possible buggy behaviours such as due to infinite loops, deadlocks, etc...), this should only affect performance of that VMU, without affecting performance or temporal behaviour of other customers VMUs.
- **Relocatability:** As hardware failures or hardware maintenance must be invisible to the IRMOS customers, seamless relocation of running VMUs to other Physical Hosts within the platform must be supported. Also, VMU images must be independent of a concrete physical host, as their availability might change between deployment and instantiation of a Virtual Service Network (VSN) [1]. Reliability must be ensured, recurring to fallback mechanisms in case of failures.
- **Deployment support**: To ease deployment for the VSN developer, the process of creating VMU images, including installation of the Operating System and deployment of the actual service, should be automated. The process must be able to guarantee that:
  - the VMU creation process does not affect performance of running VMUs: the computational resources required by a VMU while booting (or shutting down) may be higher than the ones required during normal operation, thus appropriate scheduling mechanisms need to be used so as to not affect negatively performance of other VMUs that may already be running on the Physical Host (this relates to the isolation requirement stated above);
  - whenever a new VSN needs to be started, the processes of creating and transferring the VMU images to the Physical Hosts on which they have been allocated, as well as of booting up the VMUs and establishing connectivity with the desired QoS levels, need to be completed within the scheduled VSN start-up time.
- **Easy administration of host and guest domains:** These will be further detailed in future WP6 deliverables on virtualization, namely D6.2.1 and D6.2.2:
  - Easy installation and configuration of host and guest domains
  - Automatic start-up of guest domains after reboot of the host system
  - Graphical and non-graphical monitoring of guest domains

  - Graphical and non-graphical access to guest domains
  - RAM extensions of guest domains during runtime
  - Support for various guest disk types: physical disk device, disk image file, network block device.
- **Access to network of virtualized environment from physical hosts**: Network traffic from and to the virtualized environment must be visible and controllable from the Physical Host that is hosting the VMU. The virtualization layer must support (or needs to be flanked by mechanisms that support) the secure forwarding of packets between virtualized environments, as well as controlling access to the outside world (network) from within the VMUs.
- **Persistence**: Some components might not be required for a certain timeframe in the embedding VSN's lifetime. To free resources during that time, the virtualization layer needs to be able to save and restore virtualized environments.
- **Hardware access:** The virtualization layer should be able to provide direct hardware access to virtualized environments, while guaranteeing that this cannot be abused by malicious VMUs. This is useful to allow for offloading of computational intensive tasks to dedicated and optimized hardware, e.g., DSP add-on boards or Graphics Processing Units (GPUs).
- **Source code availability:** Within the IRMOS project, it may become necessary to change some internal characteristics of the virtualization layer, for example concerning the support for live migration. Therefore, access to the source code of the virtualization software may be needed, specifically for the activities foreseen in some research tasks of WP6 (e.g., Task 6.2 and Task 6.3).

Most of the requirements stated above will be further explained and detailed in future virtualization-specific WP6 deliverables (i.e., D6.2.1), whilst the requirements this document mostly focuses on concern the deterministic processing and temporal isolation properties. This subsumes to requesting that the virtualization layer be capable to provide a scheduler between multiple VMUs concurrently running on the same Physical Host that:

- works stable and reliable on single and multi processor systems;
- distributes the total available CPU time to the different virtual machines, guaranteeing that each VMU is granted a dedicated time period which was previously configured;
- guarantees that a VMU with a high CPU intensive task does not negatively affect the performance of other VMUs concurrently running on the same Physical Host;
- allows for changing the scheduler settings during runtime (while the VMUs are running): this may be useful, for example, for assigning different scheduling parameters at boot time than at actual VSN run-time, for a VMU.

### 3.1.1 Virtualization in the ISONI infrastructure

In the IRMOS general architecture, a fundamental role is played by the Intelligent Service-Oriented Network Infrastructure (ISONI), that automates the processes of SLA negotiation, resource selection, resource booking and configuration, as well as creation, configuration, deployment and execution of VMUs. These processes require resource availability information, resource allocation policies and real-time enhanced VMU scheduling from the virtualization environment of the ISONI infrastructure throughout the life-cycle of a VSN.

### SLA negotiation / VSN request

Resource selection and scheduling at the ISONI Domain and Node level use the VSN description [1] as a formal description of the requested resources and to derive the admission policy for deployment of a VMU. Resource availability reporting is used to perform resource selection at each level of the ISONI architecture. ISONI resource allocation policies regulate VMU scheduling and resource sharing on the available resources in the infrastructure.
To support these tasks, the virtualization layer of ISONI is required to allow for resource metering, admission policy enforcement and VMU scheduling.

### VMU Factory: Creation and configuration

ISONI provides a tailored Execution Environment for the Service Components (SCs) of a Virtual Service Network [1]. Therefore, ISONI configures the VMU according to the negotiated SLA for virtualized resource admission (process scheduling, network admission control, file system access). Among the ISONI middle-ware functional blocks, it is the VMU Factory that tailors the operating system and the Service Components to be ready for deployment on a Physical Host (see [5], Section 3.1). The Execution Environment has to meet the requirements settled during SLA negotiation, i.e., the VSN description specifies the virtual machine resource requirements and OS environment, whereas the Service Component description contains the library dependencies, configuration and admission policy of those software modules that provide the Service Component specific functionality.
The VMU Factory is capable of automating the task of setting up the virtualized hardware environment (in terms of the requested resources, e.g., CPU type, block devices, network interfaces) needed as a preliminary step for running a Virtual Machine Unit. The VMU Factory might require dedicated resources and hardware emulation from the platform to create and configure the VMU prior to its deployment or it might decide to schedule the VMU preparation directly at a spare time slot on the selected Physical Host.
A VMU set-up requires an automated installation/configuration routine of the selected OS. Depending on the VMU factory realization, the OS installation procedure might require: a) virtualization of images as removable devices that are only available during installation, b) a pre-boot execution environment (PXE) during start-up of the virtual machine that receives the OS installation configurations for network installation from an installation repository, or c) template OS images that allow for VMU replication plus a further adaptation step based on individual VMU requirements, as detailed in the VSN description.
Regarding the **Service Components** to be installed and configured in the prepared VMU, a SC description specifies the library dependencies to be installed along with the software module resembling the SC. Therefore, the VMU factory requires access to the virtual machine that allows for software installation on the previously configured virtualized OS. Depending on the realization of the VMU factory, this might require from the virtualization layer: a) remote access to the VMU (e.g. console access), b) VMU start-up with resource assignments that have been modified for configuration and execution purposes, c) file transfer to install the SC software modules and configuration files, and/or d) virtualization of temporarily attached storage and network resources.

### Deployment and runtime

Upon scheduled **deployment** of a VMU on an ISONI Physical Host, the VMU is started and provisioned with its assigned resources, admission policy control is set-up and status supervision enabled.
The virtualization layer shall provide and assign the requested virtualized resources to the VMU. The real-time enabled resource virtualization shall comprise variable memory reservation, appropriate network interfaces, CPU-time scheduling, access to Storage Area Network (SAN) and temporal local block storage. Resource scheduling mechanisms are required for admission control of each real-time enabled virtualized resource in order to be shared by the virtual machines. Some devices might not be shareable and are hence recommended to be fully assignable to a single virtual machine (e.g. GPUs assigned via PCI pass-through).
Furthermore, retrieval of VMU images and scheduled execution of VMUs is required. For an efficient start-up of the VMU at the Physical Host, virtual machine hibernation is recommended.
During **runtime** of the VMU, the virtualization layer shall meter resource usage of the virtualized resources for VSN monitoring and infrastructure supervision. The CPU-time scheduler shall allow for multiple scheduling strategies to run stable and reliable on single and multiprocessor systems, so that no interference occurs among virtual machines with different CPU usage requirements.
The ISONI middleware eventually requests migration of the virtual machine onto another Physical Host for administrative or fail-safety purposes. The virtualization layer is required to support live migration of running virtual machines with a negligible downtime. The live migration algorithm shall allow fallback in case that a migration is not feasible without a noticeable downtime of the virtual machine. In fact, the failure of one virtual machine must neither affect the Physical Host system nor the execution of other virtual machines on the very same host.

### Networking

Special attention is given to the network resource virtualization in IRMOS. The ISONI addressing scheme [6] developed for VSN isolation through an address namespace concept poses several requirements on the virtualization layer.
Network virtualization shall allow for manipulation of the virtual network traffic in order to let ISONI apply the address namespace concept before the packet is transmitted. When delivered to an ISONI eXchange Box (IXB) within a Physical Host, the traffic shall uniquely identify machine VMU. Information concerning virtual level-two networking (L2) is recommended to be preserved, and virtual L2 bridging and policy-based routing are required from the IXBs. Enhanced traffic scheduling at the IXB Node requires transparent interception of virtual links to apply queuing and filtering disciplines to the packet-oriented transfer. For network redundancy, interface bonding of virtual network interfaces is required.
One special requirement, that crosses “vertically” the networking/computing barrier across an IRMOS EE node, is that the EE architecture should be capable of supporting the negotiated transfer bandwidth for virtualized network traffic between VMUs and the network interface cards. For no reasons computing resource shortages due to e.g., a single VMU overloading the node CPU(s), should be capable of causing unforeseen packet drops on the other VMUs or on the Physical Host hosting them.

## 3.2 Virtualization Layers

There are many possible approaches to virtualization that differentiate among each other for a number of technical details related to how hardware and peripherals are emulated, if the guest Operating System is aware or not of running in a virtualization environment (a.k.a., para-virtualization and full-virtualization, respectively), and if they exploit special capabilities of modern CPUs architectures designed specifically for virtualization. There are also attempts to virtualize by creating, within a single OS, multiple "namespaces" or "containers" that are capable of isolating multiple sets of processes from each other, however such an approach looses the capability to have multiple OSes hosted on the same hardware. All of these approaches deal in different ways with virtualization at the various levels (e.g., special CPU instructions and processor privilege levels, networking adapters virtualization, disk access virtualization, etc...). Therefore, they exhibit different virtualization overheads and different achievable computation and I/O performance figures of the guest machine(s). A full description of the approaches is out of the scope of this deliverable, and can be found for example in [7], [32], [33] and [35] (an interesting survey [34] also exists that dates back to 1974).
The various existing virtualization layers (a.k.a. Virtual Machine Monitors), exhibit a range of availability and licensing options, from completely open-source and free-of-charge use, to completely commercial. A non-exhaustive list may be the following[1]:

- Xen, by Citrix Systems;
- Kernel-based Virtual Machine (KVM), a project currently founded by Qumranet, a technology start up now owned by Red Hat;
- VirtualBox, by Sun;
- VMware Workstation, by VMware Inc.;
- VirtualPC, by Microsoft;
- coLinux;
- QEMU;
- User-Mode Linux (UML);
- OpenVZ / Vservers for Linux (OS-level Virtualization).

However, due to the objectives of the IRMOS project and requirements as identified in the previous section, the list of candidates may be easily narrowed down.
The need for possible modifications to the virtual machine monitor at the source code level, as well as the need for having an understanding of their internals as deep as it can be, excludes the closed/source commercial solutions. This includes Microsoft VirtualPC and various VMWare products, that, even playing a significant role on the virtualization market, would limit too much the freedom in the research that can be done within IRMOS concerning low-level mechanisms internal to the virtualization layer architecture.
We excluded virtualization technologies that did not possess a support for x86 like CPUs for the host and/or virtualized platforms, since the target of IRMOS is to run on common off-the-shelf (COTS) hardware.

[1] An impressive comparison chart among features of nearly 50 existing virtualization layers may be found on Wikipedia at the URL: http://en.wikipedia.org/wiki/Comparison_of_virtual_machines.

Furthermore, solutions without a support for live migration of virtual machines have been ruled out, because live migration constitutes a key capability for realizing the work in Task T6.2 Live Migration, Redundancy and Fault Tolerance.
The high performance requirements of the application domains that the IRMOS project is aiming at, rule out the class of fully virtualized approaches. Solutions like Bochs or plain QEmu would provide considerably less performance than the other available solutions.
We ruled out platform virtualization technologies that restrict to a specific guest operating system, since IRMOS has the goal to support a wide range of existing service network applications, possibly running on a multitude of server-class Operating Systems, to be migrated onto the Intelligent Service Oriented Network Infrastructure. This rules out also OS-level virtualization approaches, like OpenVZ or Vservers for Linux or similar extensions available under Sun Solaris or BSD systems.
This elimination process basically leaves out only the first two solutions in the above list, Xen and KVM, for evaluation in the context of IRMOS prototype development, thus these two solutions have been carefully evaluated within the IRMOS project. Both solutions have been found to fulfil the requirements on the virtualization layer as identified in Section 3.1, and moreover they are open-source VMM layers that are widely used, with a wide user base, and a good documentation concerning not only usage, but also internals. Being open-source projects, a major strength is also constituted by the wide open communities residing behind these two projects that may easily be contacted for issues related to possible development directions. A brief, non-exhaustive description of the Xen and KVM capabilities follows.

### 3.2.1 XEN

Xen [7] is a *hypervisor* that originally has been developed to support *paravirtualized* guests, i.e., OSes that are aware that they are running not directly on bare metal hardware but on a virtualized platform. This enables the guest OS to use specialized drivers and avoids the need to emulate I/O-Hardware, resulting in better performance compared to a full virtualization approach[2].
In the meantime, it has been extended by Intel to support unmodified guests by using hardware virtualization support ("Xen-HVM").
Xen has been developed at Cambridge University by a team of researchers led by Ian Pratt[3] and ships with all major Linux distributions. Commercial support is available by XenSource, a spin-off company that is now owned by Citrix as well as through independent consultants and the Xen Community.

#### Architecture

Xen consists out of two components (see Figure 2 below), the Xen hypervisor and a privileged guest, called *Domain Zero* (or Dom0). The hypervisor manages system start-up, scheduling, memory management, and hardware access. Device drivers are run in Dom0, which is also responsible for guest management. The privileged guest also provides network and storage backends, i.e., the components that are connected to the frontends that run inside the guest domains, i.e., the paravirtualized drivers.

---

[2] On the other hand, *full virtualization* means the capability to virtualize an OS without the need for any special modification to it, so it has the advantage of being completely transparent to the hosted OS.
[3] More information at the URL: http://www.cl.cam.ac.uk/research/srg/netos/xen.

As the device drivers are run in the Domain0 and thus outside of the actual kernel, Xen has to be considered as a so-called micro-kernel.

### Linux integration

As stated above, Xen can run both paravirtualized and native guests. In order to run paravirtualized guests, Xen-specific drivers are included in the Linux kernel.
The Dom0 requires additional modifications that have not been included into the main kernel trunk. Thus, it is necessary to apply patches to a stock kernel before it can be used as Dom0 kernel. This work is performed by all major Linux distributions.
Unfortunately, the official patches are usually only available for older kernels, making it difficult to combine new Xen features with bleeding edge Linux kernel technology.

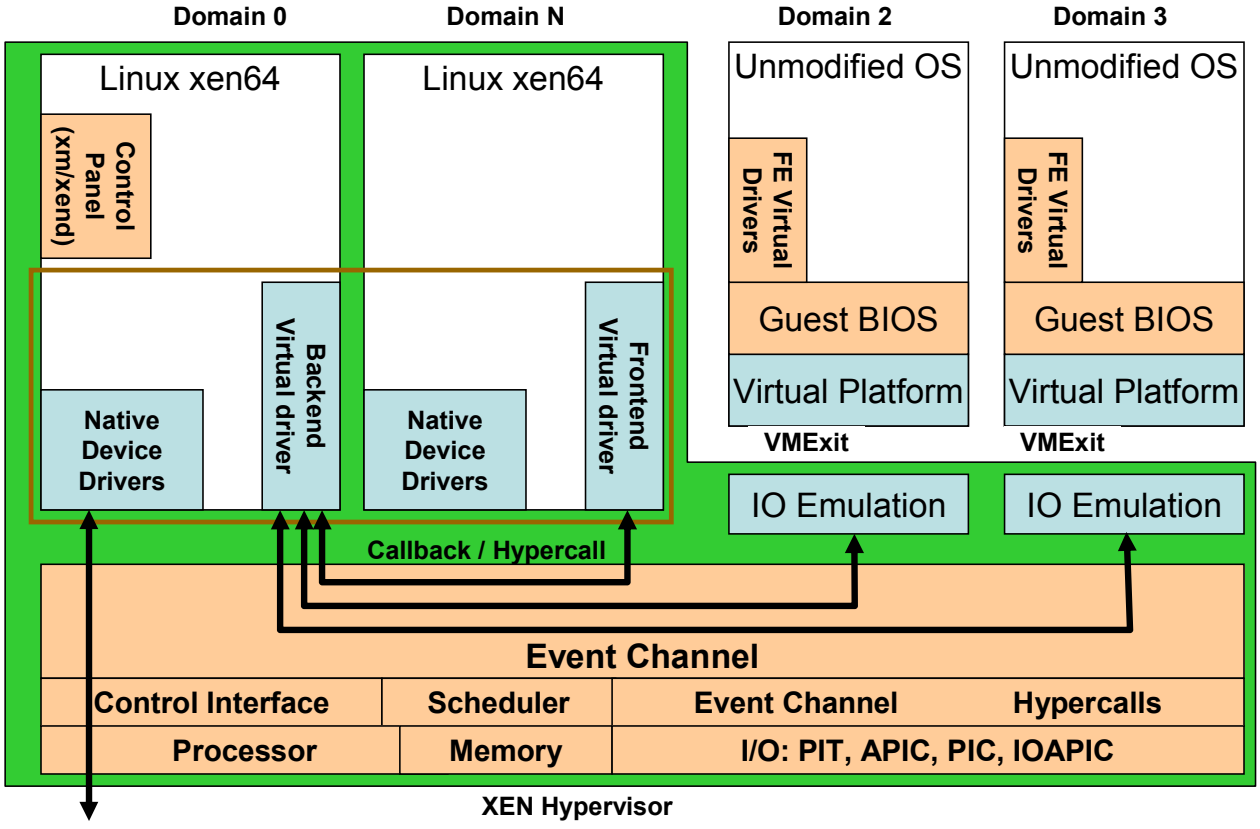


**Figure 2. Schematic Xen Architecure, from [7].**

### Xen and Real-Time Scheduling

Xen supports pluggable CPU schedulers. Currently, it is distributed with two of them:

- Simple Earliest Deadline First (SEDF) Scheduler;
- Credit Scheduler.

Xen 3.0 contains also a BVT (Borrowed Virtual Time) scheduler, but it will be removed in the newer releases of the hypervisor. SEDF is planned for removal too.

#### *Simple Earliest Deadline First (SEDF)*

The SEDF scheduler is similar to the Atropos scheduler, described for the Nemesis system in [8]. It basically uses an internal EDF scheduler to meet the desired QoS requirements, expressed by the user in terms of budget and period.

It can be configured so as to be work-conserving (i.e., should any computating power exceed the required one, it is redistributed to the competing VMUs) or not, and provides some heuristic enhancements designed for the purpose of decreasing the wakeup delays of latency-sensitive domains.
On SMP environments, the SEDF scheduler undertakes a purely partitioned approach, with no specific algorithm to initially place VMs on the available CPUs. The user can specify an affinity for a VM, forcing its allocation to a processor or to a set of processors.
From the QoS perspective, the SEDF scheduler can be used to meet the desired budget and period, depending on the configuration provided. No admission control is performed, so it must be implemented somehow in userspace.

#### *Credit Scheduler*

The Credit scheduler seems to be the scheduler of choice for the future of Xen. It has been designed ex-novo to be work-conserving (with an optional per-VMU maximum utilization limit) on SMP systems and to correctly handle load balancing issues.
To the best of the authors' knowledge, this scheduler has no equivalent in the literature, and is a purely heuristic-based fair share scheduler. It does not provide any explicit service guarantee, and, as far as the authors can deduce, it is not easy or maybe not even possible to derive them from the implementation.
It basically works assigning each VCPU credits in proportion to the number of VMs in the system and on VCPUs in each VM, and to the configuration of the VM with respect to any slack time that may become available. Each credit corresponds to some CPU time.
Even if it maintains per-CPU runqueues, scheduling decisions are taken globally. VCPUs are partitioned into two sets: UNDER, containing all the VCPUS that received less than the assigned number of credits, and the OVER one, containing the other VCPUs. At any scheduling decision the VCPUs in UNDER are favored over the ones in OVER, but there is no specific policy that defines the behavior within the same set. Credits are recalculated every system tick.
The scheduler does nothing to guarantee any upper bound on the wakeup latency of the scheduled VCPUs. It adopts a heuristic boosting mechanism, considering every VCPU that wakes up from any blocking period as if it belonged to the UNDER set.

### Strengths of Xen

One of the major strengths of Xen is its dissemination. It is both available for free as open source product, licensed under the GPL, and as a commercial product, including support, under a commercial license.
The use of paravirtualization allows Xen to reach a guest performance close to the one of the host machine on which it runs.
Xen runs on standard PC hardware, virtualization extensions of current CPUs are supported, but optional and only required for running native guests.
Due to its wide spread usage, a lot of documentation is available for Xen.

### Weaknesses

In the context of the IRMOS project, the major weakness of Xen is the lack of integration into the main Linux kernel development line.
This results in extra effort if the bleeding edge open source edition is used and makes maintenance harder.

### 3.2.2 Kernel-based Virtual Machine (KVM)

The Kernel-based Virtual Machine (KVM) [29] is a hypervisor integrated into the Linux kernel, initially designed to enhance the full virtualization performance of the processors supporting the Intel VT-x and AMD SVM virtualization extensions. Lately it has been extended to support paravirtualization for some performance-critical VM-hypervisor communications.

KVM supports the execution of unmodified or modified guests; paravirtualization is optional and limited to device drivers.

KVM has been developed originally from Qumranet, a company based in Israel, that was born around the KVM product. Now Qumranet has been bought by RedHat Inc., that is actively developing and supporting KVM.

#### Architecture

KVM is composed by two main components (see Figure 3), an in-kernel hypervisor and a user-space client.
The hypervisor is just a subsystem of the Linux kernel, that exports a dedicated interface that allows the user-space to manage virtual machines, and supports the execution of these virtual machines treating them as ordinary user processes. The kernel uses the hardware acceleration support for virtualization to execute the virtual machines. All the other resource management, i.e., memory management, CPU scheduling, etc., is handled in an integrated fashion with respect to the other kernel subsystems.
The second component, the user-space client, serves two main functions:

- export the management commands to the final user;
- emulate the VM hardware components that are not emulated directly inside the kernel, mapping them to the real hardware. In the case of paravirtualized drivers it just acts as a proxy for the kernel itself.

The user-space client is a modified version of the well-known open-source machine emulator and virtualization layer QEMU[4]; the KVM and the QEMU communities are working together to integrate the KVM specific bits into the official QEMU releases.

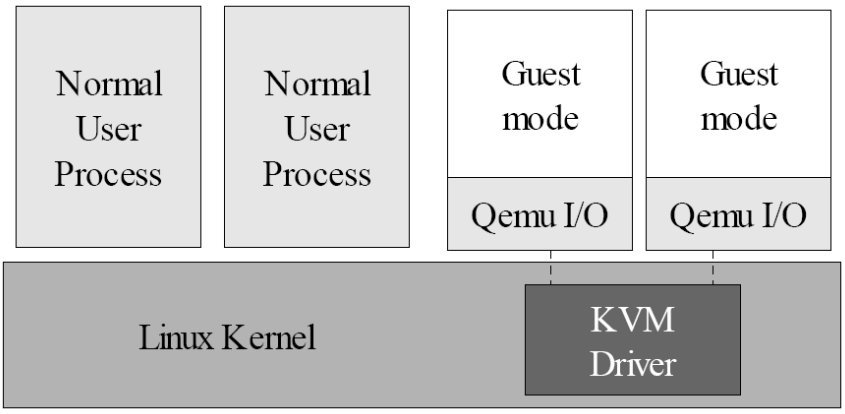


**Figure 3. Schematic KVM Architecture, from [29].**

[4] More information is available on the official QEMU website at the URL: http://bellard.org/qemu.

The fact that the development of both the user-space client and the kernel module is tightly coupled with the mainstream of two well supported open source projects as QEMU and the Linux kernel gives great stability to the future of KVM.
The introduction of paravirtualized devices allows for an easy transition from full virtualization setups to paravirtualized ones.

#### KVM and Real-Time Scheduling

As depicted in Figure 3, each instance of KVM, running an entire emulated machine along with all the on-board software (comprising the guest OS), is seen by the host OS as a Linux process (composed of multiple threads). Therefore, scheduling of VMs is performed by the host Linux OS according to the same principles that govern scheduling of standard processes within the OS. For example, as Linux is a POSIX compliant OS, it is possible to schedule multiple VMs by using Fixed Priority scheduling. However, from a real-time scheduling perspective, it is noteworthy to mention that such an approach would allow for providing very limited scheduling guarantees to individual VMs (KVM instances), mainly because of the lack of an appropriate temporal isolation mechanism inside the standard mainstream kernel. However, this is one of the key aspects we are going to improve by means of the development efforts associated to Task 6.5 within the WP6 of the IRMOS project. See Section 5.3 for a detailed discussion on the topic.

#### Strengths of KVM

The main strength of KVM is the fact that it is distributed together with the Linux kernel, and it is fully integrated into it. This greatly simplifies the development and the testing of all the involved components. Moreover, being the Linux kernel solution to virtualization, KVM will be, in the next few years, a reference for other virtualization strategies.

#### Weaknesses

The QEMU side of KVM suffers from some of the problems that affect also the mainline QEMU, i.e., some lack of documentation, lack of testing for some of the subsystems, and so on. Moreover, with respect to CPU scheduling, the fact that there is no clear distinction between regular user processes and virtual machine ones can create some problems. The framework for process grouping recently integrated in the Linux mainstream kernel should help in finding a clean solution to the problems that may arise.

### 3.2.3 Conclusions

In this chapter, an overview of virtualization and its use within the IRMOS project has been done, from a perspective that considers the possible impact of virtualization on timing and performance of the hosted services. A set of requirements on the virtualization layer have been identified and enumerated, and they have been used for the purpose of selecting existing virtualization solutions that are suitable for prototype development within the IRMOS project. Precisely, two open-source virtualization solutions have been selected, Xen and KVM, and have been described. Generally speaking, KVM tends to support less virtualization features than Xen does. However, its tight integration within the mainstream Linux kernel is a major advantage factor as compared to Xen. Moreover, as it seems that all the requirements on the virtualization layer, as identified in Section 3.1, are fulfilled by this VMM solution, the IRMOS project is

currently oriented towards choosing KVM as the reference virtualization layer for the final prototype. However, note that the IRMOS architecture will be completely independent from the specific virtualization solution that will be adopted at the Execution Environment layer, so this will be an implementation detail completely hidden within the EE. Therefore, it will be possible to support further VMM solutions fulfilling the general requirements identified in Section 3.1, should they become available in the future.

# 4 EE computing requirements breakdown

The EE of a Physical Host will have to host, in addition to the VMUs providing the actual application-level services to the end users, further software components that are needed in order to let the IRMOS platform run. These software components will provide the fundamental services over which the entire SOA machinery, comprising workflow enactment, and the ISONI infrastructure rely in order to work properly.

It is important to have awareness of such services, because their invocation (that may be quite occasional with respect to the user of the main application-level VMU hosted services) must not be allowed to interfere, in an unforeseen way, with the main application-level VMU services that may be running on a host. On the other hand, one may want to reserve basic QoS guarantees to such infrastructure services themselves, in order to have the fundamental IRMOS/ISONI platform services exhibit a minimum reasonable response time, even in presence of CPU-intensive applications hosted within the platform.

In the following, a brief overview follows about the basic platform services that are needed, mainly related to the ISONI and IRMOS WP5 services management, where the discussion mainly focuses on timing interferences and computation requirements. Specific WP5 and WP7 Project Deliverables will describe in more detail the mentioned services.

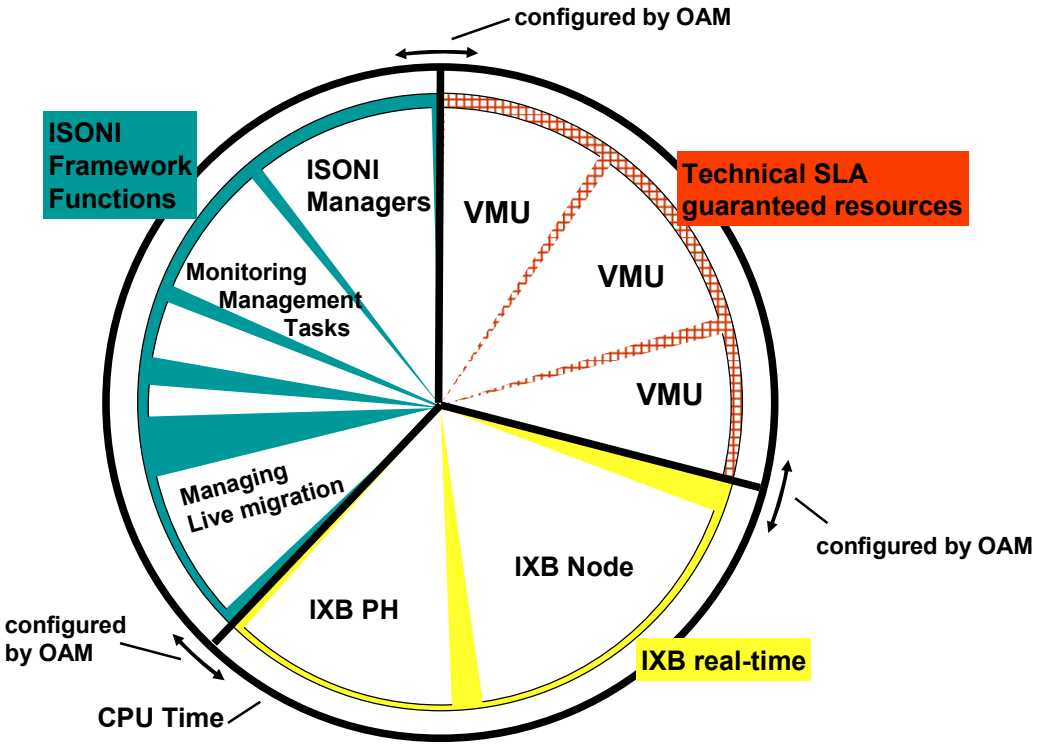


**Figure 4. Overview of the activities that will consume CPU power in an Execution Environment.**

The set of activities that are foreseen to be running on an IRMOS Physical Host may be grouped into categories and sub-categories, as depicted in Figure 4, where one would

like to configure the EE scheduler so as to provide performance guarantees to the individual activities. Specifically, the following categories are foreseen:

- VMUs that are running on the system: this category may be split on its own in various sub-categories, depending on the (main) functionality the VMU is responsible for:
  - VMUs hosting CPU-intensive services;
  - VMUs hosting services for accessing storage devices;
  - VMUs supporting networking set-up, configuration and management.
- Functions that are specific to the ISONI framework: this category may be split into the following sub-categories:
  - Execution of ISONI domain functions like:
    - ISONI Portal
    - Registrar
    - ISONI Info system
    - SLA Manager
    - Deployment Manager instances
    - Resource Manager Domain
    - Path Manager Domain
  - Execution of ISONI node functions like:
    - Resource Manager Node
    - Path Manager Node
    - Node internal measurement data accumulation and providing data to Deployment Manager
  - Live migration: may be the experience in this project will come to the conclusion, that this may also need to have a separate category due to possibly heavy computation and networking requirements
- Functions that are needed by the IXB real-time:
  - IXB real-time PH: covers the time critical tasks of an IXB PH
  - IXB real-time Node: if co-located on this PH, it covers the time critical tasks of an IXB Node.

Ideally speaking, one would like to allocate the computing power available on the Physical Host of an Execution Environment to the above mentioned activities, so as to meet the performance and timing constraints of each activity. The portions of computing power to be assigned depend on the functions realized on the dedicated PH. For example, for a PH not acting as an IXB node, the portion of IXB CPU consuming time will be a smaller portion than for a PH with IXB node functionality. Note that managing the CPU consumption time for all individual processes in a very flexible and dynamic trimmed way is a very complex task.

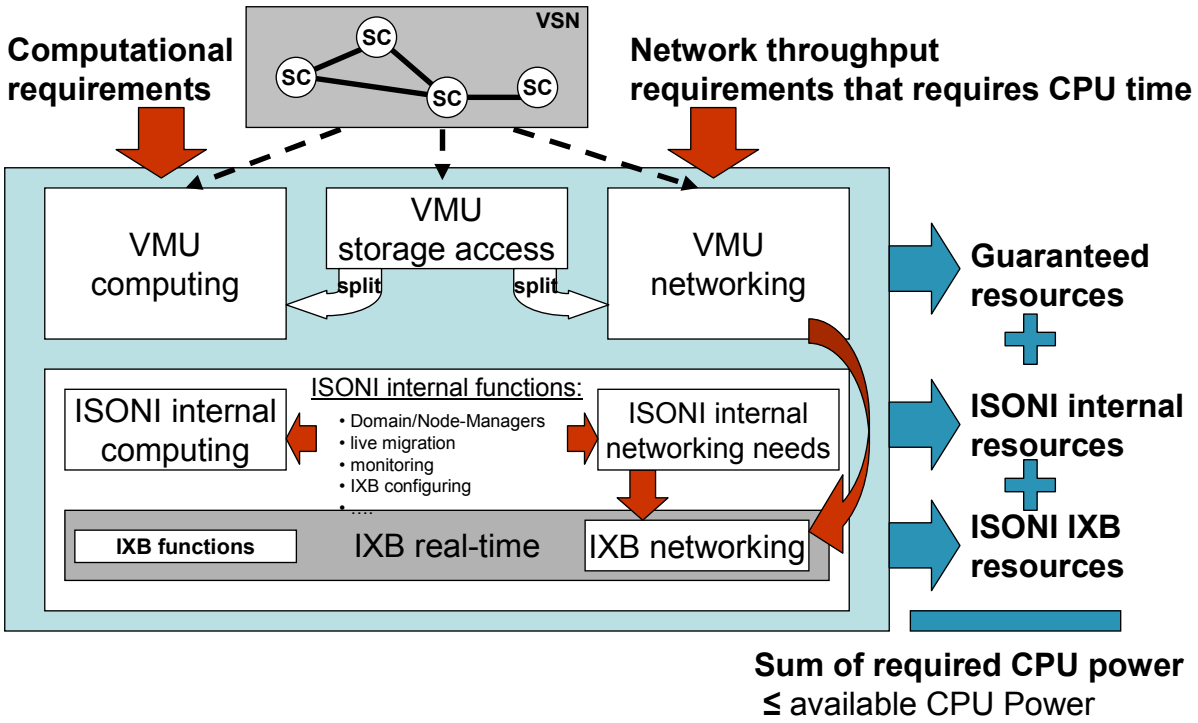


**Figure 5. CPU power consumptions complex dependencies**

As shown in Figure 5, the resource requirements deducted from a technical SLA request related with CPU consuming time impacts all main categories. VMU storage access for example can be divided in CPU power needed for storage access driver and for CPU time required for correlated traffic throughput. Each VMU throughput requirement impacts also IXB CPU time consumption. Also, additional functionality like monitoring, live migration and others add CPU time consumptions for the ISONI framework and IXB category.

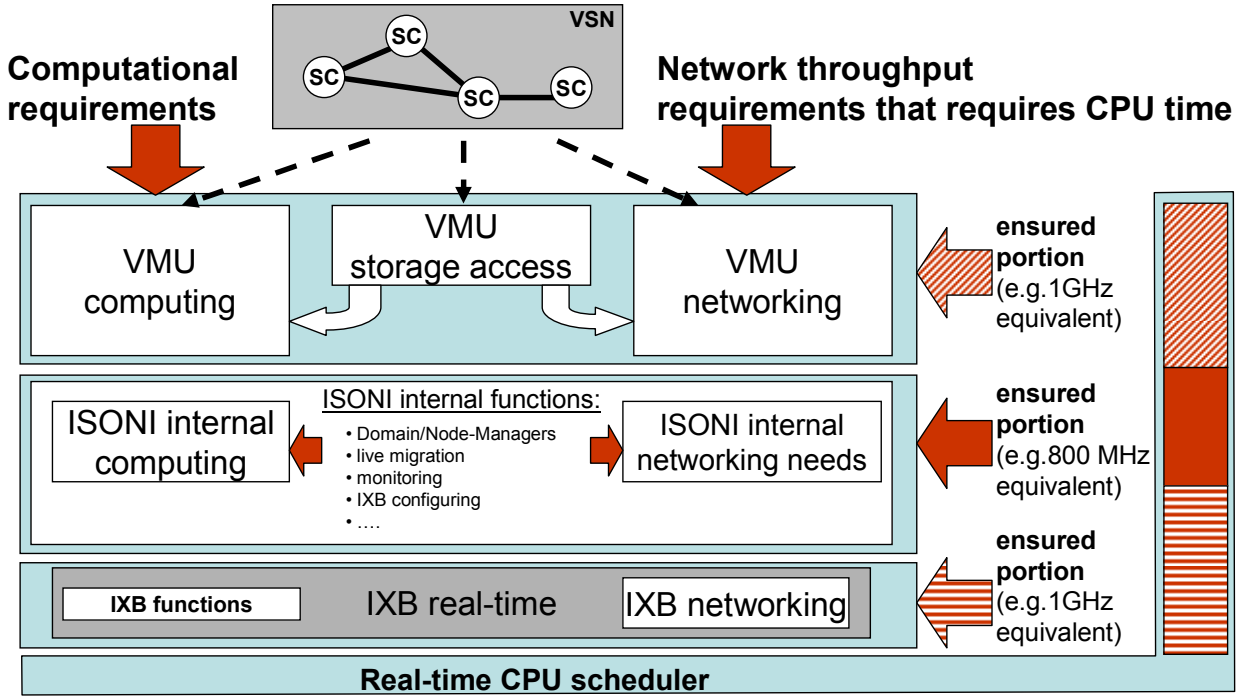


**Figure 6. CPU power consumptions with fix portions for the main scheduling categories**

As a preliminary note, it is foreseen that one possibility to reduce the complexity of computing power allocation will be constituted by the partitioning of the available computing power into fixed portions for the various categories. So, for each PH, it would be desirable to customize the scheduling parameters that govern the partitioning of the available computing power into the three general identified categories (see Figure 6 ).
Note that it is of paramount importance to provide precise scheduling guarantees to the running VMUs and the IXB real-time functions. Also, whenever the PH is running multiple Service Components with real-time requirements, it must be possible to control how the overall portion of computing power assigned to the VMU category is sub-allocated the individual VMUs.
On the other hand, not all of the above mentioned activities exhibit real-time scheduling requirements. For example, the ISONI framework functions are not expected to have such requirements, thus they may exploit the remaining CPU processing time with respect to real-time activities.
Note that, in order to tune appropriately the configuration of an EE in terms of shares of computing power needed by the functions enumerated in the various categories, an appropriate benchmarking and modelling phase will be needed.

## 4.1 The main categories of scheduling

Each fix portion allows calculating the performance/throughput of each main category. It represents the highest level of CPU time scheduling hierarchy. In addition to the main categories of scheduling, each part has an individual inner scheduling strategy. The concepts described in this chapter shall serve as a trial for modelling the requirements for a scheduling strategy. It shall be not seen as a description of a realization. The assumption in this chapter may be changed in the next stage deliverable depending on the experience gained during the project evolution.

### 4.1.1 IXB real-time

The guaranteed minimum CPU time for the complete main IXB real-time scheduling category is expected to depend on the network throughput, so it should be taken into consideration during in the decision process of allocation of functionality or resources. The IXB real-time tasks may use remaining unused CPU time of other categories in addition. Having a stable performance of the IXB is important for the health of an ISONI node.

#### Logical Functions covered by this category in detail

The IXB CPU consuming part includes also functionality which cannot be separated like IXB measurements and live migration support, but it does not include non-real-time tasks like configuring, measurement data collection and reporting.
The IXB real-time scheduling category covers (see Figure 7):

- Encapsulation of VMU traffic
- Network scheduling in respect to the different ISONI QoS classes
- Creation of measurement data at the concerned interface, where necessary
- Clock synchronization to enable one-way measurements of QoS parameters
- Time stamp tagging for measurement

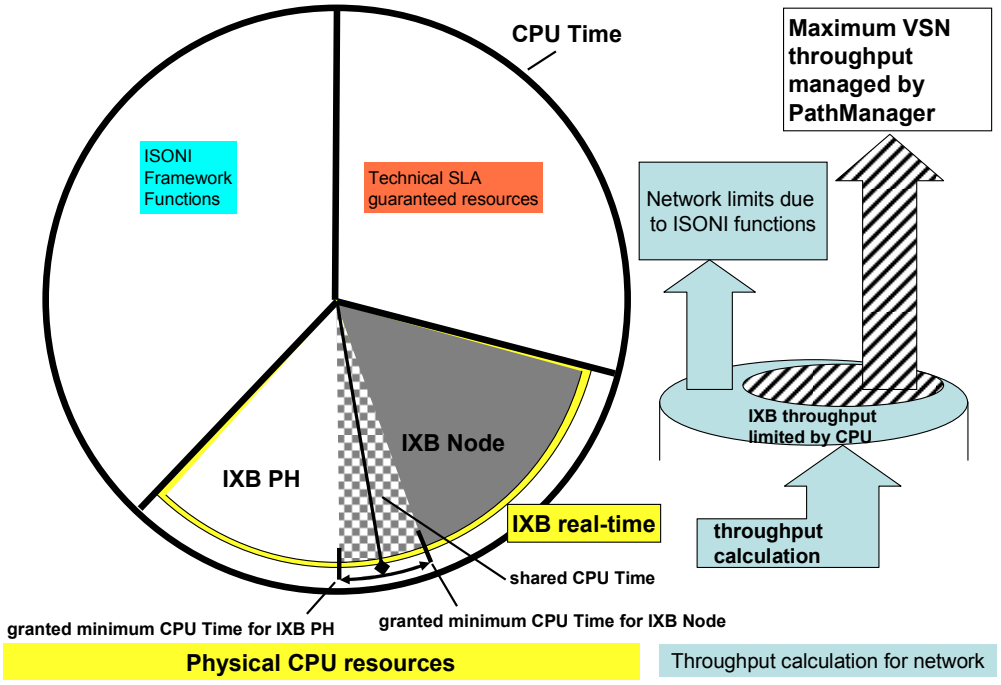


**Figure 7. Inner CPU time scheduling of IXB real-time category**

#### Inner scheduling of IXB real-time category

The inner CPU time scheduler of the real-time scheduling main category shall consider the need for dynamic adjustable portions of IXB PH and IXB Node CPU times. The ratio of the inner-node throughput performance and inter-node traffic throughput performance calculation depends on the deployed VSNs. The computing requirements of the IXB real-time category depends on the network traffic throughput. One part of the available network throughput related to consumed CPU time is used by ISONI Framework

Functions. The other part is available for the VMU intra-node and inter-node throughput network traffic, which is managed by the ISONI PathManager. The ISONI PathManager on node level needs this information as input to deduct the network resource availability, which is reported to the domain level. This throughput calculation represents rather the overall possible throughput of an ISONI Node and not the used/available bandwidth on the physical interfaces itself. Note that, on a PH, it could happen that bandwidth on physical interfaces is available, but the node throughput limits related to CPU time prevents the deployment of further VSNs/VMUs.
The outcome of this CPU time calculation for network throughput is one important input for the traffic management done by PathManager as described in Deliverable D7.2.1.

### 4.1.2 VMU technical SLA granted resources

This main category covers the granted CPU times for each VMU. Due to the fact that each deployed OS may have its own VMU internal task scheduling, ISONI grant just CPU times for the each VMU, which includes CPU time consumption for code executing, storage and networking (see Figure 8). Intra-VMU scheduling is not considered at the current stage of project. Whether a separation of CPU time consumption is needed in respect to VMU CPU, storage or networking depends on the experience that will be made during the project. The CPU time for running VMUs must be granted. Some minor fluctuations in short timeframes may be allowed, but the average must be granted (soft QoS). The portion of available CPU time for VMUs is the basis for the ResourceManager Node to report available CPU resources to the domain level.

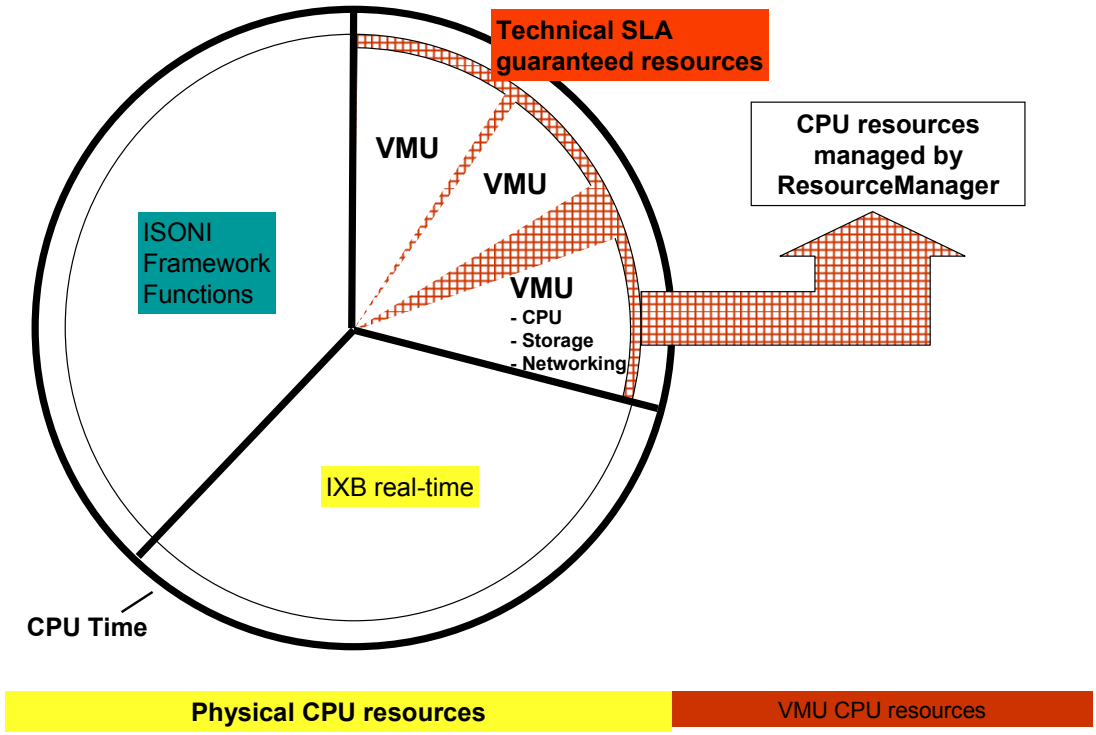


**Figure 8. Category of technical SLA granted resources**

### 4.1.3 ISONI framework functions (non real-time)

The remaining tasks, mainly the functional blocks of ISONI Domain and Node Managers, are ISONI framework task (see Figure 9), which do not necessarily need real-time. They need a certain time of CPU time in average over a longer time period. Fluctuations may be allowed. Unused CPU time may be provided to other CPU time main categories. The CPU time shall be used fairly among the concurrent functions.
Perhaps a scheduling weight is an adequate instrument paying attention to the different CPU time needs of each function. This will be elaborated during the project, if necessary. A PH heavily involved in ISONI Node or Domain managing function must have enough CPU processing time satisfying its responsibilities. But also this is scope of planning/dimensioning of an ISONI Node, which is an administrative OAM task.

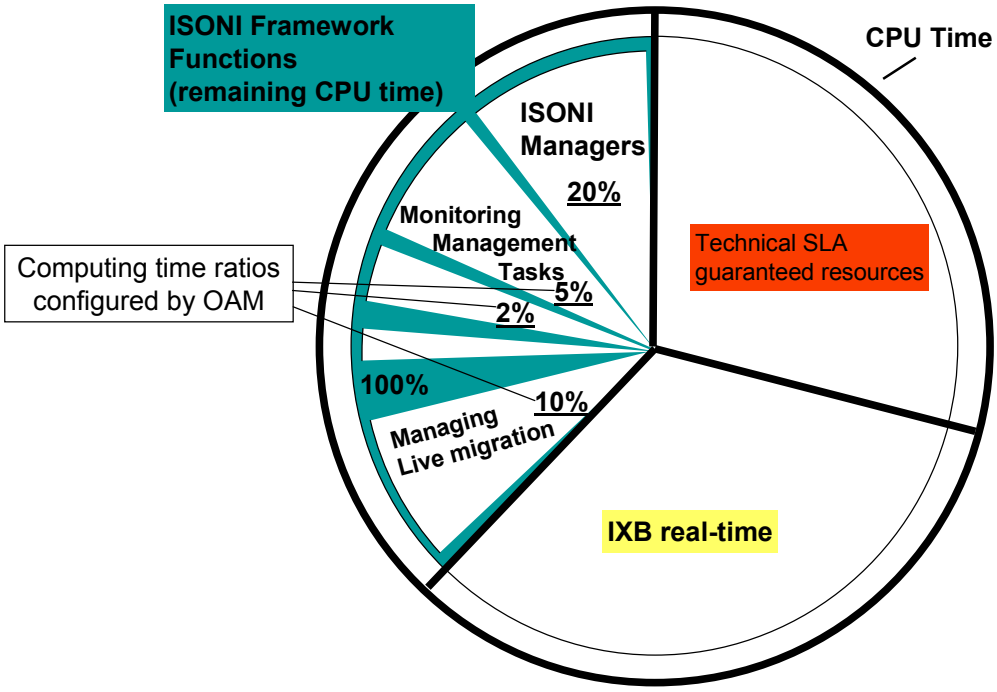


**Figure 9. Category of ISONI Framework functions**

### 4.1.4 Framework Services

In addition to what stated above, the WP5 Framework Services will require specific software components to be deployed on the individual Physical Hosts, and possibly inside the deployed VMUs themselves. These may be mainly subsumed as follows:

- Workflow Enactor Service
- Monitoring Service

It is important to get awareness of these fundamental platform services because the resources these functions are likely to consume at run-time will need to be properly considered when designing the admission policy for application SCs, in order to prevent possible undesired timing interferences.
As of now, it is envisioned that the global IRMOS Framework Services will be deployed within ISONI by means of a dedicated VSN, constituted by properly interconnected VMUs. So far, no real-time requirements have been detected for such software

components, which would enforce additional special treatments in relation to real-time scheduling. Therefore, from an ISONI perspective, these are confined within their own VMUs and can be managed like any other deployed application-VSN. As a consequence, their timing properties will rely basically on the scheduling guarantees provided to the containing VMUs and VSN, as established in the corresponding Technical-SLA.
Further description of these services is remanded to WP5 deliverables.

### 4.1.5 Possible roles of PHs in ISONI

Summarizing, the main activities that will consume computing power on a Physical Host have been classified into three main categories. Also, depending on the foreseen role of an ISONI PH, the configured portions for the CPU time assigned to the identified main categories may be different, as indicated in Figure 10 below.

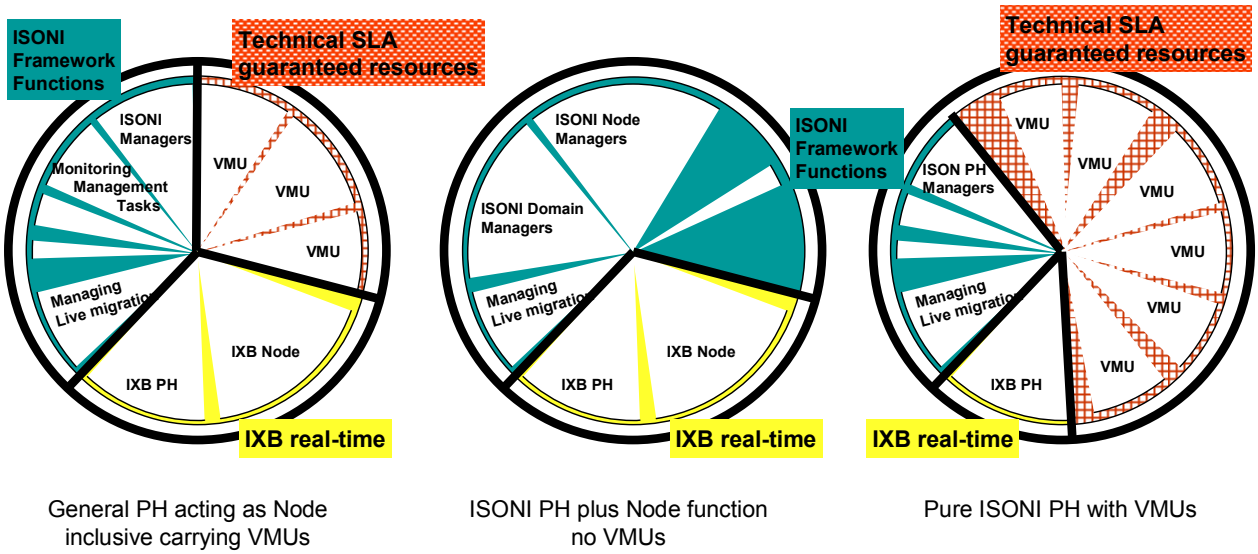


**Figure 10. Different scheduling category ratios**

In fact, some ISONI PHs may include all of the envisioned categories of activities (left cake in Figure 10). On the other hand, some ISONI PHs may even be configured for carrying ISONI node and domain functions without carrying any VMUs (middle cake). Finally, some other ISONI PHs may act as a pure ISONI PH without any ISONI Node and Domain functions optimized for carrying VMUs (right cake) at all.

# 5 Scheduling mechanisms for temporal isolation and QoS provisioning

This section deals with mechanisms that are needed in order to ensure appropriate scheduling guarantees, but at the same time enforce appropriate temporal isolation properties, across concurrently running applications within the IRMOS platform.

Within the IRMOS context, composite application workflows that carry out complex and real-time interactive applications are going to be used. These workflows require specialized allocation strategy, admission control and advance reservation policies.

*Global allocation* strategy is the procedure for determining where to execute an application, i.e., deciding on what physical nodes the service components constituting the application should be hosted, whereas *local allocation* deals with the assignment of individual tasks (and data flows among them) to the time-slices of the physical resources of the available Physical Hosts (and physical links interconnecting them). The job of local scheduling is left to the Execution Environment run-time support, i.e., the virtualization layer and/or Operating System (and also to the scheduling built into active networking elements, in case of data flow scheduling). Section 5.1 deals with global allocation, while local allocation is the subject of Section 5.2.

## 5.1 Global allocation strategy, admission control and advance reservations

The global allocation strategy needs to address the following points:

- satisfy user requirements by taking into account the QoS constraints with regard to the application as defined by the user in his SLA contract;
- take into account any preferences expressed by the user;
- deal with diverse resources that could be heterogeneous in terms of local policies and/or configuration.

Global allocation incorporates many processes such as scheduling architecture, decision making, planning scheme, scheduling strategy, and performance estimation. When dealing with real-time workflow applications decision making, the problem may become rather hard since the applications can have very diverse and heterogeneous characteristics. To this direction, the global allocation process could only take one service or part of the workflow into account and try to produce the best allocation for this part while ignoring the overall workflow which could impact its overall performance. On the other hand, the entire workflow as a whole could be considered which is much more time consuming but helps improving the overall workflow performance.

### 5.1.1 Advance reservation

Advance reservation of resources is widely seen as a mechanism to address the aforementioned problem [23]. In essence, advance reservation allows the application user to request resources for a specific time interval in the future, i.e., start time and end time, and obtain a sufficient number of resources during this time interval so as to

support the execution of the specific application while satisfying any performance constraints.
Based on application scenarios analysis, the potential IRMOS customer requests that his application should be available within a specific time interval – the reserved time interval – and for a maximum number of consumers with a specific level of QoS. However, in some scenarios the customer is unaware of when the qualified consumers are going to make use of the application, i.e., there is no fixed start-time for the execution of the application. From the customer point of view, the IRMOS platform must make sure that the consumer gets the required level of QoS each time he/she decides to make use of the application. This means that the IRMOS allocation strategy must provide support to this scenario.
However, from the IRMOS resource provider point of view, the approach foreseen in such a scenario, which involves real-time critical applications, makes things worse in terms of system utilization. In fact, if the resources that have been booked for a given reservation were not used for whatever reason (i.e., the actual consumer is not using the service), and there were no resource sharing between different reservations, then such resources would be forced to be idle, what would lead to low utilizations and non-optimum management of the resources. Therefore, the resource provider would be forced to sale resources at higher costs than strictly necessary. In this view, the following two criteria must be considered in building and evaluating the IRMOS allocation strategy and scheduling system:

1) The satisfaction of the consumers with respect to how well the allocated resources and applied scheduling scheme meets the performance constraints, and
2) The satisfaction of the resource providers in terms of how difficult or costly it is to support these performance constraints.

To this end, the combination of advance reservations with local scheduling is widely considered to be a very promising solution [24]. Many different approaches have been proposed in achieving this goal. The wide variety of these approaches and the lack of uniform means for evaluation make it difficult to compare different existing solutions.
The existing approaches on advance reservations and scheduling can be divided into two major categories: (1) Deterministic and (2) Non-deterministic/Probabilistic.

### 5.1.2 Deterministic advance reservation

Deterministic advance reservation consists of mathematical methods for reserving resources. It is based on deterministic algorithms for analyzing performance constraints by applying mathematical processes to the various layers and assumptions associated with each candidate node. In general this technique reduces the risk of missing deadlines and failures and increases the overall reliability of the system since they can be run on real-time environments efficiently. However, such methods do have limitations. Existing deterministic scheduling algorithms assume that the parameters for flexibility are static [25], are mostly focused on minimizing the response time and do not promote resource sharing while in other cases all requests for execution that overlap with previous ones are rejected. Studies on the performance of these mechanisms [26] [27] have demonstrated good results in terms of satisfying the given performance constraints. However, it is also shown that they lead in fragmentation of the resources and lower utilization.

### 5.1.3 Probabilistic advance reservation

The basic motivation behind probabilistic advance reservation is the fact that in many cases the input parameters and the run-time conditions are so many, that it is inefficient and sometimes even impossible to analytically examine the entire solution space. Probabilistic methods generally employ the principles behind decision theory and are influenced by prior probabilities that are derived from analytical methods coupled with benchmarking. These prior probabilities are used to determine the posterior probability of an advance reservation decision. These algorithms demonstrate better results in terms of resource utilization and lead to lower costs for the resources, but in general appear to be less reliable than deterministic methods in terms of the resulting performance granted to the running applications.
Given the current status of the IRMOS project, it is envisioned that once a VSN has been created within the platform, it will not be possible to change its structure at run-time, mainly due to technicalities that are out of the scope of the present document, but that will be addressed in WP7 deliverables. Therefore, we may consider the reservation process the time in which a decision is made about where and when the application service components are to execute. In this sense, once a decision is made on where to place and execute a job, no further decisions are made concerning the job during run time such as migrating a service execution to another resource when its initial contract is broken or a better resource is found for execution. The decision of the advance reservation mechanism will be based on information on load and scheduling information available to the system in correlation with low-level performance information that corresponds to the estimated execution time or other system resource demands. One critical point here is that the one-time assignment approach increases the critical factor of the advance reservation and mapping process within the IRMOS platform and it is crucial that a near optimal solution for the allocation and management of the resources is produced during advance reservation.

## 5.2 Real-time low-level CPU scheduling

Considering the problem of co-scheduling multiple VMUs with real-time/performance constraints within the same PH, as it is envisioned to happen in IRMOS, the current section focuses on KVM-oriented approaches, in which a Linux Operating System may host multiple VMUs in the form of processes that run concurrently (each VMU process, internally, may contain multiple threads that mimic the behaviour of the emulated machine along with the guest OS and other running software).
Therefore, in the following, a brief update is made about the state of the art concerning CPU scheduling inside the Linux kernel, as compared to what already presented in the Project Deliverable D2.3.1 [9], Section 4.14.1. Here, the latest changes, improvements and planned activities related to CPU scheduling are briefly presented.
Then, in Section 5.3, the actual work carried on in this area within the IRMOS Project is presented, along with planned directions of future work.

### 5.2.1 Real-time scheduling in the Linux kernel

The material that is presented next has circulated in the last few months in various forms like kernel patches, messages among developers on the Linux Kernel Mailing List, and official released documentation of the Linux kernel.

### CPU Throttling

The current kernel code already embeds a rough mechanism, known as *CPU Throttling*, that has been designed for the purpose of limiting the maximum CPU time that may be consumed by individual activities on the system. The mechanism used to be available on older kernel releases only for real-time scheduling policies for *stability purposes.* Namely, it was designed so as to prevent real-time tasks to starve the entire system forever, for example as a result of a programming bug while developing real-time applications. The original mechanism only allowed the overall time consumed by real-time tasks (no matter what priority or exact policy they had) to overcome a statically configured threshold, within a time-frame of *one second*. This used to be specified in terms of the maximum amount of time (a.k.a., throttling *runtime*, expressed in microseconds, which corresponds to the well-known concept of *budget*, in the real-time literature), defaulting to *950 ms*, available to real-time tasks within each second (a.k.a., throttling *period*).

Only recently, core kernel developers recognized the usefulness of such mechanism for purposes related to *temporal isolation* of tasks *among each other* (as opposed to being used solely between the group of real-time tasks and the one best-effort tasks). Therefore, a well-defined interface has been defined in order to support throttling both at the task/thread level, and at the task group level, by taking advantage of the `cgroup` virtual filesystem.

The latter [14] constitutes a generic support for hierarchical arrangement of elements in the Linux kernel, that is exposed at the user-space level as a special virtual filesystem that may be mounted in any location within the root Linux filesystem (for example, under `/cgroup/...`). Taking advantage of such framework, hierarchical scheduling among processes (whenever supported by the respective scheduling policies) may be configured by creating appropriate folders and subfolders in the cgroup filesystem, where each folder corresponds to a group of tasks, and special file entries in the folder configures group-specific parameters.

Thanks to this framework, the POSIX semantics of real-time task scheduling in Linux has been recently modified, adding support for group scheduling, following the general trend of adding container support to all the subsystems of the kernel.

With such a framework, whenever a processor becomes available, the scheduler selects the highest priority task in the system that belongs to any group that has some execution budget available, then the execution time for which each task is scheduled is subtracted from the budget of all the groups it hierarchically belongs to. The budget assigned initially to a group is the same on all the processors of the system, and is selected by the user.

The budget limitation is enforced hierarchically, in the sense that, for a task to be scheduled, all the groups containing it, from its parent to the root group, must have some budget left.

In the case of a per-task-group throttling configuration, a special file entry inside a task-group folder named `cpu_rt_runtime_us` allows for configuring the maximum budget consumable by the entire sub-tree of tasks having that folder as ancestor.

From the real-time guarantees perspective, the throttling mechanism is well suited to prevent real-time tasks from monopolizing the CPU due to unexpected overruns. This basically means that the time granularities the mechanism is based on are quite long, in the order of 1s-10s, and that it is not foreseen to have too many competing groups.

However, the lack of an EDF scheduler makes using different periods impossible, so the mechanism seems to be not flexible enough to guarantee reasonably short wake-up latencies (see next paragraph).
Furthermore, one of the challenges currently under discussion is related to the management of throttling on multi-processor systems. In such a case, the current throttling mechanism follows a partitioned approach:

- each group in the system has a runtime assigned in each CPU in the system;
- all the budgets of a group are initially set to the budget of the group;
- a heuristics-based feedback mechanism tries to migrate the budgets among processors, increasing the time available to a group on the processors where it consumed more CPU time, borrowing it from the processors where it was less active.

Due to the global nature of the priority scheduling (group throttling partitions the bandwidth available to groups using a partitioned scheme, but whenever a group is selected on a CPU, the task it executes is picked according to its global priority), this feedback mechanism seems to be weak at least, given that there may be not enough correlation between the required runtime on each CPU by each group.
Also, given the traditional non-real-time orientation of Linux, that lacks essential admission-control mechanisms at the kernel/scheduler level, such migrations among processors is completely heuristics-based, and does not even perform basic checks about the utilization of the processors (being defined in this case as the sum of the ratio among the budgets and periods of task groups that have been migrated onto a given CPU).
The current throttling mechanism has two major drawbacks:

1) from a real-time theoretical perspective, it works basically like the well-known "Deferrable Scheduler" algorithm [15], in the literature of Real-Time scheduling (at least looking at what happens on a single-CPU system): such scheme has been overcome by a number of other schemes that perform much better [15];
2) the current implementation enforces temporal encapsulation on the basis of a common time granularity for all the tasks in the system, that is one second; this makes it impossible to guarantee good performance on service components that need to exhibit sub-second activation and response times.

Concerning the first issue, it is well-known from the real-time literature that the Deferrable Scheduler algorithm has various drawbacks from the perspective of predictability and analysis of a system with Fixed Priority based scheduling. Such limitations have been already overcome since a long time ago by the "Sporadic Server" algorithm (see [15] for further details), that has also been standardized by POSIX among the possible scheduling policies available in the set of real-time extensions to a POSIX compliant Operating System.
According to what the official kernel documentation [10] reports, it is already planned as future work to overcome the second limitation, by implementing a per-group specific time granularity based on which the maximum budget (runtime) is enforced. Also, it is suggested that such parameter be configurable by adding a second task-group special file in the cgroup filesystem, named `cpu_rt_period_us`.

### Earliest Deadline First (EDF) Scheduling Policy

The official documentation of the current release of the Linux kernel [10] reports that an implementation of a SCHED_EDF (Earliest Deadline First) scheduling policy is currently planned. This need was also expressed by core kernel developers that IRMOS people met during the latest (10th) Real-Time Linux Workshop, in November 2008.
Given the current status of the Linux kernel scheduler, any new policy should at least be capable of supporting a well-defined behaviour for scheduling groups of threads, in a hierarchical fashion (by leveraging the cgroup framework described above), and a mechanism for dealing with Priority Inversion issues typical of real-time systems, such as Priority Inheritance (PI). Unfortunately, the current Linux PI infrastructure is geared towards the limited static priority levels ranging from 0 to 139. With deadline-based scheduling, such mechanism needs to be extended so as to realize a "Deadline Inheritance" protocol.
In fact, the same core kernel developers admit in [10] that: "*This means the whole PI machinery will have to be reworked - and that is one of the most complex pieces of code we have*".
Concerning this last aspect, it is noteworthy to mention that SSSA theorized [12][13] and implemented [11] such a mechanism, baptized as "Bandwidth Inheritance Protocol", inside the AQuoSA [30] architecture for the Linux kernel, in the context of the FRESCOR European Project [31]. Unfortunately, such mechanism has been designed for single-CPU embedded systems, thus it is not suitable for being used in the IRMOS project, which aims to support high-performance, server-side computing. Furthermore, it is not clear how to reasonably mix such protocol in those possible cases in which resources are shared between tasks scheduled by an EDF-based policy, and other tasks scheduled by other policies, in the same system (like it will happen on a Linux system). This constitutes, as of now, an open research issue.

### Priority-based scheduling versus other policies

The Linux kernel is currently largely influenced by the need to comply with the POSIX standard for scheduling services. However, from a theoretical perspective, priority-based scheduling is well-known to be non-optimal, in that, a single-CPU machine, it cannot achieve saturation of the processing capabilities, whilst EDF-based scheduling does manage to achieve a theoretically full saturation (single-CPU case).
On multi-processor systems, the situation is much more complex, and a definitive answer to the problem of an efficient use of SMP computing resources (i.e., how to nearly saturate the machine with real-time tasks) does not yet exist, in the domain of real-time scheduling. Interestingly, a theoretical result has been found quite long ago in [18], where a global scheduling algorithm called *Proportionate Fair* (a.k.a., *Pfair*) has been proved to be capable of scheduling an arbitrary real-time task set with deadlines equal to periods, on a multi-processor machine, with a theoretically full saturation of the machine.
Unfortunately, the algorithm (at least in its original formulation) requires, at each time quantum, quite heavy computations in order to figure out what tasks to schedule on the available processors in the next time quantum, it needs to be applied synchronously on all the processors, and also it requires to continuously migrate tasks among processors, what forces performance of local caches to go down in an uncontrolled fashion. These issues make Pfair nearly unusable for practical purposes, as also highlighted by the

experimental evaluation recently appeared in [20], where experimental results have been presented gathered from a real implementation of the algorithm within the Linux kernel, run on a Sun Niagara multi-core workstation, developed in the context of the LITMUS$^{RT}$ project [28].
The *Staggered Pfair* algorithm [19] is a variation of Pfair that reduces the main overhead issues due to synchronous invocation of the scheduler on the various CPUs (causing synchronous cache misses on many cores/CPUs, what results in a tremendous impact on performance) by delaying enforcement of the new schedule on the various processors of a prefixed little amount of time, at the cost of a slightly decreased theoretically achievable saturation. Still, the overhead needed in order to compute the task schedule is high, as compared to partitioned approaches. A preliminary but practical study on the overheads achieved by various real-time multi-processing approaches on Linux has been recently presented in [20]. In the work it has been shown that the most promising approaches seem to be the ones based on global scheduling approaches applied to clusters of CPUs, so as to take advantage of the inherent cache-grouping mechanisms typical of modern multi-core architectures. In fact, in new generation multi-core machines, multiple "logical processors" may be realized in terms of physical processors, cores inside the same processor, and even hardware threads inside the same core, with cache memories placed at various levels of the hierarchy and logical processors insisting *in groups* on the same cache memory.

## 5.3 CPU Scheduling in IRMOS

As shown in the previous section, the Linux kernel currently lacks an appropriate real-time strategy that may be directly used within IRMOS, for the purpose of ensuring appropriate scheduling guarantees with sufficiently short wake-up and response times to the running SCs of a VSN. This section discusses the current development roadmap for realizing possible mechanisms that may be used for such purposes.

### 5.3.1 Sporadic Server

In the context of the IRMOS project it was developed [16] an implementation of the Sporadic Server algorithm, as mandated by the POSIX standard [17], under the name of SCHED_SPORADIC scheduling class. The implementation has been presented at the latest Linux Real-Time Workshop in Mexico, where useful feedback from the community has been collected. During the development steps, also a patch[5] has been produced for the purpose of fixing the behaviour of the standard CPU throttling mechanism of Linux in the case of hierarchical arrangement of tasks and task groups. This patch has been considered a urgent bugfix by core developers, thus it has been immediately merged into the mainstream Linux kernel. However, it is not yet clear what will be the outcome concerning the POSIX SCHED_SPORADIC policy implementation.
Interestingly, one of the suggestions made by core kernel developers has been the one to adapt the currently developed patch (which is compliant to a POSIX class standard) so as to exhibit the same interface as the Linux CPU throttling, in order to entirely replace the current throttling mechanism by a Sporadic Server algorithm.
Note that such development activities have been mostly targeted at "spreading" real-time knowledge within the core kernel development team, focusing on a real-time policy

[5] More information may be found at the URL: http://lkml.org/lkml/2008/10/3/279.

that is largely known and standardized. However, it is very difficult to have some piece of software accepted by the core kernel developers community[6].
However, further development efforts are already planned within the IRMOS WP6 that aim to produce a better scheduling policy which is based on Earliest Deadline First, rather than on Fixed Priority like the Sporadic Server, as described in the next paragraph.

### 5.3.2 CPU Scheduling Roadmap

Considering the current real-time scheduling policy supported by the Linux scheduler, based on Fixed Priority, the Sporadic Server implementation provided within IRMOS is the mechanism that "best fits" in the picture, in the sense that it does not subvert the priority-driven scheduling principles of Linux, yet it adds the property of "temporal encapsulation", i.e., the possibility to inhibit processes that try to "eat" more CPU than they were foreseen to. This may be advantageous in the context of the IRMOS project, for the purpose of running multiple VMUs in the same Physical Host, for example by means of the KVM virtualization layer. Thanks to the SCHED_SPORADIC policy, it would be possible to schedule VMUs both with real-time guarantees and with containment of the possible timing interferences among each other.

However, from a theoretical perspective, priority-based scheduling is well-known to be non-optimal, as already discussed in Section 5.2.1. For example, it cannot achieve saturation of the processing capabilities of a single-CPU machine beyond nearly 69%, whilst EDF-based scheduling does manage to achieve a theoretically full saturation (theoretically 100%) in such a case. Therefore, a variety of EDF-based algorithms have been proposed in the literature for scheduling real-time tasks on SMP systems, and, given the current status of the project, it is envisioned to use an EDF-like mechanism within IRMOS as well.
As discussed above, the Linux kernel currently lacks a CPU scheduling policy that is suitable for the IRMOS project applications. Currently, development activities (such as the Sporadic Server implementation) are being carried on, that serve the primary purpose of building a "stable presence" in the community of core kernel developers, which constitutes a basis for developing software components that may have any chance of being accepted in the mainstream kernel.
Actually, at the moment a completely new scheduling algorithm is under development especially suitable for IRMOS virtualized applications, that will meet the following requirements (these requirements have been preliminarily presented and discussed during the presentation of the paper in [22]):

- it allows for a completely partitioned EDF scheduling: this is the scheduler that may potentially achieve the maximum performance, and it is the optimal solution for single-CPU VMUs; unfortunately, for multi-CPU VMUs, such an approach may suffer of the fragmentation and low-utilization issues typical of partitioned scheduling approaches;
- it may allow for dynamic migration of real-time tasks across CPUs, in order to mitigate the just highlighted drawbacks, and also in order to avoid a full-fledged

[6] Often it happens that core developers rewrite their own version of a functionality, inspired by patches sent by others or general approaches discussed on the Linux Kernel Mailing List.

heavyweight bin-packing implementation, as usually needed in order to decide how to (optimally) allocate a set of real-time tasks to a set of processors;
- in alternative or addition, it may allow for clustering of CPUs so as to achieve a global EDF-based scheduling policy at the cluster level, whilst schedulers from different clusters may remain independent: this is the approach that is believed to be optimum in case of many cores per-CPU, as industrial practice seems to be geared towards production of such "monsters", despite the bottlenecks and shortcomings [21] due to the use of a unique shared means of access to the main memory, that constitutes a bottleneck for each multi-core chip;
- it may have a special management of spin-locks by exploiting para-virtualization approaches, by which the host may be given awareness of when the guest processors are spin-locking: this is supposed to be useful for improving efficiency in scheduling multi-CPU VMUs;
- it may exhibit some minimum non-preemptable time quantum: this is supposed to increase cache efficiency, allow for a more predictable benchmarking and reduce the maximum number of preemptions (and associated scheduling overheads); however, this may also affect negatively activation delays thus interactivity of hosted VMUs.

Note that the above list of requirements may dynamically change while the project evolves and implementation proceeds, as the optimum set of choices will need to be validated through extensive experimental validation and overhead measurements performed on prototype implementation(s).

### Scheduling parameters

Concerning the basic attributes that will be needed in order to properly set-up a real-time VMU running on a Physical Host, the scheduler that is planned to be developed will be capable of dealing with the traditional parameters typical of the real-time task model:
- budget: computation time expected to be consumed at each SC instance/invocation;
- period: minimum period between subsequent invocations;
- deadline: relative deadline within which each invocation should terminate and produce the expected output.

Under such a triplet of parameters, the scheduling guarantee that is provided by the underlying kernel-level mechanism is the following, no matter how it is realized: *guarantee that, within each time window of duration equal to the period, the associated task is scheduled for an overall time equal to the budget within the first sub-frame equal to the specified relative deadline.*

It is important to note that the above parameters cannot be replaced by a simple parameter like a "percentage of usage". In fact, it is important to give the capability to specify what is the time granularity by which each application should receive its guaranteed scheduling time frames by the underlying OS.

As an example, consider a (single-thread) application that wakes up and blocks periodically, so as to occupy nearly 25% of the processor (i.e., a real-time video transcoder would easily exhibit such a behaviour).

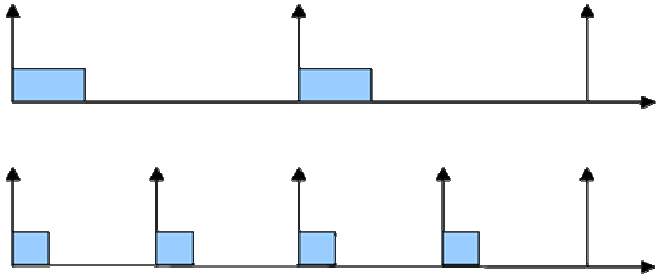

**Figure 11. Different ways to reserve "25%" of the CPU to a real-time task.**

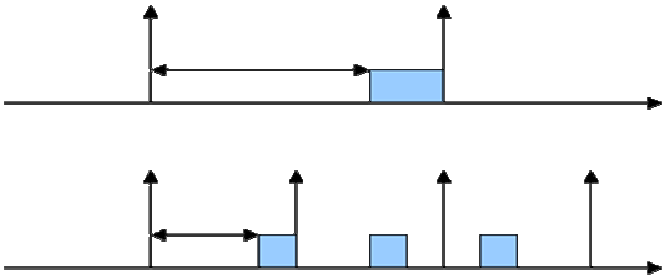

**Figure 12. Worst-case wake-up/response latencies for the two cases.**

Figure 11 depicts two possible (budget, period) pairs corresponding to an allocation of the 25% for the considered application. These choices correspond to very different worst-case wake-up and response latencies, when other real-time activities are co-scheduled on the same system, as shown in Figure 12. In fact, usually, the CPU allocation granularity (reservation period) is chosen so as to match the expected application activation period.

For SMP enabled VMUs, it is expected that each SC/VMU may be needed to be enriched with such information as the maximum parallelization degree supported by the enclosed SC software running inside the VMU. In fact, it is important to note that a model is needed for the purpose of foreseeing how the performance of a SC scales with the number of available concurrent virtual processors.

In fact, doubling the number of processors assigned to a SC does not necessarily imply that the execution time of each invocation of the SC will be halved. For example, consider the deployment of a legacy SC written as a single thread with no concurrency capabilities at all. Such an application would not be capable of exploiting the computing power of additional CPUs at all, thus it would not benefit from the allocation of additional CPUs. Similar arguments may apply for software components written specifically for exploiting dual-core architectures widely used nowadays, for example.

The way in which parallel real-time SCs may be deployed optimally on a network of SMP-capable PHs within IRMOS is still an open cross-WP research issue (involving benchmarking and modelling issues) that needs further investigations.

### 5.3.3 Scheduling multiple SCs within the same PH

Within IRMOS, an investigation has been performed on the way to support the execution of multiple SCs within the same PH. Such situation would of course occur as a consequence of the will of providers to share non-saturated physical resources among

concurrently running VMUs, as far as it is possible to respect the performance constraints as specified in the T-SLA.
Clearly, the immediate way for such a support would be to have one VMU for each SC to be hosted on a PH. However, it is noteworthy to highlight that, once a real-time capable scheduling policy will have been developed for the Linux kernel, then, with an Execution Environment adopting KVM as the VMM layer, it will be possible to use it both:

- at the *host level*, for scheduling multiple VMUs on the same Physical Host with scheduling guarantees and temporal isolation;
- at the *guest level*, for scheduling multiple services running within the same VMU with scheduling guarantees and temporal isolation for the individual activities that need them inside the VMU.

This opens up the theoretical possibility to host multiple SCs within the same VMU environment, as opposed to hosting multiple SCs inside completely independent VMUs. In fact, supposing to have for example a couple of Service Components that may be hosted on the same Physical Host, it is possible to identify two main deployment options (see Figure 13 below):

1) each SC is deployed within a different VMU (left sub-picture): the virtualization layer scheduler is responsible for allocating appropriately the physical hardware to the various services so as to satisfy the temporal constraints derived from the established SLA;

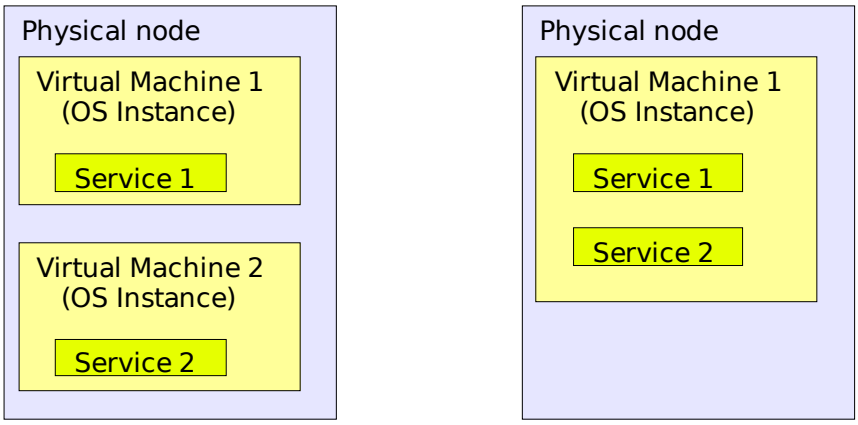


**Figure 13. Difference between the first (left) and second (right) approach.**

2) multiple SCs are deployed within the same VMU (right sub-picture): both SCs run within the same Operating System, thus the OS is mainly responsible for an appropriate scheduling of the hosted services, together with the VMM layer scheduling the OS itself.

### SCs in different VMUs

Major drawback of the first approach is the need for having *as many OS instances as* the number of independent services that need to be hosted onto the same Physical Host. For each OS instance, the platform needs to be capable of starting from a pre-configured VM image (with installed all the software that is needed by the service it is going to host), boot the machine and deploy on it the required service instance. This introduces a set of overheads that must be appropriately taken into consideration in the IRMOS general

design and access control policy. For example, the Physical Host will exhibit at run-time replicated OS-level services, which will lead to memory and scheduling overheads (globally, there will be more tasks to manage).
In this approach, the virtualization layer scheduler is responsible for scheduling multiple VMs (thus the SCs contained therein) on the same node by providing the necessary individual temporal guarantees.
In order to appropriately configure the VM layer scheduler, the overall requirements of the bundle SC plus guest OS need to be considered. Such information is planned to be gathered through the Monitoring and Metering Services that will be available within ISONI, which will have the capability to benchmark VMUs from a "black-box" perspective. This constitutes a simplification with respect to the second approach (see below).
Another potential problem is that, when deploying multiple services on multiple VMs on the same physical node, it may be necessary to specify a static partitioning/allocation of the available resources to the virtualization layer. This may become somewhat restrictive for those scenarios in which the requirements of a service exhibit substantial dynamic variations over time, like it happens typically in multimedia due to the use of compression technologies.
Major advantages of the approach are:

- an enhanced security level, because data of different service instances are isolated within a VMU domain;
- the capability to migrate SCs to alternate locations, independently from one another;
- simpler SC management infrastructure, because VMUs may be always considered as black-boxes, for example for monitoring and benchmarking purposes.
- As due to the current status of the project, this approach seems the one that is believed to be the most appropriate for being implemented within the IRMOS platform (see also the considerations relative to the second approach below).

### SCs in the same VMU

In such an approach, a *single OS instance* may be shared between multiple service instances that run onto the same Physical Host.
Major advantage of such approach is that, in case of multiple instances of the same kind of service, or of instances of different services that have similar requirements in terms of run-time software components and libraries that need to be accessible on the OS, they may run within the same OS instance (running within a VMU). This results in a better usage of memory and computational resources on the physical node due to reduced per-SC overheads.
Furthermore, for those activities that exhibit fluctuations in time of the workload (as it happens typically with live multimedia streaming), in the context of the OS scheduler it may be easier to adopt flexible scheduling strategies that allow for dynamic adaptation of the scheduling parameters for the running activities depending on the instantaneous measured workload.
Major drawbacks of such an approach are:

- a decrease in the security level, because data of independent services are managed within the same OS instance (but still the OS security services may guarantee an acceptable security level for applications);

- in case of necessity, it is only possible to migrate the entire VMU to another location, along with all of the service instances it is hosting;
- Metering and Monitoring Services would need to be capable of gathering information on the individual SCs running within a VMU, thus they would need to interact with the contained OS in order to gather such information (as opposed to gathering it directly from the VMM layer, as measured on the VMU as a whole);
- the dynamic instantiation of a new SC within a VMU that is already running other SCs might potentially require:
  - a dynamic re-modulation of the scheduling guarantees that are assigned to the VMU as a whole;
  - dynamic changes on the topology of the VSN the VMU belongs to.

The latter issue would introduce a need for renegotiation of Technical SLAs, something that, from a preliminary analysis, seems quite complex to manage, as T-SLAs will be more oriented towards a static allocation of resources for the T-SLA duration. However, extensibility of T-SLAs is a feature still under discussion within the Consortium.

Furthermore, in such an approach, the interactions between the scheduler of the virtualization layer and the scheduler of the OS should be appropriately taken into account. Basically, the situation is similar to the so called *hierarchical scheduling*, a feature that is commonly found in real-time OSes. The difference here is that the two levels of scheduling are not done within the same scheduler of an OS, but result from the interaction between the scheduler of the virtualization layer and the one(s) of the hosted OS(es). As far as the first scheduler is capable of providing temporal guarantees to the hosted VMs, it is still possible to use a OS scheduler that provides temporal guarantees to the individual activities.

Furthermore, in such an approach, two important issues need to be considered, both arising from the use of virtualization on the physical nodes:

1) the OS scheduler cannot have direct access to the underlying hardware, but it is mediated by the virtualization layer;
2) each OS instance hosted onto a node shares the physical node with possibly other OS instances.

Both of these factors need to be appropriately taken into account when designing the (guest) OS scheduler. For example, a scheduler is realized mainly in terms of setting timers. These will not be settable directly on the bare hardware, but only through the virtualization layer. This is expected to result in a decrease of the precision with which a time-sensitive OS scheduler may be realized, thus a lower precision in providing temporal guarantees to individual running services. The extent of such precision loss will have to be carefully evaluated in order to assess the suitability of the approach for the applications targeted in the project.

Concerning the second point above, its importance will mainly be related to the final set-up and architecture of a physical node in terms of how many OS instances would be allowed to run in the case of an OS with RT-scheduling capabilities.

*Due to these issues, as of now, this option for scheduling SCs is not planned to be implemented within IRMOS, even if further theoretical investigations on its feasibility may be carried on within the project.*

Also, it is worth to mention that, even with an implementation supporting exclusively purely static SLAs within IRMOS, one could imagine to deploy a “special” VSN for the purpose of deploying dynamically “special” SCs inside it, possibly pushing multiple

"special" SCs within the same VMU. This would be possible by "renting" an appropriately over-provisioned VSN from an IRMOS provider, with general-purpose VMUs configured with as much software as may be needed by "special" SCs to be instantiated, and as many links as foreseen to be needed. Such a provider could manage to exploit the gained increased efficiency in managing resources in order to achieve lower rates for offered services, but the level of QoS that would be achievable in such a case should be further investigated.

### Impact of co-scheduling on SC performance

In order to appropriately allocate available resources to SCs, it is important to consider reliable estimates of the performance of a given SC, when deployed on a given PH. Such issue is mainly outside of the WP6 scope, as it involves mainly the adoption of appropriate techniques for benchmarking, workflow-based application modelling and performance estimations. However, from a real-time scheduling perspective, it is worth to mention that the performance exhibited by a SC when scheduled with fixed real-time reservation parameters may be quite different depending on what other activities are co-scheduled at the same time on the same PH. The timing interference, in such a case, is not due to the classical scheduling overheads due to kernel-code that needs to be executed in order to carry on context switches among processes, because it remains fixed (once scheduling parameters are known), but it is due to changes in the pattern of interferences of the different co-scheduled applications on the cache.
Therefore, a preliminary study has been done in WP6 for the purpose of evaluating what is the impact of co-scheduling multiple applications on the same PH, on the performance exhibited by the co-scheduled applications, obtained at varying characteristics of the serviced applications, in terms of size of accessed data.
To this purpose, a simple experimental set-up has been considered, where two batch tasks have been executed on the same PH, scheduled with a fixed Round-Robin policy, under Linux. The task under consideration was a simple matrix inversion (but a similar experiment has been repeated with matrix exponentiation typical of physical simulations), whereas the data size variations have been obtained varying the size of the computed matrix.
With such a set-up, the execution times for completing the inversion of a random matrix with a given size have been measured multiple times, while the co-scheduled task was changed so as to manipulate matrices of different sizes. Note that the two tasks were exclusively CPU-intensive without I/O operations, therefore they were always active during the experiment and could not block. Well, it has been confirmed that changing the second task data-set size influenced the execution time on the first task. As depicted in Figure 14, the execution time of the first task (with a fixed data-set size) was measured to vary from about 3,5 seconds when co-scheduled with an application with a nearly null data-set size (the `yes` program, launched so as to not produce any output), to over 5 seconds when co-scheduled with a matrix inversion operation involving a very large matrix (4 million elements).

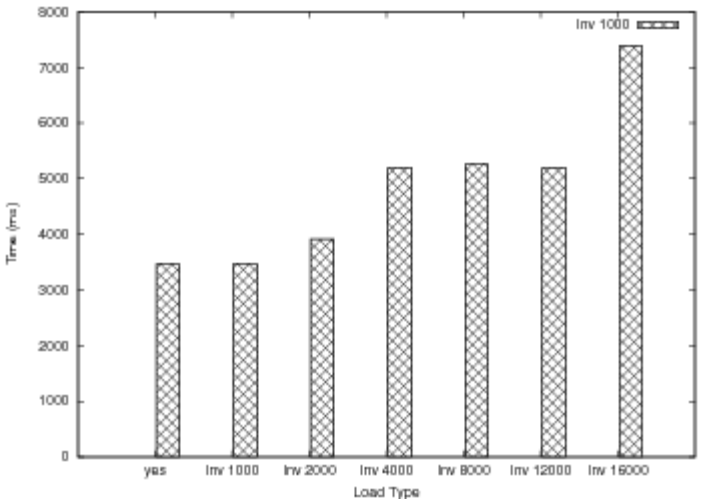


**Figure 14. Execution time obtained at varying data size of the co-scheduled task.**

Of course, the significance of the gathered execution times variability needs to be carefully considered with respect to the expected behaviour of real IRMOS applications. However, it is expected that a simple way to deal with such issues during the SC deployment and resources allocation would be to adopt an appropriate level of resource over-provisioning, in order to avoid the risks related to a possible SLA violation due to the just mentioned phenomenon. However, this topic needs further investigation.

### 5.3.4 Preliminary integration ideas

The low-level (kernel-level) components that are needed for the management of physical resources, such as a custom CPU scheduler in the case of computing elements, will need to be integrated with higher-level services belonging to the IRMOS platform.
Focusing on the KVM virtualization layer, the envisioned software components that might support real-time CPU scheduling in IRMOS are depicted in Figure 15.
According to such an approach, in order to support real-time scheduling for the management of the available CPUs on a Physical Host, it is planned to develop:

- a modified Linux kernel with a real-time scheduling policy specifically tailored on the IRMOS needs (see Section 5.3.2);
- a User-space Interface allowing for the use and configuration of the developed real-time capabilities of the kernel, probably in form of a dynamically linkable library, that will communicate with the real-time kernel scheduler with one of the usual mechanisms that are available for such purposes:
  - addition of new system calls, or modification to the semantics of existing system calls (i.e., the `sched_setscheduler()` call);
  - virtual device in the `/dev` filesystem, used by calling the `ioctl()` system call;
  - entries in the `/cgroup` virtual filesystem (see Section 5.2.1);
- a Real-Time Virtualization layer, allowing for the launch of VMUs with real-time scheduling guarantees and temporal encapsulation: this would come probably in the form of a set of scripts and/or command-line applications that use KVM in conjunction with the user-space interface to real-time scheduling;
- SOA-based components exposing the real-time capabilities of an EE to the overall IRMOS infrastructure: the capability of an OS to provide temporal guarantees to

individual service instances (i.e. the capability for its resources to be "sliceable") should be exposed in a SOA fashion to the other components of the IRMOS platform.

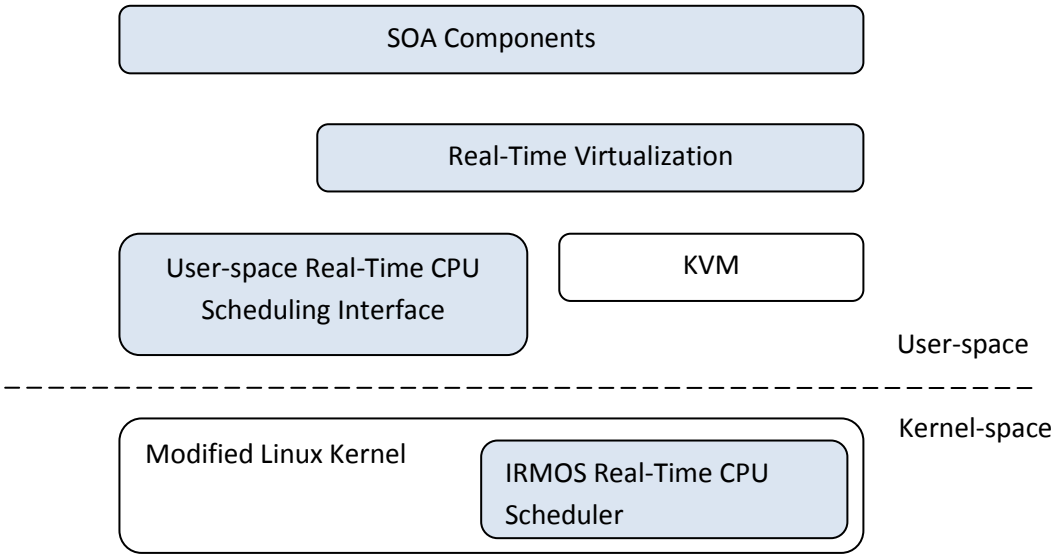


**Figure 15. Preliminary envisioned architecture for supporting Real-Time CPU scheduling within an IRMOS EE.**

Note that the latter component may also be available without necessarily being in conjunction with virtualization, if needed, i.e., it might allow the allocation of "slices" of the available computing power to other node-level activities residing on the host OS, that are not necessarily embedded within VMUs.
As a final remark, note that the considerations made in this section have to be considered as very preliminary, given the current status of the project, and these topics will be details in much greater detail in future deliverables, such as D6.4.2.

## 5.4 Storage

Providing real-time guarantees to end-to-end distributed applications requires the capability of guaranteeing certain pre-agreed QoS levels to individual data flows that are being streamlined from a storage device. While remanding to specific deliverables the full details about the storage architecture of IRMOS nodes, this section briefly summarises the basic concepts and ideas by which, at the moment, such capability is planned to be provided within the project.

### 5.4.1 Object-based Storage Devices (OSD)

Object-based Storage Devices [1] (OSD) is the next-generation reference standard for storage, designed to overcome the limitations of current storage technology, due to the interface by which they are accessed.

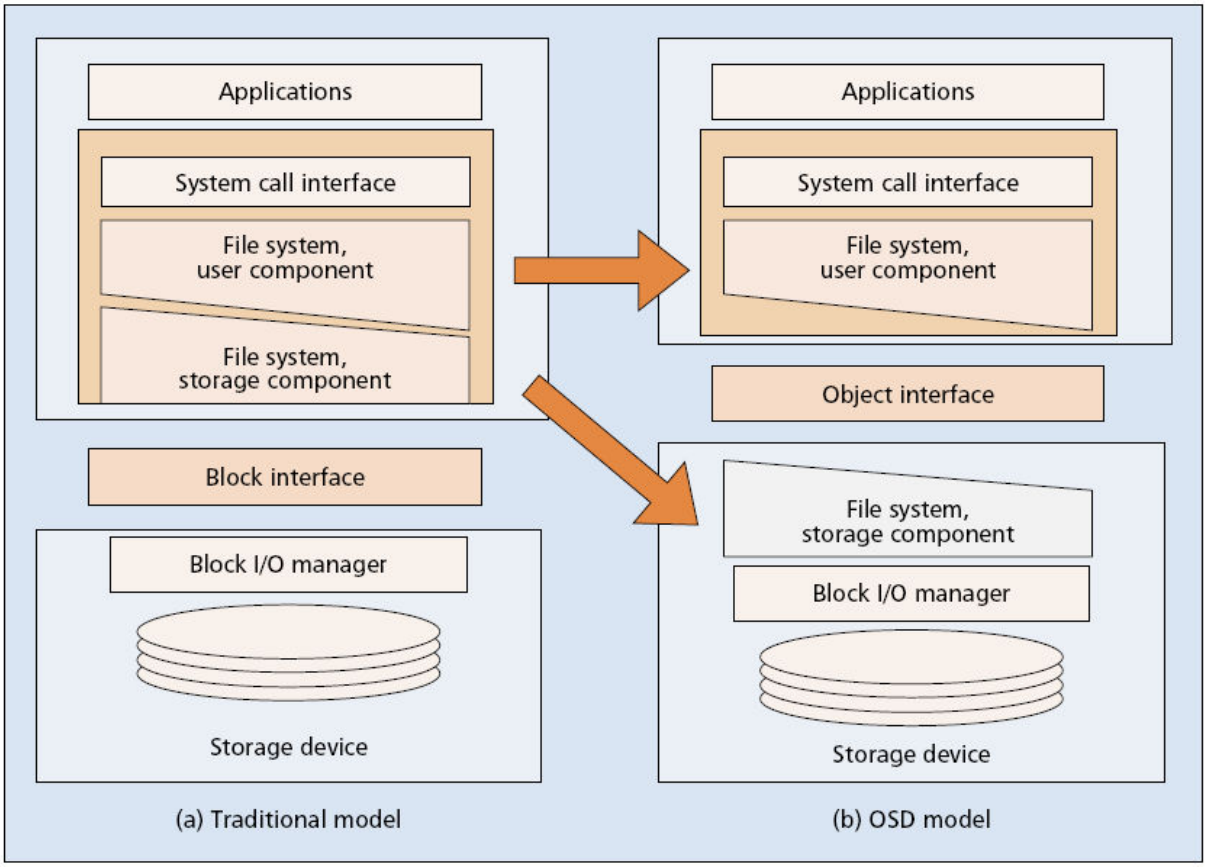


**Figure 16. Object-based Storage Devices, from [1].**

In fact, the existing technologies in use, such as Direct-Attached Storage (DAS), Network-Attached Storage (NAS), and Storage Area Networks (SANs), basically allow for either a secure cross-platform data sharing at the file level, with limitations in performance due to the file server, or they allow for a direct block-level access by clients, with limitations in terms of security and access control mechanisms. OSD was born for providing a new interface for a scalable, high-performance, cross-platform, secure data sharing architecture.

The OSD model foresees a concept of object that is composed of data, user-accessible attributes, and device-managed meta-data. The data stored in an object is opaque to the object-based device. The user-accessible attributes describe both opaque and non-opaque characteristics of the object. As highlighted in Figure 16 (from [1]), the OSD architecture offloads space allocation from storage applications, so that applications need only to deal with what kind of access to what objects they need, while the OSD infrastructure is capable of managing the access at the block-level, so as to achieve the maximum possible performance.

### QoS Support in OSD

While the current specification for SCSI OSD does not directly address the problem of providing QoS to concurrent streams accessing a device, it allows for the configuration of optional attributes. These attributes can be used to create a QoS management system. For example, they may be used for specifying the maximum latency or minimum

throughput that are required by an application in order to exhibit the expected run-time behaviour.
Storage systems provide QoS acting on two distinct layers, filtering the requests coming to the managed devices (a.k.a., throttling) and scheduling the requests once they are ready to be dispatched to the device itself. The usage of OSDs allows for a much easier control and classification of the high level activity per each device, making it possible to easily associate the requests reaching the lower level scheduler to homogeneous streams, simplifying and making more effective the job of the low level scheduler.

## 5.5 Network

Scheduling mechanisms for networking aimed at providing temporal isolation and QoS guarantees across multiple concurrent streams sharing one or more physical link segments within IRMOS/ISONI is mainly the scope of the WP7 Project Deliverables, and fall within the category of real-time support to the IXB.
Detailing, the IXB functionality with respect to connectivity is described in the Project Deliverable D7.1.1 [6], which describes the traffic treatment for interconnecting VMUs of VSNs. The network scheduling strategies and queuing disciplines (shaping, policing, …) concerning network QoS issues are going to be described in the upcoming Project Deliverable D7.3.1. The IXB functionalities described in both deliverables are IXB real-time relevant.
On the other hand, the Project Deliverable D7.2.1 will describe the PathManager functionality, which is related to ISONI framework functions, which do not have real-time requirements.

# 6 Conclusions

This document discussed the main issues that have been identified for the realization of mechanisms, within the IRMOS platform, that will allow for predictable performance of the hosted services and applications, on top of the virtualized Service Oriented Infrastructure environment provided by ISONI within IRMOS.

Even if the achievement of such a goal encompasses both computing, networking and storage technological areas, this document mainly focused on the computing area, as further ongoing work on storage and networking will be reported in future WP6 and WP7 deliverables, respectively. With this regards, real-time CPU scheduling constitutes a fundamental building block (along with QoS support at the networking and storage level) over which it is possible to build predictable response times for real-time applications run within the envisioned IRMOS platform. This is essential for the purpose of predicting the optimum resource allocation needed by an application that is about to be started, and for respecting the established SLA with the final user.

This document first focused on the identification of the concurrently running activities that are consuming compute power in an IRMOS Execution Environment, and these have been enumerated and classified into three main categories. Then, it proceeded to briefly overview which virtualization layers might be interesting to be used within IRMOS. The identification of the virtualization layer is essential for designing the scheduling mechanism that is needed for ensuring appropriate performance guarantees to concurrently co-scheduled VMUs. As of now, KVM seems the most appropriate tool to base IRMOS development upon, and it is well-integrated within the Linux kernel mainstream features.

Focusing on the KVM virtualization approach, in which VMUs run essentially as Linux processes, this document overviewed issues related to the realization of real-time scheduling policies within the Linux kernel, and reported about the design and development efforts undertaken within the project so far.

Also, the current roadmap for future development in the area of CPU scheduling has been presented, also showing in a very preliminary way how the components that are planned in tasks T6.4 and T6.5 of WP6 are expected to fit within the IRMOS overall architecture.

This document will be supplemented by the second version in deliverable D6.4.2, in which the feasibility of the proposed approach will be further investigated thanks to extensive experimental evaluations made on real prototypes.

# 7 References

[1] Thomas Voith, Markus Kessler, Dominik Lamp, Karsten Oberle, *ISONI Whitepaper*, IRMOS Project Website, September 2008

[2] Mike Mesnier, Gregory R. Ganger, Erik Riedel, *Object-Based Storage*, IEEE Communications Magazine, v.41 n.8 pp 84-90, August 2003

[3] IRMOS WP2 Market and Technical Requirements Analysis – D2.1.1 Initial version of Requirements Analysis Report, June 2008

[4] IRMOS WP2 Market and Technical Requirements Analysis – ID2.1.2 Updated version of Requirements Analysis Report, October 2008

[5] IRMOS WP6 Execution Environment – D6.1.1 Formal description language for application requirements of Execution Environment, October 2008

[6] IRMOS WP7 Intelligent Networking – D7.1.1 ISONI addressing schemes, November 2008

[7] Ian Pratt, XEN and the Art of Virtualization, 2006

[8] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, Eoin Hyden, *The design and implementation of an operating system to support distributed multimedia applications*, IEEE Journal of Selected Areas in Communications, vol. 14, no. 7, pp. 1280-1297, 1996. Available online at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.6923

[9] IRMOS WP2 Market and Technical Requirements Analysis – D2.3.1 State of the Art on IRMOS technologies, July 2008

[10] *Real-Time Group Scheduling*, Official Linux Kernel Documentation. Available in current kernel distribution archive as Documentation/scheduler/sched-rt-group.txt, and at: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git

[11] Dario Faggioli, Giuseppe Lipari, Tommaso Cucinotta, *An Efficient Implementation of the BandWidth Inheritance Protocol for Handling Hard and Soft Real-Time Applications in the Linux Kernel*, in Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008), Prague, Czech Republic, July 2008

[12] Gerardo Lamastra, Giuseppe Lipari, Luca Abeni, *A Bandwidth Inheritance Algorithm for Real-Time Task Synchronization in Open Systems*, in IEEE Proceedings of the Real-Time Systems Symposium, London (UK), December 2001

[13] Rodrigo Santos, Giuseppe Lipari, J. Santos, *Improving the schedulability of soft real-time open dynamic systems: The inheritor is actually a debtor*, Journal of Systems and Software, vol. 81, pagg. 1093 – 1104, 2008

[14] Paul Menage, *CGROUPS*, Official Linux Kernel Documentation, available in current kernel distribution archive as Documentation/cgroups/cgroups.txt, also available at the URL: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git

[15] Giorgio C. Buttazzo, *Hard Real-time Computing Systems Predictable Scheduling Algorithms and Applications*, Springer-Verlag New York Inc., November 2004, ISBN: 0387231374

[16] Dario Faggioli, Antonio Mancina, Fabio Checconi, G. Lipari, *Design and Implementation of a POSIX Compliant Sporadic Server for the Linux Kernel*, 10th Real-Time Linux Workshop, Colotlán, Jalisco, Mexico, Oct, 29th – Nov, 1st, 2008

[17] IEEE Standard for Information Technology – Portable Operating System Interface (POSIX). Available online at: http://www.opengroup.org/onlinepubs/009695399

[18] James H. Anderson, *Pfair scheduling: Beyond periodic task systems*, in Proc. of the 7th International Conference on Real-Time Computing Systems and Applications, 2000. Online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.7464

[19] Philip Holman, James H. Anderson, *The staggered model: Improving the practicality of Pfair scheduling*, 24th Real-Time Systems Symposium – Work In Progress Session, Cancun, Mexico, December 3rd, 2003

[20] Björn Brandenburg, John M. Calandrino, James H. Anderson, *On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study*, Real-Time Systems Symposium, Barcelona, 2008

[21] Samuel K. Moore, Multicore is Bad News for Supercomputers – Adding cores slows data-intensive applications, IEEE Spectrum, December 2008

[22] Tommaso Cucinotta, Gaetano Anastasi, Luca Abeni, *Real-Time Virtual Machines*, in Proceedings of the 29th IEEE Real-Time Systems Symposium – Work In Progress Session, November 30th - December 3rd, 2008, Barcelona, Spain

[23] U. Schwiegelshohn, P.Wieder and R. Yahyapour, *Resource Management for Future Generation Grids. In Future Generation Grids*, V. Getov, D. Laforenza, A. Reinefeld (Eds.), Springer, CoreGrid Series, 2005.

[24] S. McGough, A. Afzal, J. Darlington, N. Furmento, A. Mayer, and L. Young, *Making the Grid Predictable through Reservations and Performance Modelling*, The Computer Journal, 48(3):358-368, 2005.

[25] U. Farooq, S. Majumdar, E. W. Parsons, *Efficiently Scheduling Advance Reservations in Grids*, Technical Report, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, August 2005.

[26] A. Sulistio, R. Buyya, *A Grid Simulation Infrastructure Supporting Advance Reservation*, in the Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems, Cambridge, Boston, USA, November 2004.

[27] W. Smith, I. Foster, V. Taylor, *Scheduling with Advanced Reservations*, in the Proceedings of the IEEE/ACM 14th International Parallel and Distributed Processing Symposium, Cancun, Mexico, May 2000.

[28] Linux Testbed for Multiprocessor Scheduling in Real-Time Systems (LITMUSRT), http://www.cs.unc.edu/~anderson/litmus-rt.

[29] KVM: Kernel-based Virtualization Driver, Whitepaper, Qumranet 2006, available at the URL: http://docs.huihoo.com/kvm/kvm-white-paper.pdf.

[30] L. Palopoli, T. Cucinotta, L. Marzario, G. Lipari, AQuoSA - Adaptive Quality of Service Architecture, Software: Practice and Experience, Vol. 39, No. 1, pg. 1-31, 2009.

[31] *Framework for Real-Time embedded Systems based on Contracts (FRESCOR)*, European Project No. FP6/2005/IST/5-034026, http://www.frescor.org.

[32] Virtualization Overview, VMWare Whitepaper, available at the URL: http://www.vmware.com/pdf/virtualization.pdf.

[33] *Windows Server Virtualization – An Overview*, Microsoft Corporation, May 2006, http://download.microsoft.com/download/3/2/2/32212eab-a431-4cd4-8567-cf951b1322de/Virtualization.doc.

[34] Robert P. Goldberg, *Survey of Virtual Machine Research*, IEEE Computer Magazine, 7(6):34—45, 1974

[35] Keith Adams, *A comparison of software and hardware techniques for x86 virtualization*, in Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), pg. 2—13, October 21st – 25th, 2006, San Jose, California, USA